

Dynamic Graphical Models and the Graphical Models Toolkit (GMTK)

Jeff A. Bilmes, University of Washington Seattle
<http://melodi.ee.washington.edu/~bilmes>

August 16, 2020

Contents

I	Introduction and Background	13
0.1	What is GMTK? What is this document?	15
0.2	GMTK Feature Overview: Is GMTK for me?	15
0.2.1	User level GMTK Features	16
0.2.2	Advanced low-level GMTK Features	20
0.2.3	What GMTK doesn't support (yet)	21
0.3	Acknowledgments	22
0.4	Where to report bugs?	23
0.5	How to cite this document and GMTK in general?	23
0.6	Notation	23
0.7	Graph Notation and Terminology	26
0.8	For Further Reading	26
0.8.1	Background in Speech Recognition and Language Processing	26
0.8.2	Background in Graphical Models	27
0.8.3	Hidden Markov Models: HMMs	27
0.8.4	Discriminative Parameter Training in HMMs	27
0.8.5	Explicit Control of ASR system via Bayesian Networks	27
0.8.6	Latent Modeling	27
0.8.7	Observation Modeling	28
0.8.8	Miscellaneous Topics	28
0.8.9	General Structure Learning	29
0.8.10	Quick List: Dynamic Bayesian Network References	29
II	Overview of Graphical Models	31
1	Introduction to Graphical Models: Its a Family Affair	33
1.1	Introduction	33
1.2	Conditional Independence	40
1.3	Examples of Graphical Models	43
1.3.1	Basic Models	45
1.3.2	Principle Component Analysis and its generalizations	46
1.3.3	Discriminative models LDA/QDA/MDA/QMDA	48
1.3.4	Grid models	50
1.3.5	Dynamic models	50
2	Graphical Models Overview	51
2.1	Families and sub-Families	51
2.2	What are graphical models anyway?	52

2.2.1	Static Graphical Models	53
2.2.2	Gaussian Graphical Models	61
2.2.2.1	Gaussians and Independence	63
2.2.2.2	Partitioned Matrices and the Schur Complement	64
2.2.2.3	Gaussians and Conditional Independence	65
2.2.2.4	Gaussians and Bayesian Networks	68
2.2.3	Gaussian Parameterization and Sufficient Statistics	71
2.3	What families are not representable using graphs?	72
3	Undirected Graphical Models	73
3.1	What is a graphical model?	73
3.2	Markov Random Fields - a form of undirected model	77
3.3	Factor Graphs	87
3.4	Generality and Specificity	88
3.5	Examples	90
4	Directed Graphical Models: Bayesian Networks	91
4.1	Bayesian Networks	91
4.1.1	Examples	101
4.1.2	Specificity	102
4.1.3	Continuous Examples	104
4.2	Dependency networks	104
4.3	Bayesian Networks and Causality	105
4.4	Bayesian Networks with Constraints	105
4.5	Ancestral Graphs	106
4.6	Chain Graphs	106
4.7	Summary	108
5	Evidence in Graphical Models, and Soft and Virtual Evidence	109
5.1	Chapter Overview	109
5.2	Traditional Evidence and Zero Probability Events	110
5.2.1	Evidence as a sample from a statistical process	111
5.2.2	Evidence as probability revision	112
5.2.3	Probability of evidence	113
5.2.4	Posterior Probabilities	114
5.2.5	Most likely (Viterbi) values	114
5.3	Evidence as Smart Sums	114
5.3.1	Computing the probability of evidence	115
5.3.2	Computing the posterior probability of a set of vars	116
5.3.3	Computing the maximum	116
5.4	Scaling Deltas, and Virtual children in a Bayesian network	116
5.4.1	Evidence scores other than unity	117
5.4.1.1	Probability of Evidence	118
5.4.1.2	Posterior Probability	118
5.4.1.3	Most likely assignments	118
5.5	Generalization of Evidence, Uncertain Evidence, and Virtual Evidence	118
5.5.1	Why only the ratios matter?	119
5.5.1.1	Score and Probability of virtual evidence	120

5.5.1.2	Posterior probability of hidden variables	122
5.5.1.3	The Variable Assignment with the Maximum Score: Viterbi	122
5.5.2	Sample space interpretation of virtual evidence	123
5.5.3	How can we use $P(V = 1 X_E = \bar{x}_E)$ without having $P(V = 1, X_E = \bar{x}_E)$? A philosophy.	124
5.5.4	Virtual Evidence, Bayesian Inference, and Bayesian Reasoning	127
5.5.4.1	Soft Evidence vs. Virtual Evidence	128
5.6	Virtual Evidence, Undirected Graphical Models, and Factor Graphs	129
5.7	Applications	131
5.7.1	Expansion of Bayesian Network Models	131
5.7.2	Virtual Evidence, Hybrid ANN/HMM systems, and hybrid deep neural networks/DGMs	131
5.7.2.1	Prior Normalization of ANN Outputs	132
5.7.2.2	No Normalization of ANN Outputs	134
5.7.2.3	Hybrid ANN/HMM systems as Pearl's virtual evidence	134
5.7.2.4	Generalized Hybrid Deep NNs/Bayesian Networks	136
5.7.3	Virtual Evidence and Backoff-based Language Models	137
5.7.4	Virtual Evidence and the IBM Machine Translation Models	137
5.8	Evidence in Markov random fields	137
5.9	Conclusion and Further Reading	138
6	Inference on Trees, Triangulated Graphs, and Junction Trees	139
6.1	Introduction	139
6.2	Inference on Trees	141
6.3	Morphing from variable elimination to belief propagation	150
6.4	Multiple Tree Queries	154
6.5	Alternate forms of messages - speed/memory issues	158
6.5.1	Dividing out node functions from edge functions	159
6.5.2	Maintaining distinct node separator functions	160
6.5.3	Message passing as state-space traversal	162
6.6	Tree Queries with arbitrary S	162
6.7	Certain Bayesian network queries can be optimized	163
6.8	Non-tree graphs	164
6.8.1	Triangulated Graphs	168
6.8.2	Nonminimal triangulations for big cliques in sparse models	172
6.8.3	Higher order trees and graph triangulation	173
6.8.4	How to identify triangulated graphs	185
6.9	Multiple Queries	187
6.10	From JT back to normal tree	211
6.11	Bayesian networks and chordal graphs	212
6.12	complexity of exact inference	212
6.13	other tricks	213
6.14	Other applications of triangulated graphs	213
6.15	message passing on rings	213
6.16	space-time complexity trade offs	213
7	Overview of Approximate Inference Methods	215

III Overview of Dynamic Graphical Models	217
8 Dynamic Graphical Models	219
8.1 Template Graphical Models	220
8.2 Stochastic Processes, Discrete-time Markov Chains, and Correlation	222
8.3 Markov Chains	223
8.3.0.1 Higher order Markov chains/conversion of n^{th} order to first order	224
8.3.0.2 State transition probabilities	225
8.3.0.3 Markov chains: stationarity vs. time-homogeneity	227
8.3.0.4 Chapman-Kolmogorov equations	228
8.4 The Many Faces of Hidden Markov Models	229
8.4.1 HMM as random balls drawn from random urns	229
8.4.2 HMM as Random function of a Markov Chain	230
8.4.3 HMMs as exponential (in T) sized mixture distribution	232
8.4.4 HMMs as Stochastic Finite State Automata	234
8.4.4.1 Deterministic FSAs	235
8.4.4.2 Non-deterministic FSAs	236
8.4.4.3 Epsilon Arcs	237
8.4.4.4 Transducers FSAs	238
8.4.4.5 Weighted or Stochastic FSAs	239
8.4.4.6 Mealy and Moore FSAs	240
8.4.4.7 Push Strategy for HMM Inference	242
8.4.4.7.1 Rabiner vs. Jelinek HMMs:	245
8.4.5 HMMs as a Trellis or a “Lattice”	245
8.4.6 Dynamic Time Warping (DTW)	248
8.4.6.1 Dynamic Time Warping and Dynamic Programming	254
8.4.6.1.1 String to String Alignment:	257
8.4.6.1.2 DTW Discussion:	258
8.4.7 Morphing from DTW to HMM	259
8.4.8 HMMs as smoothed (or regularized) unary potential functions	262
8.4.9 HMM as Graphical Models	264
8.4.10 Rabiner vs. Jelinek HMMs as graphical models	266
8.4.11 HMM inference offline and online, the forward/backward algorithm, Viterbi, etc.	267
8.4.11.1 The HMM Forward Algorithm and Collect Evidence	268
8.4.11.2 The HMM Backward Algorithm and Distribute Evidence	269
8.4.11.3 The α and β initializations	269
8.4.11.4 The HMM Probability of Evidence	270
8.4.11.5 The HMM Probabilistic Queries	270
8.4.11.6 The HMM Probabilistic Queries and Parameter Learning	270
8.4.11.7 The HMM forward/backward recursions vs. the graphical model elimination algorithm	272
8.4.11.8 HMM filtering, smoothing, & prediction as elimination	276
8.4.11.9 HMM clique posteriors	277
8.4.11.10 Viterbi path, Viterbi decoding, and most probable explanation (MPE)	279
8.4.11.11 Other HMM recursions	282
8.4.11.12 HMM scaling issues, numerical dynamic range, underflow, normalization, and log arithmetic	286
8.4.12 Profile HMMs	290

8.5	Computing the k best assignments	290
8.5.1	Finding k -best states in HMMs using R-semi-modules	292
8.5.2	Finding the k best paths	294
8.5.3	From Scores to Paths	296
8.5.3.1	A recursion for computing the backtracking pointers during the forward pass	297
8.5.3.2	After final time T recursion	298
8.5.3.3	Backward pass for k -best backtracking	299
8.5.3.4	Resource Complexity	300
8.5.4	k -best without memory penalty	301
8.5.4.1	General case	307
8.5.5	Analysis	307
8.6	Trading off speed and memory in dynamic inference: the Island algorithm	308
8.6.1	Island Algorithm Analysis	311
8.6.2	Island and k -best	313
8.7	What HMMs can do	314
8.8	Conditional Random Fields (CRFs)	314
8.8.1	The class of HMM models	314
8.8.2	Directed and Undirected HMMs are equivalent	315
8.8.3	When might one wish to use directed vs. undirected models?	318
8.8.4	The class of CRF models	320
8.8.5	Generative vs. Discriminative models for classification	321
8.8.6	HMMs are identical to CRFs	323
8.8.7	Observations dependent on pairs of state variables	325
8.8.8	The observations and temporal integration	326
8.8.9	HMM delta features, and why they help speech recognition performance	327
8.8.10	The observations and parametric form	329
8.8.11	CRFs, Convexity, and Training	330
8.8.12	The label-bias problem	331
8.8.12.1	Conditional Markov Models (CMMs)	331
8.8.12.2	Label Bias and CMMs	333
8.8.12.3	Information Theoretic Description of Label Bias in CMMs	336
8.8.12.4	The label-bias problem and HMMs	337
8.8.12.5	The label-bias problem and hybrid Deep MLPs/Markov chains	337
8.8.13	What about the observation-bias problem?	338
8.8.14	How to overcome label/observation bias.	340
8.8.15	Are CRFs better than HMMs? Are HMMs better than CRFs?	341
8.8.16	Why ever use generative models?	342
8.8.16.0.1	It is conceptually easy to think of problems generatively:	344
8.8.16.0.2	Unsupervised learning, clustering, data interpretation, inducing hidden causes in data:	345
8.8.16.0.3	The decoding problem is easy:	345
8.8.16.0.4	Generative decomposability:	345
8.8.16.0.5	Parameter sharing/tying is easy with generative models:	346
8.8.16.0.6	Generative models might require less training data to do well:	346
8.8.16.0.7	Rarity of events:	346
8.8.16.0.8	Dealing with missing information:	346
8.8.16.0.9	Generative models may be trained discriminatively:	347

8.8.16.0.10	Generative models can be structured discriminatively:	347
8.8.16.0.11	Class skew/imbalance:	347
8.8.16.0.12	Dependence between the parameters of $p(y)$ and $p(x y)$:	348
8.8.16.0.13	Brown's argument:	348
8.8.16.0.14	Initial Parameters for Discriminative Models:	349
8.8.16.0.15	Engineering/real world advantages:	350
8.8.17	Why use discriminative models?	350
8.9	Factored and Structured State Spaces: Vectorial HMM	351
8.9.1	Factorial HMMs	353
8.9.2	Hierarchical HMMs	355
8.10	Dynamic Bayesian Networks (DBNs)	357
8.11	Dynamic Graphical Models	361
8.11.1	Generalized Dynamic Bayesian Network Templates: Prologue, Chunk, and Epilogue	362
8.11.2	Generalized DGM template examples	364
8.11.3	Formal description of generalized DBN template	369
8.11.4	DGM Template Validity	371
8.12	Other topics	376
8.12.1	HMMs can represent many DGMS	376
8.12.2	Reasons for using DGMS rather than HMMs	377
8.12.3	Additional generic examples of DBNs/DGMs	378
8.13	Structured Prediction in Machine Learning	383
8.14	The problem with the Viterbi/MPE/MAP assignment in DGMs	385

IV Applications of Dynamic Graphical Models 389

9	Dynamic Graphical Model Examples for Automatic Speech Recognition	391
9.1	Graphical Model Speech Architectures	391
9.1.1	Basic phone-based bi-gram decoding structure	392
9.1.2	Basic phone-based tri-gram decoding architecture	395
9.1.3	Cross-word tri-phone architecture	396
9.1.4	Tree-structured lexicon architecture	398
9.1.5	Transition Explicit Model	399
9.1.6	Multi-observation & multi-hidden stream semi-asynchronous architecture.	399
9.1.7	Architectures over Observed Variables	400
9.1.8	Polyphase speech recognition	400
9.1.9	Architecture Summary	402
10	Dynamic Graphical Model Examples for Natural Language Processing	403
11	Dynamic Graphical Model Examples for Computational Biology	405
11.1	Example: Segment Modeling in Bioinformatics	405
12	Dynamic Graphical Model Examples for Activity Recognition	407

V Inference In Dynamic Graphical Models	409
13 Inference In Dynamic Graphical Models	411
13.1 Inference in HMMs: The Search Space	412
13.2 Inference in DGMs	417
13.3 Graphical aspects of inference in DGMs	418
13.3.1 The Case of Unroll and Compute	418
13.3.1.1 Unroll and Compute with the Factorial HMM	419
13.3.1.2 Unroll and Compute with a more complex example	420
13.3.1.3 Where slice-by-slice elimination fails	421
13.3.2 Finding Periodicity	425
13.3.2.1 Modified Sections	425
13.3.2.2 Left Interface	426
13.3.2.3 Right Interface	429
13.3.2.4 Left or Right Interface, which to use?	430
13.3.2.5 Examples of left and right Interfaces	431
13.3.2.6 Cases where left and right interface are different	433
13.3.2.7 When neither the left nor right interface is optimal, but another “interface” is.	437
13.3.2.8 Where no default interface is optimal	442
13.3.2.9 The M (vertex cut span) and S (chunk skip) parameters, and template restrictions	443
13.3.2.10 Computing the optimal vertex cut	448
13.3.2.11 Finding M , the vertex cut span	451
13.3.2.12 The chunk skip parameter S	451
13.3.3 Further Template Restrictions	452
13.3.4 Higher order models: recovering the lost Markov property	456
13.3.5 Online inference, Kalman smoothing and filtering and sections and interfaces	460
13.3.6 Cases where no monolithic interface is optimal for original template: factored interfaces	460
13.3.7 Summary	461
13.4 Inference strategies for DGM sections	461
13.4.1 Generic Messages in a DGM Junction Tree	463
13.4.2 Clique-based Limited Extent Asynchrony	466
13.4.3 Sparsity Preserving Search-based Message Passing	468
13.4.4 Exploiting Commonality	472
13.4.5 Approximate inference in DGMs	472
13.4.5.1 Exploiting sparsity via pruning	472
13.4.5.2 Sampling approaches	476
13.4.5.3 Factored interfaces, and assumed-density filtering approaches	477
13.4.5.4 Multi-pass and Coarse-to-fine approaches	477
13.4.5.5 Static inference	478
13.5 Mixed max/sum decoding: semi-ring mixing	479
13.5.1 A class of approximation algorithms	482
13.6 Finite Precision Numerics and DGMs	483
13.7 Discussion	483

VI GMTK Toolkit	485
14 Toolkit Overview	487
14.1 GMTK’s “Hello World”	487
14.1.1 Single variable “Hello World”	487
14.1.2 Two-frame HMM-based “Hello World”	490
14.2 Using GMTK: The process of specifying, training, and then using a model	494
15 Representing Structure in GMTK	497
15.1 GMTKL: The GMTK Structure Language	497
15.1.1 GMTK’s Random Variables	499
15.1.2 RV Dependencies: Parents and Children	502
15.1.3 Random Variable Weights	503
15.1.4 Switching Parents and Dependencies	508
15.1.5 Switching Weights	511
15.1.5.1 Word Insertion Penalties, and their Generalization	512
15.1.6 GMTKL Templates and Unrolling	513
15.1.6.1 Templates and Multi-Rate/Multi-Scale Processing	515
16 Representing Parameters in GMTK	521
16.1 Numerical vs. Non-Numerical Parameters	521
16.2 The GMTK basic parameter “object”	521
16.3 ASCII/Binary files, Preprocessing, and Include Files	522
16.4 Input/Output Parameter Files	525
16.4.1 Input Master File	525
16.4.2 Output Master File	526
16.4.3 Special Input/Output Trainable File	527
17 GMTK Representation Objects	529
17.1 Collection Objects	529
17.2 Conditional Probability Tables CPTs	531
17.2.1 Dense CPTs	531
17.2.2 Special Internal Unity Score CPT	533
17.2.3 Sparse CPTs	534
17.2.4 GMTK’s Deterministic Integer Maps: Decision Trees and C++ Integer Maps	540
17.2.5 Decision Trees with Leaf Integer Formula	541
17.2.5.1 Lists of decision trees: decision tree files	545
17.2.5.2 Leaf Node Integer Formula	546
17.2.5.3 Per-Segment Decision Trees	547
17.2.5.4 Uses of Decision Trees	550
17.2.6 Compiled C++ functions as deterministic maps	551
17.2.7 Built in deterministic mappings	551
17.2.8 Deterministic CPTs	552
17.3 Simple 1-D distributions: SPMFs and DPMFs	553
17.3.1 Dense probability mass functions (DPMFs)	554
17.3.2 Sparse probability mass functions (SPMFs)	554
17.4 Gaussians and Gaussian Mixtures	554
17.4.1 Mean and Diagonal-Covariance Vectors	555

17.4.2 Gaussian Components	555
17.4.3 Special Internal Names: Zero and Unity Score Components	556
17.4.4 Mixtures of Gaussians	556
17.5 Dlink matrices and structures	557
17.5.1 Full-covariance Gaussians	558
17.5.2 Banded Diagonal and/or Sparse Factored Inverse Covariance Matrices	562
17.5.3 Sparse Global B Matrix	564
17.5.4 Buried Markov Models with Linear Dependencies	565
17.6 GMTK Parameter Sharing/Tying	568
17.6.1 GEM training and parameter Sharing/Tying	573
17.7 Virtual Evidence (VE) and time-inhomogeneous VECPTs	573
17.8 Deep Models, Virtual Evidence (VE) and DeepVECPTs	573
17.9 The Global Observation Matrix	573
17.9.1 Dlink Structures and the Global Observation Matrix	574
17.9.2 Multiple Files and the Global Observation Matrix	575
17.9.3 Multiple Files and the Global Observation Matrix	576
17.10 Virtual Evidence (VE) and VECPTs	576
18 The Main GMTK Programs	577
18.1 Integer range specifications	577
18.2 Observation/feature file formats	578
18.3 Generic Options	579
18.4 gmtkTriangulate	579
18.4.0.1 GMTK Triangulation Search Engine	579
18.5 gmtkEMtrain	580
18.5.1 gmtkEMtrain, EM iterations, and parallel training	587
18.5.2 gmtkEMtrain tips	588
18.5.2.1 Determinants becoming too small	588
18.6 obs-tools	588
18.7 gmtkViterbi	588
18.8 gmtkScore	589
18.9 gmtkSample	590
18.10 gmtkParmConvert	591
19 GMTK Under the hood	593
19.1 Some GMTK error messages	595
19.1.0.1 Warnings about not enough accumulated probability	595
19.1.0.2 Zero clique errors	595
19.1.0.3 Can not successfully compute either a left or a right interface	596
19.2 GMTK System Issues	596
VII Appendices	603
A History and Development Milestones of GMTK	605
B GMTKL Grammar	611

VIII Bibliography **617**

IX Index **647**

Part I

Introduction and Background

0.1 What is GMTK? What is this document?

The Graphical Models Toolkit (GMTK) is an open source, publicly available toolkit that allows a researcher or scientist to rapidly prototype, test, learn, and then use statistical models under the framework of dynamic graphical models (DGMs) and dynamic Bayesian networks (DBNs). GMTK can be used for applications and research in speech and language processing, bioinformatics, activity recognition, econometrics, or any type of offline or streaming time series or sequential data application.

GMTK has many features, including exact and approximate inference; a large variety of built-in factors including dense, sparse, and deterministic conditional probability tables, native support for ARPA backoff-based factors and factored language models (FLMs), parameter sharing, gamma and beta distributions, dense and sparse Gaussian factors, heterogeneous mixtures, deep neural network factors and CPTs, and time-inhomogeneous trellis factors; arbitrary order embedded Markov chains; a GUI-based graph viewer; flexible feature-file support and processing tools (supporting pfiles, HTK files, ASCII/binary, and HDF5 files); parameter learning algorithms; rich graph triangulation engine and exact and approximate dynamic inference procedures; and both offline and streaming online inference methods that can be used for both parameter learning and prediction. More information is available in the documentation. All in all, GMTK offers a flexible, concise, and expressive probabilistic modeling framework with which one may rapidly specify a vast collection of temporal statistical models.

This document describes the graphical models toolkit GMTK, its facilities and its supporting programs. The document also includes a tutorial on graphical models, and dynamic graphical models. DGMs include models such as hidden Markov models, dynamic Bayesian networks, auto-regressive switching Gaussian regression models, structured and/or hidden conditional random fields (CRFs), and so on. Applications of such models include speech recognition and other speech processing tasks (such as keyword spotting, speaker identification and verification, language identification based on acoustics); language processing (such as part of speech tagging, named entity recognition, dialog act tagging); and computational biology applications such as in proteomics (e.g., peptide and protein identification, secondary and higher-level structure prediction) and genomics (unsupervised gene discovery, SNP detection, sequence clustering). GMTK also has been used to produce new multi-string alignment algorithms (e.g., generalizations of Levenshtein distance where the local costs may also be learnt automatically). GMTK has over the years been used by many laboratories internationally and for many different applications.

This document is meant to serve as the primary documentation for GMTK, and includes descriptions of its features and capabilities, its limitations, the various programs associated with GMTK, a FAQ about GMTK, various tutorials that use GMTK, and lastly a reference manual. The document also includes a self-contained tutorial on graphical models and DGMs (including a tutorial on hidden Markov models), and also a section that gives a list of modular DGM subcomponents, techniques, and tips that over the years have repeatedly proven themselves useful for variety of tasks.

Unfortunately, if you are reading this sentence, then this document, even given its current length, is woefully incomplete. There are still incomplete sections, as well as a number of notes and TODOs to myself that take the form of . On the other hand, the document in its current form will still be very useful for understanding graphical models, dynamic graphical models, and methods for taking advantage of GMTK.

0.2 GMTK Feature Overview: Is GMTK for me?

GMTK has many capabilities, most of which are outlined below. This section is written for a user who is interested in potentially using GMTK and who wishes to quickly determine if GMTK is suitable for their needs.

Perhaps the first remark to make is that GMTK is currently meant more as a toolkit (meaning there are

a set of top-level programs that you call from the command line and which are often called from scripts or other programs), rather than a library (where you have an API which you can call from your own front-end C++ code). GMTK it has its own graph specification language and tokenizer/parser, many commands and command-line options, and the idea is that the user does most everything via that mechanism. Of course, given the GMTK source code, the core GMTK routines for inference could easily be linked to and it would be possible to make ones own front-end program, but GMTK is not really set up for that at the moment.

In the next several sections, we list the main user-level features of GMTK, some low-level features that perhaps the expert might be interested in, and lastly list a set of things that GMTK does not currently do. There are other frequently asked questions that you can read in Section ??.

0.2.1 User level GMTK Features

We first describe user-level features.

- Textual DGM Graph Specification Language - we have found that using an ASCII text language to specify the graph is often faster and easier than inputting a graph using a GUI-based interface. Like a programming language, GMTK obeys a textual syntax that the user must follow to specify a graph. The GMTK syntax is simple, but on the other hand there is a lot you can do. In this document, you will see numerous examples of the syntax, including a GMTK “hello world” (see §??).
- GUI-based graph viewer (`gmtkviz`) - On the other hand, once a graph is specified, with this tool one can quickly format it, view it, and print it to an .eps file for inclusion in a paper or talk, to explain it to others, or to recall what it does years later.
- Dense multidimensional discrete conditional probability tables (CPTs) are used to implement directed edges (or V-structures) in a Bayesian network. These are specified as dense tables of the form $p(x_i|x_{\text{parents}(i)})$. The tables grow exponentially big with the number of parents.
- Sparse multidimensional discrete conditional probability tables (sparse CPTs) to implement directed edges (or V-structures) in a Bayesian network. This is done via sparse tables of the form $p(x_i|x_{\text{parents}(i)})$. The sparse tables are implemented by there being an integer mapping from the domain of the set of parents to an integer. This integer indices into a sparse table of value-probability pairs corresponding to the child. The sparse mappings and tables must be specified beforehand, and allow extremely high dimensional CPTs to be expressed using very few parameters, thereby avoiding the exponential parameter growth of dense tables.

A sparse table must be pre-specified in advance to GMTK. While training a dense table might result in a table with lots of zeros, the user must explicitly re-code that dense table into a sparse table using GMTK’s sparse table facilities.

- Deterministic multidimensional discrete conditional probability tables (deterministic CPTs) to implement directed edges (or V-structures) in a Bayesian network as sparse tables of the form $p(x_i|x_{\text{parents}(i)})$. This is an extreme form of sparsity where for every set of parent values there is only one possible child value that has any probability (and it has probability of unity). This is implemented by specifying a deterministic function that maps from the domain of the parent variable values to the child value which has probability one.

Deterministic CPTs are useful for producing sequencers (a chain of random variables that can only sequence through a specific set of values but when a change occurs might be a random event). Also, there are facilities for deterministic CPTs to change depending on the segment (i.e., “iterable decision trees”) which is useful for, say, training a speech recognition system where a word transcription is given for every training utterance, but the underlying training graphical model remains fixed.

- Trellis/Lattice data structures to allow a subset of an exponential number of dynamic trajectories in a discrete state space to be specified in a polynomial (or even linear) amount of space. This uses the “lattice” constructs that are often found in speech recognition, but here they are generalized for use by any dynamic graphical model. Support for HTK lattice file format as well (to take advantage of other lattice manipulation tools that are available). Another way of thinking about a trellis is as a sparse time-inhomogeneous (i.e., time-dependent) CPT that is specified in an efficient data structure, both on disk and in memory.
- Flexible deterministic mapping facilities. For sparse or deterministic tables above (and also in a few other places), one needs to build a deterministic mapper that maps from a set of integers to an integer. This can be done either using interpreted decision trees with integer formulas for leaf nodes (using a C-like syntax but with additional internal high-level functions available). Alternatively, when speed is most important, the deterministic mapping functions can be written in C++ and linked into GMTK. There are a number of useful built-in deterministic mapping functions available

The deterministic mapping facility is also used to map from a set of integer random variable values to an offset in a table of Gaussian mixtures, and is also used to select from a number of cases in the switching parent facility (see below).

- Standard dynamic Bayesian network (DBN) templates, where two frames are specified, each of which specifies nodes and intra-slice edges, and between the two frames is a specification of the inter-slice edges.
- Generalized DBN templates, where you specify a prologue (to occur at the beginning of a sequence), a chunk (to be expanded/unrolled in the model to fill the available length of a sequence), and an epilogue (to occur only at the end of a sequence). Each of the prologue, chunk, and epilogue can be any number of frames long, but it is the (potentially multi-frame) chunk that gets repeated to expand the model to any desired length. Thus, multi-frame and multi-rate models can easily be expressed.
- High order sequential models. I.e., temporal dependencies need not be first order (as in a standard HMM) but can span across more than one frame and more than one chunk. One can, for example, easily represent N -gram language models for very large N (within available computational resources). One can not, however, specify a dependency between a variable in the prologue and a variable in the epilogue.
- Static Networks (in addition to dynamic). While GMTK is optimized for static networks, it is fine to create a static network as well, or even a sequence of disconnected static networks.
- Directed models may have edges pointing either forwards in time, backwards in time, or both directions in time simultaneously (assuming no directed cycles). One can, for example, create coupled bi-directional HMMs.
- Switching Parent Functionality - otherwise known as value-specific or context-specific conditional independence, or “Bayesian multinetworks”, useful as another parametric form of what otherwise would be a very high-dimensional table. For example, one can easily express what would be a three-dimensional table as two two-dimensional tables as in $p(a|b, c) = p(a|b, s = 0)p(s = 0) + p(a|c, s = 1)p(s = 1)$. Here, the variable s acts as a switching parent that switches between a being dependent either only on b (when $s = 0$) or only on c (when $s = 1$).
- Native support for ARPA language models - in speech recognition and language processing, backoff-based language models are quite commonly used where one wants to use an n -gram model of the form $p(w_t|w_{t-1}, w_{t-2}, \dots, w_{t-n+1})$. When n is large, estimating a table of this size is prohibitive. A

standard low-parameter implementation of this CPT developed by the language processing community is to use backoff, and this allows for a wide variety of smoothing methods to be represented (such as Good-Turing, Witten-Bell, Kneser-Ney and so on) in the same framework. There is a standard file format for such models, and GMTK can read them directly into a language model CPT.

- Native support for factored language models (FLMs) - when large tables use heterogeneous random variables (i.e., $p(a|b, c, d, e)$ where the domain alphabets of each of the random variables a, b, c, d, e are different from one another), factored language models (FLMs) can help. They use a similar strategy of backoff as do ARPA language models, but a different file format due to the potential heterogeneity of the random variables. Factored language models are a strict generalization of the ARPA language models mentioned above, since when the variables all share the same alphabet FLMs reduce to ARPA language models. GMTK can directly read FLM files into a FLM CPT.

We note that both of the above language model facilities in GMTK are compatible with those produced by the SRI Language model toolkit (SRILM) and all of its FLM generalized backoff options. In fact, it is even possible to train language models with hidden variables using embedded Viterbi training as was done in [211]).

- Sparse linear conditional Gaussians over real-valued observation vectors - i.e., distributions of the form $p(x|z, q)$ where $x|z$ is a multidimensional Gaussian conditioned on the value z and parameters selected by discrete value q . This allows one to specify standard multi-dimensional Gaussians (i.e., z is empty) with either a diagonal, block-diagonal, sparse, patterned, or full covariance matrix.
- Multidimensional Gamma distributions over real-valued observation vectors. By “multidimensional Gamma”, what is meant is a vector of scalar Gamma distributions, not some of the multidimensional generalizations of scalar Gamma distributions that have appeared in the statistics literature.
- Multidimensional Beta distributions over real-valued observation vectors. By “multidimensional Beta”, what is meant is a vector of scalar Beta distributions, not some of the multidimensional generalizations of scalar Beta distributions that have appeared in the statistics literature.
- Mixtures of multivariate and heterogeneous components. That is, one can create a mixture of components where each component might be of a different type, one might be a standard multivariate Gaussian, another might be a Gamma distribution, another might be a sparse conditional Gaussian, and so on. These can be learned in tandem using the EM algorithm (see below).
- Flexible observation file format support. Including support for plain ASCII text files, raw binary files, ICSI pfiles, HTK files, and HDF5 files. A number of tools (the `obs-` tools) are also included that make it easy to manipulate (e.g., print, merge, process, and create) observation files, as well as do a few simple transformations such as PCA/KLT, Gaussian normalization, silence addition, and so on. Some of this facility was taken from the ICSI pfile utilities, but here it can apply to any of the files that GMTK supports.
- Feature-file processing. That is, feature files can be up and down sampled, merged, concatenated, stretched, and shrunk, all based on command-line arguments to any of the GMTK programs. Multiple streams can be merged into one stream, where processing can be done either separately on each stream or jointly after the streams have been merged together. Individual sets of segments can be selected on the command line. Subsets of frames of individual segments can be selected (e.g., to, say, use every even frame). Individual real-valued features can be temporally filtered (using a specified FIR filter).

- Model parameter transformations to/from HTK, the hidden Markov Model Toolkit (HTK). <http://htk.eng.cam.ac.uk/>. That is, it is possible to train HMMs in GMTK and then use them in HTK, and vice versa.
- Parameter sharing. Different mixtures can share components. Different Gaussian components can share means, or sub-portions of their covariance matrices. Support for training in the shared setting via either an EM or a generalized EM (GEM) training algorithm (selected automatically). For sparse CPTs, different sparse components can share the same set of values.
- Gaussian Vanishing/Splitting algorithm, where components may either “split” or “vanish” during training. This helps automatically determine the number of components appropriate for a mixture, and yields a heuristic for producing essentially a non-parametric model. The schedule of splitting and vanishing is controllable on the command line.
- Parameter training algorithm: Generative training via the EM algorithm and generalized EM in the case of certain parameter sharing options.
- Centered and non-centered l_2 regularization of conditional Gaussian regression coefficients. The centered version corresponds to standard l_2 regularization, and the non-centered version corresponds to a form of parameter adaptation.
- Laplace and Dirichlet smoothing of discrete dense CPTs during training. Note that other forms of smoothing are implicit in the language model and factored language model CPTs mentioned above.
- Both exact and approximate inference. Exact inference is combined with a rich graph triangulation engine (see below). Approximate inference in GMTK uses pruning and sampling methods, which have been quite successful in sequential models. For example, there are many different pruning heuristics in GMTK, including existing and well-known methods (such as beam pruning, histogram or state pruning, percentage pruning, and KL-divergence pruning), and novel pruning methods such as predictive pruning and diversity pruning. Pruning can be done at the clique or at the separator level. A special form of EM pruning is additionally available during EM training.
- Linear and *island* inference algorithm. Standard forward/backward inference can proceed in the standard way having an $O(T)$ time and memory cost, or via the island algorithm that yields a $O(\log T)$ memory cost and $O(T \log T)$ compute cost. Command line options to control the base of the log and the linear section threshold at which we drop down to linear inference. This allows achieving intermediary points between the extremes of $(O(T), O(T))$ time and memory (in the linear inference case), and $(O(T \log T), O(\log T))$ time and memory (in the island case). This is very useful for very long segments commonly found in computational biology applications.
- Viterbi decoding - also called MAP inference. This produces the most likely sequence of hidden variable values given the observations.
- Online inference, for streaming applications. This includes versions of Kalman-style filtering, prediction, and smoothing, but in this case generalized for the DGM case. Flexible streaming options for integrating information from multiple synchronized information streams.
- Virtual evidence CPTs - CPTs can provide scores to inference via the virtual evidence mechanism. The virtual evidence scores can come from the observation file and thus be time dependent. In other words, one can produce time-inhomogeneous CPTs of the form $P_t(C = 1 | \text{parents})$ that change at each frame. This makes it possible, for example, to implement hybrid MLP/HMM, SVN/HMM,

MLP/DBN, Deep MLP/DBN models, where probabilities from, say, an MLP or a deep model are factored into the state of, say, an HMM.

- Deep Virtual Evidence CPTs — neural networks (such as multi-layered perceptrons) and deep neural networks can express evidence directly to any random variable via a virtual evidence mechanism. These are called DeepVECPTs.
- Deep CPTs — it is also possible to use a deep neural network to express a discrete CPT. I.e. a deep model can be used for the implementation of CPT, e.g., $p(a|b, c, d)$, where b, c, d are the inputs to the deep NN and the output of the deep NN gives a probability distribution over a .
- Deep neural network training code — if you have a modern C++ compiler, GMTK will also compile code that can be used to train a variety of deep models. While this code runs on a CPU (rather than on a GPU) and thus will be relatively slow, it is still possible to get a system working and training even a many-layered (≥ 3) MLP on a CPU, including the GMTK TIMIT tutorial which is available. To do this in earnest, however, it is recommended that you use a GPU for any deep model training, and GPU-based deep neural network training code is not part of GMTK.
- Switching weights (penalties, scales, and shifts). Any probability score p of a random variable X can be modified as $\text{penalty} * p^{\text{scale}} + \text{shift}$ and moreover the penalty, scale, and shift can change depending on the values of X 's parents. This generalizes to the DGM case concepts common in speech recognition such as language model scale factors, acoustic model scale factors, pronunciation model scale factors, and word insertion penalties (which are all easily specifiable in GMTK).
- Probabilistic inference using dynamic programming with optimized low-level data structures — either single pass or multi-pass decoding (say via a lattice) is possible.
- Verbose debugging/trace inference output at various levels, controllable on the command line. This can be very useful when you are debugging your model, or you wish to take a look at what the inference code is doing. A warning, here, is that one can quickly generate terabytes of data with these options.
- Algorithms for parameter tying based on various clustering heuristics. A program `gmtkTie` is available that allows for data-driven determination of what parameters should be shared, and what states should be tied or merged.
- Parallel computing is key to get things working fast on modern microprocessors. GMTK supports parallel EM training, so that it is possible to get almost linear speedup with an arbitrary number of processors. This has made it possible to train very large systems relatively quickly.

0.2.2 Advanced low-level GMTK Features

A number of GMTK features might be of interest to the more advanced users and to users who are more interested in the underlying inference algorithms.

- Boundary/separator determination to find good DGM chunks that are to be repeated/unrolled. This sometimes produces an inference algorithm that runs much faster than the naïve chunk based on a slice and either its left or right interface.
- Max-flow (and submodular) based boundary and separator determination.

- Triangulation of DGM chunks rather than the entire unrolled graph.
- Separate GMTK triangulation engine `gmtkTriangulate`. Many different triangulation heuristics are supported.
- An anytime triangulation search - here, you can let GMTK spend as much time as you are willing to give it to try to find a good triangulation. This includes simulated annealing-based triangulation and exponential-time exhaustive search triangulation.
- Virtual Evidence Separators – separators can themselves come from virtual evidence functions. This feature has sometimes sped up inference by factors of 10,000.
- Big cliques can be fast cliques (when determinism abounds). Non-elimination based triangulation heuristics.
- Separator (rather than clique) driven inference to quickly and dynamically remove zeros. Yields exact and approximate inference algorithms (via dynamic pruning)
- Topological, ascending cardinality, fail first, and many other variable ordering heuristics.
- Almost optimal packed clique table representations, improving cache use, speed, required memory resources.
- Shared DGM clique-table value sharing, further reducing memory demands.
- No C++ STL used for inner loops. Custom internal data structures used for the innermost loops, PHiPAC-style [33] optimizations, and memory management.

0.2.3 What GMTK doesn't support (yet)

GMTK can do many things, but not everything. Below are a set of features that are commonly asked about that GMTK does not (yet) support.

- *k*-best lists - GMTK currently produces only the 1-best (or Viterbi) path, but it is often desirable to produce the top *k* paths. Getting this working is high on the priority list of features to add next.
- Discriminative training - currently GMTK only supports a generative loss function (log likelihood) and training via the EM algorithm. Again, this is high priority to address next.
- Undirected edges and log-linear factors of the form $e^{\lambda f(x)}$ where λ is a vector of parameters and $f(x)$ is a vector of features based on random variable vector x .
- Hidden continuous random variables - right now, GMTK only supports observed continuous random variables. On the other hand, we have had success in 2D mapping problems using quantized continuous variables represented as discrete variables via the use of approximate inference.
- Online inference - currently, the observation mechanism is such that you need to read in an entire observation file first and then start doing inference. In online inference you process the next chunk of observations as it comes in, and they can come in indefinitely. Note that GMTK *can* do Kalman filter style inference (i.e., given the current chunk of observations produce posteriors over the hidden variables), it can't at this time do online inference (due only to the lack of observation file support).

- Continuous time Bayesian networks (CTBN) - while GMTK does not support continuous time, we have found that one can simulate this quite well using observed random variables that indicate the time of the event of the current frame. That way, the random distributions can allow for a dependency on a (non-fixed) amount of time between frames or slides (or “chunks”, to use GMTK’s terminology).
- Parallel decoding. While parallel model training is supported, at this time, computing, say $\text{argmax}_{q_{1:T}} p(q_{1:T}, \bar{x}_{1:T})$ is still done in a single thread.
- Single pass re-training of observation distributions. This is a feature that HTK has used in the past to be able to quickly boot a Gaussian-based model based on one set of observations using another already-trained Gaussian-based model on a different set of observations, something that can be done in a single pass.

It is planned to have many of these features working by the version 2.0 release of GMTK.

0.3 Acknowledgments

I take full responsibility for all of the bugs in and limitations of this documentation and bugs in the code. I of course do **not** take any legal responsibility or accountability for any of the bugs in either the documentation or the code, nor do I take responsibility or accountability of any kind, form, or manner for any consequences of those bugs. Please see the license agreement before you start using GMTK.

On the other hand, many people have contributed to GMTK in major ways, and this includes: Chris Bartels, Karim Filali, Gang Ji, Simon King, Richard Rogers, and Arthur Kantor. I also want to especially thank Geoff Zweig for contributing to an older year-2001 version of GMTK — while none of this old code remains in the current version of GMTK, his inspirational Ph.D. and Masters thesis work at U.C. Berkeley in the late 1990s continue to be an influence. I would also like to single out and thank Richard Rogers who over the past few years has helped make not just valuable but also critical improvements and enhancements to GMTK, making it a significantly more useful toolkit.

I’d also like at this point to thank some of the most frequent users of GMTK and who have painstakingly and meticulously reported bugs that (I hope) have all been fixed at this point. Those individuals include Zafer Aydin, Chris Bartels, Ozgur Cetin, Chiaping Chen, Karim Filali, Joe Frankel, John Halloran, Michael Hoffman, Thad Hughes, Simon King, Raghunandan Kumaran, Chandrashekhar Lavania, Max Libbrecht, Gang Ji, Xiao Li, Karin Livescu, Jon Malkin, Bill Noble, Sheila Reynolds, Richard Rogers, Ajit Singh, Amar Subramanya, Kai Wei, and Habil Zare.

I would also like to acknowledge the many users to have been able to get their research done using GMTK. GMTK over the years has been used for many applications in many domains, and has been cited by many publications at this time, both by myself [14, 39, 15, 20, 34, 38, 79, 82, 80, 143, 212, 210, 209, 213, 240, 267, 276, 292, 340, 351, 355, 405, 406, 410, 413, 449, 460, 16, 17, 247, 266, 270, 268, 272, 411, 408, 407, 409, 100, 6, 286, 354, 88, 18, 271, 274, 273, 196] and many others internationally (some but not all of the publications that have used GMTK include [77, 76, 75, 74, 119, 132, 133, 153, 151, 176, 185, 242, 278, 277, 346, 380, 374, 375, 381, 394, 395, 396, 431, 437, 455, 73, 78, 120, 121, 123, 152, 216, 228, 236, 282, 285, 202, 362, 382, 427, 390, 435, 384, 327, 383, 416, 91, 352, 387, 446, 415, 215, 293, 391, 291, 12, 316, 424, 68, 388, 423, 90, 147, 422, 419, 89, 302, 232, 451, 303, 301, 417, 85, 450, 231, 376, 118, 122, 441, 442]). Many of the applications cited above have been able to use GMTK in ways that I initially would never have imagined. I hope that this documentation will help enable further use of GMTK and allow further new research to be done.

I would also like to thank the countless users who have contacted me over the years by email with questions about GMTK. Much of the documentation below is based on emails I sent out answering such

questions. And for those of you who have been asking me about when the new version of the documentation will finally be finished, and when the source code will finally be released? Well, here it is at last. Thanks for your patience, and apologies for my delays!! ☺

Lastly, I would like to thank the National Science Foundation (NSF). Much of GMTK is due to funded provided by the NSF. This is either indirectly via its support of the initial JHU workshop in 2001. It is also most recently due to the National Science Foundation (NSF) grant CNS-0855230. Other NSF funding that has indirectly contributed to GMTK via supporting research that used GMTK includes IIS-0905341, IIS-0093430, IIS-0434720, and IIS-0326382. Other funding agencies that have helped in various ways to support projects that lead to improvements to GMTK include DARPA's ASSIST Program (contract number NBCH-C-05-0137), NIH awards R01 GM096306 and P41 GM103533, and an ONR MURI grant exi(No. N000140510388). In all cases above, the opinions expressed in this work are those of the author and do not necessarily reflect the views of the funding agency. I would also like to thank the following organizations for their generous gifts that, by helping to fund students, have indirectly benefited GMTK. This includes Microsoft, the Intel corporation, and Google.

0.4 Where to report bugs?

All bugs in the GMTK source code should be reported via GMTK's bug tracker. To report a bug, visit the following URL <https://j.ee.washington.edu/trac/gmtk>.

All bugs, typos, or sections that are unclear in this documentation should be reported to me, Jeff Bilmes, via email. See my web page <http://melodi.ee.washington.edu/~bilmes> to get my email address. Thanks very much for letting me know!

0.5 How to cite this document and GMTK in general?

If you find GMTK useful and you wish to mention it in your references either as a tutorial on graphical models, dynamic graphical models, or as reference material for GMTK itself you can cite it. Here is a citation you can use for now until it becomes a more proper form of publication.

```
@Manual{bilmesGMTKdocs2020,
  title = {Dynamic Graphical Models and the Graphical Models Toolkit (GMTK)},
  author = {Jeff A. Bilmes},
  organization = {University of Washington, Seattle},
  year = {2020},
  annote = {Preliminary Documentation}
}
```

If you are interested in citing a published source on dynamic graphical models from the GMTK perspective, you can use the citation [50].

0.6 Notation

To read this documentation, it would be beneficial to have familiarity with certain basic concepts, including those from speech recognition [347, 109, 206, 452, 453, 200], natural language processing [227], bioinformatics [134, 182], graphical models [336, 208, 258, 219], although there will be review material starting in Chapter 2.

The rest of this section will introduce notation that will be used throughout the documentation. Capital letters (e.g., X, Y, Q) refer to random variables, lower case letters (e.g., x, q) refer to values of those random variables, and the following (e.g., D_X, D_Q) refers to possible values so that $x \in D_X, q \in D_Q$.

If X is distributed according to p , it will be written $X \sim p(X)$. Probabilities are denoted $p_X(X = x)$, $p(X = x)$, or $p(x)$ which are considered equivalent. When we just write “ x ” as the value of a random variable, what we will often mean is the probabilistic event $\{X = x\}$, meaning the event that the random variable X has taken on value $x \in D_X$. For notational simplicity, $p(x)$ will at different times symbolize a continuous probability density or a discrete probability mass function. The distinction will be made as needed.

We use a matlab-like notation $1 : N$ to denote the set of integers $[N] \triangleq \{1, 2, \dots, N\}$. A set of N random variables (RVs) is denoted as $X_{1:N} \equiv X_{[N]}$. We sometimes use $U \equiv 1 : N$ to denote the universe of all possible variables. The word “universe” is used deliberately to mean that $X_{1:N} = X_U$ consists of all of the variables in some underlying physical process that is being modeled by a Bayesian network’s corresponding probability distribution. We will use $X \equiv X_{1:N} \equiv X_U$. Given any index subset $S \subseteq 1 : N$ (alternatively $S \subseteq U$), where $S = \{s_1, s_2, \dots, s_{|S|}\}$, the corresponding subset of random variables is denoted $X_S = \{X_{s_1}, X_{s_2}, \dots, X_{s_{|S|}}\}$. In general, we use upper case letters such as A, B, S, U , refer to index sets.

It will be necessary to refer to sets of integer-indexed random variables. Let $A \triangleq \{a_1, a_2, \dots, a_T\}$ be a set of T integers. Then $X_A \triangleq \{X_{a_1}, X_{a_2}, \dots, X_{a_T}\}$. If $B \subset A$ then $X_B \subset X_A$. It will also be useful to define sets of integers using Matlab-like ranges. As such, $X_{i:j}$ with $i < j$ will refer to the variables X_i, X_{i+1}, \dots, X_j . $X_{<i} \triangleq \{X_1, X_2, \dots, X_{i-1}\}$, and $X_{\neg t} \triangleq X_{1:T} \setminus X_t = \{X_1, X_2, \dots, X_{t-1}, X_{t+1}, X_{t+2}, \dots, X_T\}$ where T will be clear from the context, and \setminus is the set difference operator. When referring to sets of T random variables, it will also be useful to define $X \triangleq X_{1:T}$ and $x \triangleq x_{1:T}$. Additional notation will be defined when needed.

There might be sources of information (e.g., the origin of virtual evidence), however, that lie entirely outside this universe.

We use the power-set notation for the set of all sets. That is, given that U is a finite set of elements, then 2^U is the set of all subsets, i.e., $2^U = \{A : A \subseteq U\}$. This notation comes from function mappings. I.e., 2^U is the set of all functions that $f : U \rightarrow \{0, 1\}$, meaning a subset of U can be seen as function that maps from all members of U to binary numbers indicating set membership. This notation is often generalized to indicate functions between two domains (or sets), say \mathcal{D}_1 and \mathcal{D}_2 , so that $\mathcal{D}_1^{\mathcal{D}_2}$ is the set of all functions $f : \mathcal{D}_2 \rightarrow \mathcal{D}_1$.

Upper case letters, such as W, X, Y , and Z , refer to random variables and lower case letters (w, x, y , and z) their values. We may refer to $p_X(X = x)$ as $p_X(X = x) = p(X = x) = p(x)$ meaning the probability that the (vector) RV X is equal to the (vector) value x . Each random variable can take one of a set of values. We herein are primarily going to refer to discrete RVs for simplicity. While we will discuss observed continuous random variables, hidden continuous random variables require more notational machinery (Lebesgue integration) but are otherwise similar.

The set of values that a RV X_i may take on (sometimes referred to as X_i ’s domain) is denoted D_{X_i} . The size of the set D_{X_i} (denoted $|D_{X_i}|$) is referred to the *cardinality* of the set D_{X_i} and we therefore also impart a “cardinality” to the random variable X_i . We may therefore also refer to the set of values that a vector RV X_S may take on as $D_{X_S} = D_{X_{s_1}} \times D_{X_{s_2}} \dots D_{X_{s_{|S|}}}$, which is the Cartesian product of the individual sets $D_{X_i}, i \in S$. Therefore, we have that $x_S \in D_{X_S}$. Note also that $D_{X_U} = D_X$ is the set of all possible values of the entire universe X_U of random variables.

It is always possible to compute the marginal distribution over any set of variables. I.e.,

$$p(x_S) = \sum_{x_{U \setminus S}} p(x_U) \quad (1)$$

By this equation, the sum means that we sum over all values of all variables in the universe other than the variables X_S . We use the notation $U \setminus S$, which means all the members of the set U except for those in S . $U \setminus S$ can be read “U except for S”, “S removed from U”, or even simply “U minus S.”

The *event* that the set of all RVs has a specific value is denoted as $\{X_{1:N} = x_{1:N}\}$ or alternatively, $\{X_U = x_U\}$. Such an event is one where the entire universe of variables is seen to have a set of values, x_i , for $i \in U$. The probability of the event that a subset of random variables X_S is a particular value x_S is $p_{X_S}(X_S = x_S) = p(x_s)$ for any $S \subseteq U$.

A probability measure space is (Ω, \mathcal{F}, P) where Ω is an arbitrary set of points $\omega \in \Omega$, \mathcal{F} is a σ -field of measurable subsets of Ω (i.e., a set of subsets of Ω that are measurable), and P is a probability measure [27]. We will use symbols such as $\eta, \nu \in \mathcal{F}$ for measurable events.

A random variable is defined as a function on a sample space. I.e., a random variable maps measurable events to numeric quantities, therefore we can say that $p(X_U = x_u) = P(\{\omega : X_U(\omega) = x_u\})$. When we have defined a universe of random variables X_U , we are saying that we have partitioned the space Ω into measurable events of the form $X_U = x_u$, and from the perspective of the underlying measure P , it is only via this partition $p(X_U = x_u)$ that we have all knowledge about the underlying measure P , although there can of course exist multiple measures that lead to the same probability distribution, since the partition $X_U = x_u$ need not be the most fine grained.

A random variable X_i is called a *constant random variable* if it is the case that $p(X_i = a) = 1$ for some $a \in \mathcal{D}_{X_i}$. If it is the case that $p(X_i = x_1) > 0$ and $p(X_i = x_2) > 0$ for some $x_1 \neq x_2$, then the variable X_i is not constant and is a true “random” variable. If it is the case that $p(x_i|x_{\pi_i}) > 0$ for all $x_i \in \mathcal{D}_{X_i}$ and $x_{\pi_i} \in \mathcal{D}_{X_{\pi_i}}$ then we say that the conditional probability table $p(x_i|x_{\pi_i})$ is *dense* — we use the word dense since if the table was stored exactly, then all entries would be non-zero. If $p(x_i|x_{\pi_i}) = 0$ for some (but not all) $x_i \in \mathcal{D}_{X_i}$ and $x_{\pi_i} \in \mathcal{D}_{X_{\pi_i}}$ then we say that the conditional probability table $p(x_i|x_{\pi_i})$ is *sparse* — we use the word sparse since the CPT table could be represented as a sparse matrix. If it is the case that

$$p(x_i|x_{\pi_i}) = \begin{cases} 1 & \text{if } x_i = f(x_{\pi_i}) \\ 0 & \text{else} \end{cases} = \delta(x_i, f(x_{\pi_i})) \quad (2)$$

for a particular deterministic function $f : \mathcal{D}_{X_{\pi_i}} \rightarrow \mathcal{D}_{x_i}$, then we say that the conditional probability table is *deterministic*. This means that for a given set of parent values x_{π_i} there is only **one** value of the child variable that has any probability, and that value of the child variable, thus, has **all** the probability (it is the only thing that can happen).

By convention, we will in general use variables X as “input” variables, variables Y or Q as “output”, variables and other variables such as H , will be used for intermediary variables. Also, \bar{x} generally means that x is observed (a constant). Thus, we might use a probability distribution as

$$p(y, q, h, \bar{x}) = p(y, q, h|\bar{x})p(\bar{x}) \quad (3)$$

where \bar{x} are input observed variables, the goal is to predict both y and q and h are hidden variables. For example, the prediction might be based on

$$p(y, q|\bar{x}) = \sum_h p(y, q, h|\bar{x}) \quad (4)$$

Of course, any of these variables might be vectors.

Other variables will be defined in the context where they are used.

0.7 Graph Notation and Terminology

A graph G consists of a pair $G = (V, E)$ where V is a finite set of vertices and $E \subseteq V \times V$ is a set of edges. A given edge is denoted $(u, v) = e \in E$ with $u, v \in V$. We also use $G = (V, E)$ as functions, in that $V(G)$ consists of the vertices of a graph G and $E(G)$ is the set of edges. This is done to allow us to refer to the vertex set of two graphs. For example, given two graphs G_1 and G_2 we might say that $V(G_1) = V(G_2)$ (meaning the two graphs are over the same vertex set) but $E(G_1) \subset E(G_2)$ (meaning that G_1 is a spanning subgraph of G_2).

In graph theory, there are often multiple names to refer to vertices and edges of a graph. For example, a vertex of a graph might be referred to either as a vertex, a node, a point, or even a state. An edge of a graph might be referred to either as an edge, an arc, a transition, or a link. In order to avoid confusion, when referring to graphical models (cf. § 2.2), we will say that $G = (V, E)$ is a graph where V consists of either vertices and E is a set of edges. When referring to the graph corresponding to an a finite state automata (FSA) (cf. § 8.4.4) or, as we will see, to the state transition graphs for HMMs (cf. § 8.4.4) we will refer to the vertices as “nodes” and the edges as “arcs”. There is a third type of graph that we will commonly use in this document, and that is a trellis or lattice graph and these are used for things like HMM lattices (cf. § 8.4.5). In the trellis case, we will refer to the vertices as “points” and the edges as “links.”

GMTK also uses decision trees, and since they are trees, they are also graphs. Hence, for decision trees we will use “nodes” for vertices (e.g., leaf nodes) and “arcs” for edges.

Suppose we are given a graph $G = (V, E)$. Often, we are able to identify each edge $e \in E$ with two vertices, meaning $e = (u, v)$ with $u, v \in V$. Such graphs, where there is only one possible edge between two nodes, are called simple graphs. If it is the case, however, that there are two distinct edges $e, e' \in E$ over the same node pair, then the edges are called parallel. Such edges can be identified using a tuple consisting of a node pair and a unique index (e.g., $e = (u, v, 1)$ and $e' = (u, v, 2)$). . Thus, graphs that do not have parallel edges are simple.

0.8 For Further Reading

While the documentation below contains a self-contained tutorial on both static and dynamic graphical models (and includes models such as hidden Markov models, conditional random fields, and dynamic Bayesian networks), the reader is encouraged to read other sources as well. This section provides a list of references to accompany the GMTK documentation. The list is far from complete, and is intended only to provide a useful starting point and also to give a bit of historical perspective. The references below should offer a good starting point towards an in-depth study of dynamic Graphical models.

0.8.1 Background in Speech Recognition and Language Processing

There is a wealth of material in speech and language processing. Probably the best place to start are the various text books that are available. The text [347] provides a good and rigorous overview of the general problems in speech recognition, as does [109]. [350] is a bit older, but is also quite thorough and is from a signal processing and analysis perspective. [206] is a great text on hidden Markov models and language modeling techniques. [60, 310, 250] gives an approach to speech recognition using neural networks and [311] includes material both on speech and audio processing. [201] provides an account of hidden Markov models in speech recognition, and [200] is an up-to-date account of various techniques in speech recognition, speech synthesis, and spoken language processing. [453] is a text on HTK (the hidden Markov models toolkit), but there is plenty of material on general speech recognition. Also, [452] provides an overview of large vocabulary techniques. [226] is a very thorough text on natural language processing as is [294], both of which are great sources of information on the tagging problem. Also, [65] is a classic text in auditory

scene and sound analysis. The modeling of pronunciations is described in [86, 363, 136, 150, 226], and that of language in [369, 324, 342, 26, 87, 206, 370].

0.8.2 Background in Graphical Models

While readers will certainly not be unfamiliar with graphical models, we include here the standard references for completeness. Some standard texts on graphical models include [336, 438, 321, 373, 3, 207, 258, 71, 98, 338, 135, 208, 320]. A number of review articles are also available [189, 191]. [217] is a good collection of various papers on graphical modeling, and [219] is one of the few books to have become a classic before it has even been published. Bayesian multi-nets and switching dependencies are described in [188, 168, 41]. Examples of multinets used in practice include mixtures of tree-dependent distributions [300] and class-conditional naïve Bayes classifiers [155, 154]. Also, see the book by Bishop [58] and Murphy [314].

0.8.3 Hidden Markov Models: HMMs

Most of the texts listed in Section 0.8.1 have chapters on HMMs. Perhaps the earliest discussions of hidden Markov models are given in [393, 183, 59, 171, 21, 22]. There are a number of accounts of HMMs and their use in speech recognition [265, 10, 264, 225, 349, 348, 345, 99, 42, 30]. Inference in hidden Markov models and Bayesian networks was shown equivalent in [398]. [288] is another text on hidden Markov modeling.

Note that a complete description of HMMs (in fact many different perspectives on HMMs) is given in §8.4.

0.8.4 Discriminative Parameter Training in HMMs

Discriminative training methods have been developed which adjust the parameters of each model to increase not the individual likelihood but rather approximate the posterior probability or Bayes decision rule. Methods such as maximum mutual information (MMI) [10, 66], minimum discrimination information (MDI) [139, 141], minimum classification error (MCE) [223, 222, 234], and more generally risk minimization [130, 428] essentially attempt to optimize $p(M|X)$ (conditional likelihood) by adjusting whatever model parameters are available, be they the likelihoods $p(X|M)$, posteriors, or something else. Discriminative structure learning is described in [45, 39].

0.8.5 Explicit Control of ASR system via Bayesian Networks

One of the first papers on graphs whose purpose was to represent the explicit control constructs in an ASR system (specifically parameter tying) was [462], and was closely followed by [468, 467, 465, 469, 466] and more recently [39, 464, 38, 463]. Other papers have presented algorithms that help to optimize performance in such networks [114, 199]. A brief general outline is given in [113].

0.8.6 Latent Modeling

Over the years, many research papers have used represent additional latent information by use of an HMM with an expanded state space. An early work specifically using dynamic Bayesian network is [462]. In [39], a number of models are proposed and tested for this purpose, including a hidden noise condition model, hidden articulatory modeling, and speaker clustering. General articulatory modeling was used in speech in [111, 237, 229, 238, 239] and hidden articulatory modeling is tested in [357, 358, 359, 399, 281]. Factorial HMMs [170] were attempted for speech in [283] and for multi-channel source separation in [353]. Latent discretized pitch and energy are modeling in [402, 400]. Graphical models for language modeling

is described in [39, 41]. Factored language models are described in [241, 36]. Large vocabulary speech systems using graphical models are described in [72].

0.8.7 Observation Modeling

A number of projects have extended the statistical dependencies of HMMs and have been called auto-regressive HMMs or correlation HMMs. Perhaps the first was [436, 66]. These are auto-regressions over features themselves. Both [66] and [235] tested implementations of such models but improvements were found only when delta features were excluded. Similar results were found by [125] but for segment models [330, 166]. In [333], the random variables were discrete only. In [329], a parallel algorithm was presented that can efficiently perform inference with such models. Other instances of such state-dependent auto-regressive modeling on features include [397, 315, 214, 401] (they appear to have been independently re-discovered a number of different times). Improvements were reported with delta features in [444] using discriminative output distributions [443]. In [262, 263], successful results were obtained using delta features but where the conditional mean, rather than being linear, was non-linear and was implemented using a neural network. Also, in [418], benefits were obtained using mixtures of discrete distributions.

In Buried Markov models (BMMs), improvements were found with delta features, but here the dependencies were sparse, chosen individually between feature vector elements, and according to an data-driven hidden-variable dependent information-theoretic and discriminative criteria [45, 44, 43]. BMMs were applied across multiple separate features in [39].

Note above that auto-regressive HMMs refers to an auto-regression over feature vector elements. These AR-HMMs are not to be confused with the auto-regressive, linear predictive, or hidden filter HMMs in [344, 345, 224, 347]. These latter models are HMMs that have been inspired from the use of linear-predictive coefficients directly on the sampled speech signal.

Segment models are described in [331, 166, 330]. The use of dynamic or delta features [137, 161, 162, 163] has become standard in state-of-the-art speech recognition systems. Acceleration, or delta-delta, features may similarly be defined and are sometimes found to be additionally beneficial [439, 260]. Delta features analyzed with graphical models is described in [41].

Neural networks can also be used as models of observations and have recently been hugely successful for speech recognition when the neural networks have many layers (i.e., deep models). Neural network outputs can be fed directly to Gaussians (either by adding a log transformation or by using the final NN layer as a linear layer, something called the “tandem” approach [193]) or alternatively their probabilities can be directly applied (what has been known as Hybrid HMM/ANN models [60, 310]). This can be seen as a form of virtual evidence. To see how to use Neural network outputs to be fed into GMTK via virtual evidence (thereby implementing and generalizing Hybrid HMM/ANN models), see Section 5.7.2.

0.8.8 Miscellaneous Topics

Structure learning problem in DBNs for speech recognition is described in [43, 44, 28, 45, 39, 116, 115, 31]. Mixed-memory Markov models for speech are described in [328]. A number of papers have used Bayesian networks to represent speech as a series of separate bands [304, 101, 102, 461, 103, 178]. Multi-band and multi-modal (audio/visual) ASR fusion with DBNs is described in [167, 323, 322, 23]. Even HTK has exponentially weighted product multi-stream capabilities [453].

Linear discriminant analysis (and its heteroscedastic generalizations) are described in [255]. A number of GM-inspired models for de-noising are described in [128, 5]. An auto-regressive model for de-noising is described in [284]. Speaker recognition with GMs was done in [377].

0.8.9 General Structure Learning

Often, the true (or the best) structure for a given task is unknown. This can mean that either some of the edges or nodes (which can be hidden) or both can be unknown. This has motivated research on learning the structure of the network model from the data, with the general goal to produce a structure that accurately reflects the important statistical properties in the data set [93, 189, 69, 251, 154, 156, 92]. These can take a Bayesian [189, 190] or frequentist point of view [69, 251, 189]. Structure learning is akin to both statistical model selection [275, 70] and data mining [106]. Several good reviews of structure learning are presented in [69, 251, 189]. [156] provides a recent and good NIPS tutorial, along with a list of references. Structure learning from a discriminative perspective, thereby producing what is called *discriminative generative models*, was proposed in [28].

0.8.10 Quick List: Dynamic Bayesian Network References

All speech and language based BN systems are DBNs. Here are a few references for some relevant DBN work done to date. Dynamic Bayesian networks [107] are a generalization of BNs [336, 208] to explicitly include time. There are a number of papers on these networks, and how to do exact and approximate inference [244, 246, 169, 157, 63, 62, 448, 372, 104, 312, 313] and their applications [55, 149]. The 2nd edition of [373] will have an excellent chapter on DBNs. Probabilistic relational models [158] have also been extended into the dynamic domain [378]. For speech recognition, dynamic Bayesian networks (DBNs) [468, 462, 312, 463, 45] have been used quite successfully.

Part II

Overview of Graphical Models

Chapter 1

Introduction to Graphical Models: Its a Family Affair

1.1 Introduction

Our primary concern is to model, accurately represent, and understand signals that can occur anywhere in the universe. We may wish to understand one class of signals alone, or we may also wish to understand how one set of signals relates to another set of signals. The signals can be diverse in form, some might be discrete, some continuous, some high-dimensional, and others low-dimensional. The signals could be purely natural in their origin (such as weather patterns, or radiation from outer space), they could arise due to human social activity (e.g., macro level economic indicators, the stock market, flu-spread patterns) or individual activity (speech recognition, visual object recognition), or could arise due to biological function (e.g., systolic pulses or other vital signs of a living organism, or at a much lower level such as protein-protein interaction or genomic variation).

We are particularly interested in how certain signals relate to each other. For example, one might be interested in the process by which a mental state gets transformed into an acoustic speech waveform (the reverse of which would constitute a speech recognition system). Or we may wish to model the process by which a statement in one language gets transformed by an interpreter into another language. We may wish to model the process of mitosis, where a cell separates into two offspring cells. Or we may wish to model the way in which social relationships evolve over time (if person A and person B are friends, is person A likely to become a friend of person C?).

In theory, it may be that each of these processes are deterministic and are governed by the laws of physics. Assuming all information is known about the processes, it seems that we conceivably could explain precisely and exactly how a process proceeds over time. This would require a sufficiently complicated model of the process and a sufficient amount of information about the process in order to choose our model and a sufficient amount of computational resources to make predictions based on this model. All of nature perhaps could be described deterministically using a suitably complex set of non-linear differential equations at the sub-atomic level.

In practice, however, the small details that entail a complicated process are never available. In general it is not technologically practical to observe all of the intricate details of a process. It is hard to even imagine, for example, how many sensors we would need to have in order to obtain a perfect picture of the current state of the world's weather. And this is notwithstanding the unavoidable Heisenberg uncertainty principle, whereby even the process of observing a system can disturb it and make it different than what was observed. It is therefore never possible to gain enough information about a complicated process so as to produce a model that mimics it perfectly.

Even if it were possible, such a model would be extremely difficult to understand, maintain, and deal

with computationally. Therefore, rather than striving to achieve the perfect representation of nature, we instead are willing to accept a much simpler model, one that only approximates reality. The simple model should be sufficiently accurate so that it can make reasonable predictions for a given application, but again it must not be too complicated so that it is intractable to deal with.

Any given model might be parameterized, and the parameters govern the flavor of that particular model. As an example, an extremely simple model might relate the variable x with the variable y in a linear fashion.

$$y = mx + b \quad (1.1)$$

The model states that y is linearly related to x via the parameters m and b . We note that this model is “deterministic” in that it does not allow for any variability in the relationship between x and y . That is, this model insists that x and y have a linear relationship — if it was the case that x and y had only an approximate (rather than true) linear relationship, there would be no way, using this model alone, to account for this. This is an *all or nothing* model.

We note that such a model can also be seen as a probability distribution as follows:

$$p(x, y) \propto \begin{cases} g(x) & \text{if } y = mx + b \\ 0 & \text{else} \end{cases} \quad (1.2)$$

where $g(x)$ is some non-negative function of x alone. This model says that it is impossible for y and x to be related other than deterministically.¹ The model also states that anything other than this specific linear relationship, under parameters m and b is impossible.

In general, we are much happier with an approximation given by a model if it not only imparts accurate predictive power to a user but also if it can express uncertainty about that prediction. That is, the simple model above expresses no uncertainty, but another model might say that there is *noise* associated with the above relationship, as in:

$$y = mx + b + \epsilon \quad (1.3)$$

where $\epsilon \sim \mathcal{N}(0, \sigma^2)$ is a Gaussian-distributed random variable with mean 0 and variance σ^2 . The value σ^2 determines how uncertain the model is — if $\sigma^2 = 0$ then there is no uncertainty, and as σ^2 gets larger, uncertainty increases. We note that the term *noise* has historically suggested that the signal is indeed linearly related but that, due to various measurement errors, this relationship when expressed in real data is not purely linear. That is, noise is seen only as the “senseless” part of the signal, and is a nuisance that ideally would not be encountered since it does not exist in nature.

Another way of thinking about these errors is that they indeed express deviations from linearity but they might in fact be due to those complicated but real aspects of the physical process that are not well represented by a linear function. I.e., they might exist, and if we had a rich (but complicated) enough model, there would be no random component. For the reasons mentioned above, however, it might not be necessary to understand their relationship exactly.

Generally speaking, the most popular measure of uncertainty used in recent years is one based on probability. That is, we might have a collection of distinct variables x_1, \dots, x_N that describe a process, and we model the relationship between these variables as saying that there is a probability distribution over those variables

$$p(x_1, \dots, x_N). \quad (1.4)$$

This distribution must satisfy the axioms of probability $p(x_1, \dots, x_N) \geq 0$ and $\sum_{x_1, \dots, x_N} p(x_1, \dots, x_N) = 1$. Again, one could strive to acquire a suitably complex set of equations that express this relationship, in

¹We use the \propto symbol to indicate that the probability distribution is proportional to the value on the r.h.s. Moreover, any positive number proportional to unity would not integrate to unity over the domain of x and y , and a more precise statement would use a generalized function. Our intent here is to convey that anything other than determinism is impossible.

which case for any $N - 1$ values of the variables, the N^{th} variable value would be functionally determined — in other words, for any $i \in \{1, \dots, N\}$ there would exist a function $f_i : D_{X_{\{1, \dots, N\} \setminus \{i\}}} \rightarrow \mathbb{R}$ such that

$$p(x_{\{1, \dots, N\} \setminus \{i\}}, \hat{x}_i) \propto \begin{cases} 1 & \text{if } \hat{x}_i = f(x_{\{1, \dots, N\} \setminus \{i\}}) \\ 0 & \text{else} \end{cases} \quad (1.5)$$

Such a collection of functions would be overly complex and unnecessarily detailed. Instead, we could consider finding a probability distribution such that for every set of values of x_1, \dots, x_N we can obtain a probability

$$p(x_1, \dots, x_N) \quad (1.6)$$

that states how probable this combination of values is. In fact, the resulting distributions we use might never say that any combination of variables is impossible,

$$p(x_1, \dots, x_N) > 0 \quad \forall x_1, \dots, x_N \quad (1.7)$$

meaning that anything is possible, however improbable it might be. Such a “positivity” assumption, as we will see (even below in this chapter), will have enormous theoretical and practical (and computational) benefits — that is, zeros in a distribution lead to a variety of difficulties.

On the other hand, this relaxation on the model, from a deterministic one to one where anything is possible and has a non-zero probability score, can significantly increase the size (or volume) of possible models. It might now be difficult to identify the best candidate model to represent some physical process. This is because the distribution $p(x) = p(x_1, \dots, x_N)$ can be formed in many different ways.

Often, we assume there exists a true distribution from nature and that we have independent samples drawn according to distribution. Alternatively, we assume there exists a true deterministic system in nature, and we know this system is particularly complex, and we observe instances of this system at work. In this latter case, we are willing to live with an approximation of this system made by some probability distribution. Yet a third possibility is some combination of the two. In all cases, we obtain knowledge about this unknown system via a combination of data samples, and any other knowledge scientists might impart on the process. That is, we are given a sampled set

$$\mathbf{D} = \left\{ x^{(j)} \right\}_{j=1}^{|D|} \quad (1.8)$$

where for each j , $x^{(j)} = (x_1^{(j)}, \dots, x_N^{(j)})$ is a vector, and where

$$x^{(j)} \sim p_{\text{true}}(x_1, \dots, x_N). \quad (1.9)$$

This sampled data set is then used, along with scientific knowledge about the domain from which the data is drawn, to produce a resulting approximating probability distribution $p(x)$ of the true physical process. That is, our goal is to take \mathbf{D} and select one distribution from the set of all possible distributions over x .² We want our resulting distribution $p(x)$ to be “close enough” to $p_{\text{true}}(x)$ that it might serve our purposes, where “our purposes” might span the range from a proving a theorem that shows that with enough data $p(x)$ gets arbitrarily close to $p_{\text{true}}(x)$ at a particular rate, or might be purely practical and would allow us to efficiently compute certain quantities of interest with the resulting $p(x)$.

In general, when we use \mathbf{D} to learn $p(x)$, it will not be possible to allow $p(x)$ to range over all the possible distributions over x . Instead, we force the resulting $p(x)$ to be within some restricted *family* of models. Lets say that \mathcal{U} is the set of all distributions $p(x)$ and $\mathcal{F} \subset \mathcal{U}$ is a subset of distributions. There are many reasons to a-prior restrict the learning to \mathcal{F} , and this includes:

²A Bayesian interpretation still fits within this framework, as some of the variables might themselves be parameters that govern the distribution over other variables, and at some point up in a hierarchical Bayesian model, the meta parameters are fixed thereby determining a joint distribution over both signal and parameter random variables.

- Learning over \mathcal{U} might be computationally complex. I.e. if the size of \mathcal{U} might be very large and diverse, and more importantly, there might not be any structure associated with the models within \mathcal{U} (i.e., it might lack convexity or submodularity).
- The size of \mathbf{D} might not be large enough to obtain a reasonable estimate. There might be too much flexibility in \mathcal{U} so that \mathbf{D} is essentially memorized, and any resulting model would not be smooth enough to generalize to unseen data points. Given the data set size, we prefer to select from a simpler set of distributions in order to “regularize” the process. This is an Occam’s razor issue.
- Searching for a model within the smaller set \mathcal{F} rather than the larger set might produce a more accurate model. That is, even though \mathcal{U} contains a model that is more accurate than the best model in \mathcal{F} , computational reasons might preclude the possibility of finding it, so that any practical algorithm has little or no chance of finding it. .

On the other hand, \mathcal{F} might be such that finding the optimal model is relatively easy (e.g., it might be a convex subset), and searching within \mathcal{F} might keep a search procedure from being distracted by the difficulties (e.g., local optima) within \mathcal{U} . Another reason why we might prefer the smaller family of models \mathcal{F} over \mathcal{U} is that there might be a useful framework in which to reason over members of \mathcal{F} – as we will see, graphical models provides such a framework.

In general, this is depicted in Figure ??.

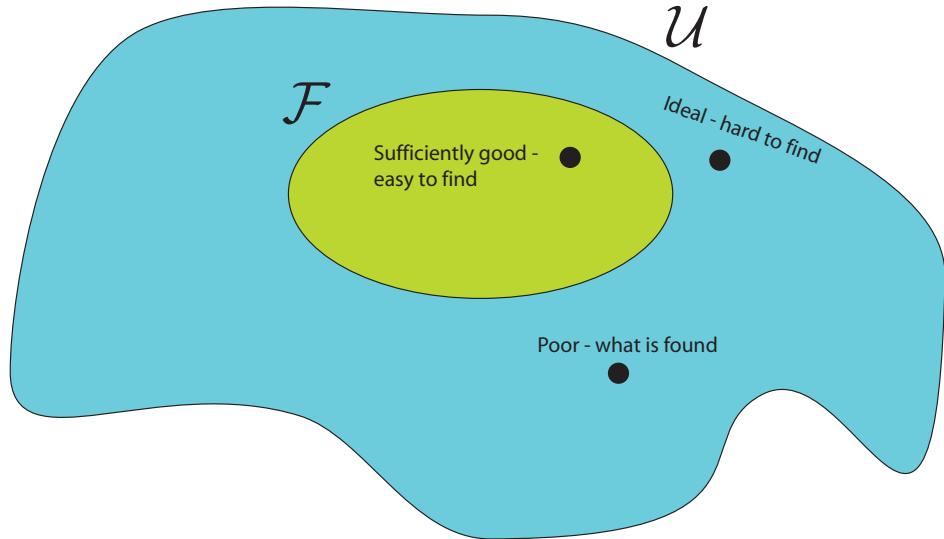


Figure 1.1: \mathcal{U} is the set of all probability distributions over x and $\mathcal{F} \subset \mathcal{U}$ is some strict subset. Even though \mathcal{F} is smaller than \mathcal{U} , it might be better for a given set of training data \mathbf{D} to select the best model from within \mathcal{F} than considering all of \mathcal{U} . The optimal point in \mathcal{U} is indicated by a black dot but it might be harder to find than the optimal point in \mathcal{F} , and moreover the best model in \mathcal{F} might be better than the best model that may be practical to find in \mathcal{U} .

The notion of a family is particularly important. In fact, we might have a chain of increasingly complex families of distributions $\mathcal{F}_1 \subset \mathcal{F}_2 \subset \dots \subset \mathcal{F}_K$ each one containing the previous. Let us suppose that the distribution $p(x)$ is governed by a set of parameters θ , and we write this as $p_\theta(x)$. We assume that θ spans the set of all possible distributions for $p(x)$. This means that θ might be both discrete and continuous, and might actually itself be of arbitrary dimension (to account for non-parametric models).

A framework for identifying the correct model, given samples of data, is an optimization problem that includes a cost associated with how well the model respects the training data as well as a cost associated with the complexity of the family. We can quite generally represent this cost function as:

$$J(\theta) = L(\mathbf{D}, \theta) + \lambda R(\theta) \quad (1.10)$$

where $L(\mathbf{D}, \theta)$ is a risk function associated with how well a given model under parameter vector θ respects the training data, and $R(\theta)$ is a regularizer associated with the parameters that identifies the family associated with p_θ , and $R(\theta)$ expresses the complexity of the family corresponding to the parameter θ , and λ is an accuracy-regularization tradeoff parameter. Our goal is to identify a θ that finds the best tradeoff between these two potentially competing criterion.

In fact, there are likely to be critical values of λ , or $\lambda_0, \lambda_1, \dots, \lambda_K$ that separate the boundaries between these sub-families, under the assumption that each member of a given sub-family has the same complexity. This means that if we were to find

$$\theta^* \in \operatorname{argmin}_{\theta} J(\theta) \quad (1.11)$$

then we would have $p_{\theta^*} \in \mathcal{F}_k$ whenever $\lambda_{k-1} \leq \lambda < \lambda_k$. Optimizing the cost function $J(\theta)$ can be seen as first identifying the family $\mathcal{F}(\lambda)$ associated with the value λ and then optimizing the risk function staying within that family.

We might in fact know beforehand that we wish only to work within a particular (and suitably simple) model family \mathcal{F} so that our optimization problem becomes:

$$\theta^* \in \operatorname{argmin}_{\{\theta: p_\theta \in \mathcal{F}\}} J(\theta) \quad (1.12)$$

While such a constrained optimization might seem like it would be much more difficult than the unconstrained version, in certain surprising cases it is extremely simple to solve, such as learning 1-trees or learning Gaussians. See chapter .

This text is about graphical models, and the reason we are spending so much time discussing families is that, as we will see, a graphical model is a graphical depiction of a family of probability distributions, where certain formal graph-theoretical properties can be usefully exploited to reason about and compute with all members of the family simultaneously. This is quite a powerful property and we will expand on this thought in future chapters, but before doing so, we wish to consider what operations we might wish to perform on and with distributions, problems that graphical models can help us with.

If we are dealing with a distribution over a single signal x , then we might wish to ask the following questions:

- Given a candidate signal $\bar{x} = \bar{x}_U$, how likely is that \bar{x} is true? This is simply computing $p(\bar{x})$.
- Given a candidate signal that has missing information, how likely is it? I.e., we might partition $U = E \cup H$ where $E \cap H = \emptyset$. We might have only partial information about an instance of a candidate signal \bar{x}_U in the form of \bar{x}_E . This is computing $p(\bar{x}_E) = \sum_{x_H} p(x_H, \bar{x}_E)$.

It might be that we are interested in modeling the relationship between two signals x_1 and x_2 (both of which might be vectors). For example, x_1 could correspond to a set of inputs, and x_2 could correspond to a classification or summary of x_1 , or instead we could be modeling a complex function f in $x_2 = f(x_1)$, and wish to do this via a joint distribution $p(x_1, x_2)$. In such case, the kinds of questions we might wish to answer are:

- Given a pair \bar{x}_1, \bar{x}_2 how likely is this pair? This means computing $p(\bar{x}_1, \bar{x}_2)$.

- Given a particular \bar{x}_1 , compute the most likely x_2 , i.e., $x_2^* \in \operatorname{argmax}_{x_2} p(\bar{x}_1, x_2)$ or equivalently $x_2^* \in \operatorname{argmax}_{x_2} p(x_2 | \bar{x}_1)$. If x_2 is discrete, we might wish to compute a number of different x_2 's that are highly probable, such as an N -best list. If x_2 is continuous, we might wish to compute a volume in D_{x_2} that contains a certain top percentage of the probability mass.

The optimization problem mentioned above is an instance of machine learning, where from a data set D , we wish to find a set of parameters θ that best explain D . I.e., we might ask

- Given D find $\theta^* \in \operatorname{argmax}_{\theta} J(\theta)$ for a given cost function J . This simple operation encompasses most of the “learning” aspect of machine learning since many learning problems can be formulated as an optimization problem over some objective function.
- Once we have θ^* , how can we interpret its values? I.e., θ^* can be seen as a short surrogate of D . Examining the learnt parameters might tell us something more informative than looking at D alone. This is a data-mining and pattern discovery problem.

Given the learnt θ^* , we might use either $p_{\theta^*}(x)$ or $p_{\theta^*}(x_1, x_2)$ as if it was the true distribution.

- Finding $x_2^* = \operatorname{argmax}_{x_2} p_{\theta^*}(x_2 | \bar{x}_1)$ can be seen as an instance of unsupervised learning or data clustering, where the x_2^* is the cluster of \bar{x}_1 .

The risk function in Equation (1.10) also determines the type of questions we might wish to ask. For example, we might set L based on

- If $L(D, \theta) = \sum_{j=1}^{|D|} \log p_{\theta}(x^{(j)})$, then the learning process is generative, and the goal penalized maximum likelihood learning.
- If $L(D, \theta) = \sum_{j=1}^{|D|} \log p_{\theta}(x_2^{(j)} | x_1^{(j)})$, then the learning process is “discriminative” and the goal penalized conditional maximum likelihood learning. This would be appropriate for a classification or regression setting where the application only wishes a conditional distribution.

- For

$$L(D, \theta) = \sum_{i=1}^{|D|} \ell \left(\Delta(x_2^{(j)}, x'_2) - (\log p_{\theta}(x_2^{(j)}, x_1^{(j)}) - \log p_{\theta}(x'_2, x_1^{(j)})) \right) \quad (1.13)$$

with $x'_2 \in \operatorname{argmax}_{x_2} p_{\theta}(x_2, x_1^{(j)})$ and where $\Delta(x_2^{(j)}, x'_2)$ is a normalizing labeling loss (such as hinge loss), and $\ell()$ is a local point-wise margin-based loss, then this becomes max-margin learning, a form of discriminative training.

The above learning schemes can be optimized in a variety of ways. For example, computing the gradient can itself require a more primitive operation on the distribution.

Other times, we may wish to take our $p(x)$ and simply compute expected values of other quantities, as in

$$g(x_E) = \sum_{x_H} g(x_H, x_E) p(x_H, x_E) \quad (1.14)$$

where $g(\cdot)$ might be an arbitrary function.

In any of the above problems we would wish to solve, there are certain basic operations on $p(x)$ that need to be performed, namely computing probabilities and summing over many of these computed probabilities. While it might seem like computing such probabilities would be easy, in fact it can be exponentially difficult

(in the number of variables) to perform such a computation naïvely. For example, simply computing the marginal

$$p(\bar{x}_E) = \sum_{x_H} p(x_E, X_H) \quad (1.15)$$

would require an $O(r^N)$ computation since for each value of \bar{x}_E (and there are $r^{|E|}$ of them), we need to sum over $r^{|H|}$ values.

Sometimes it might be that we know that there is a much more efficient way to compute the above quantities, but it is computationally difficult to find it. At other times, we might know that computing the above quantity exactly is intractable, and it is feasible only to resort to an approximation.

Graphical models provide a mechanism that allows us to reason about when any of the above problems might be feasible, and when it is not. Graphical models provide a mathematically formal visual mechanism to reason about important properties of families of probability distributions. Graph theory

Graphical models [258, 336, 207, 219] are a set of formalisms each of which describes families of probability distributions. There are many different types of graphical models each having its own *semantics* that govern how the graph specifies a set of factorization constraints on multi-variate probability distributions. Of course factorization and conditional independence go hand-in-hand, so factorization constraints typically (but not always) involve conditional independence properties. We will be discussing factorization and conditional independence below.

There are five properties that graphical models convey, and this includes:

1. Structure: A method to explore the structure of “natural” phenomena (causal vs. correlated relations, properties of natural signals and scenes)
2. Algorithms: A set of algorithms that provide “efficient” probabilistic inference and statistical decision making
3. Language: A mathematically formal, abstract, visual language with which to efficiently discuss families of probabilistic models and their properties.
4. Approximation: Methods to explore systems of approximation and their implications. E.g., what are the consequences of a (perhaps known to be) wrong assumption?
5. Data-base: Provide a probabilistic “data-base” and corresponding “search algorithms” for making queries about properties in such model families.

Bayesian networks are only one type of graphical model where the graphs are directed and acyclic (DAG). In a Bayesian network (BN), the probability distribution over a set of variable $X_{1:N}$ factorizes with respect to a directed acyclic graph (DAG) as $P(x_{1:N}) = \prod_i P(x_i|x_{\pi_i})$ where $\pi_i \subset U$ are the set X_i ’s immediate parents of variable x_i according to the BN’s DAG. There are many additional (and provably equivalent) characterizations of BNs as we will see. Because of the strong relationship between factorization and conditional independence, the above factorization implies that a Bayesian network expresses a large number of conditional independence statements to the extent that it has missing edges in the graph. Moreover, it is the common factorization properties of the family of probability distributions that makes for efficient inference of the probabilistic questions we speak about above.

What are the advantages of factored representation? In general, a scalar distribution $p(x_1)$ requires a 1-D table. The conditional distribution $p(x_6|x_2, x_5)$ requires a 3-D table, and distribution $p(x_{1:6})$ requires a 6-D table. So the table size is exponential in number of variables N . With a factored representation the, the table size is exponential only in the largest factor, rather than N , and the number of tables linear in number of nodes. This can have enormous space savings, not to mention computational savings as we’ll see.

A graphical model specifies a family \mathcal{F} of models rather than any one model in particular. An algorithm for a given graphical model will apply to any particular instance of a model that lives within the graph's corresponding family. The graphs, along with a set of rules, convey the formal mathematical properties that must hold for all members of the families, and we can reason about all members of the family simultaneously rather than reasoning about only one particular distribution. This is extremely powerful, as it allows us to develop general algorithms for a large class of models. It has real practical benefits as computer code can be written that apply to many quite different models simultaneously. In other words, a given model family is encoded by a graph

In the next section, we define and give a number of properties of factorization, independence, and conditional independence. Having a complete understanding of this is crucial to understanding the families that graphical models represent.

The subsequent section will then briefly list a number of different examples of graphical models by giving the corresponding graph and giving a brief description of what that model represents. An observation that is immediate from these examples is that many commonly known statistical methods are instances of graphical models.

1.2 Conditional Independence

In this section, we discuss conditional independence and its relationship to factorization. This is important since graphical models essentially in one way or another encode conditional independence or factorization properties of a family of distributions.

Definition 1 (factorization). *Given a function $f(\cdot)$ defined over N variables $x_{1,\dots,N}$, we say that $f()$ factorizes over x_A and x_B if $\forall x$ there exists functions $g(x_A)$ and $h(x_B)$ such that*

$$f(x_A, x_B) = g(x_A)h(x_B). \quad (1.16)$$

We note that A and B may intersect.

Independence and conditional independence is just a form of factorization.

Definition 2 (independence). *For any probability distribution p over a set of random variables X_V , given any collection of two sets $A, B \subset V$, we say that X_A is independent of X_B if $p(x_A, x_B) = p(x_A)p(x_B)$ for all possible values x_A, x_B . This is written using the binary conditional independence relation $X_A \perp\!\!\!\perp X_B$.*

Conditional independence is just a slight generalization.

Definition 3 (conditional independence). *For any probability distribution p over a set of random variables X_V , given any collection of three sets $A, B, C \subset V$, we say that X_A is independent of X_B given X_C if $p(x_A, x_B | x_C) = p(x_A | x_C)p(x_B | x_C)$ for all possible values x_A, x_B, x_C . This is written using the ternary conditional independence relation $X_A \perp\!\!\!\perp X_B | X_C$. Whenever $X_A \perp\!\!\!\perp X_B$ (i.e., $C = \emptyset$), we call this marginal independence or just independence as above.*

Lemma 4 (conditional independence). *For any probability distribution p over a set of random variables X_V , given any collection of three sets $A, B, C \subset V$, then $X_A \perp\!\!\!\perp X_B | X_C$ holds iff there exists non-negative functions $g(x_A, x_C)$ and $h(x_B, x_C)$ such that $p(x_A, x_B, x_C)$ can be written as*

$$p(x_A, x_B, x_C) = g(x_A, x_C)h(x_B, x_C)$$

for all values of x_A, x_B, x_C such that $p(x_C) > 0$.

Proof. Necessary: Given $X_A \perp\!\!\!\perp X_B | X_C$, then $p(x_A, x_B | x_C)p(x_C) = p(x_A | x_C)p(x_C)p(x_B | x_C)$, and then take $g(x_A, x_C) = p(x_A | x_C)p(x_C)$ and $h(x_B, x_C) = p(x_B | x_C)$.

Sufficient: Given $p(x_A, x_B, x_C) = g(x_A, x_C)h(x_B, x_C)$. Then

$$p(x_A, x_C) = g(x_A, x_C) \sum_{x_B} h(x_B, x_C) = g(x_A, x_C)h'(x_C) \quad (1.17)$$

$$p(x_B, x_C) = h(x_B, x_C) \sum_{x_A} g(x_A, x_C) = h(x_B, x_C)g'(x_C) \quad (1.18)$$

$$p(x_C) = g'(x_C)h'(x_C) \quad (1.19)$$

$$p(x_A | x_C) = p(x_A, x_C) / p(x_C) = g(x_A, x_C) / g'(x_C) \quad (1.20)$$

$$p(x_B | x_C) = p(x_B, x_C) / p(x_C) = h(x_B, x_C) / h'(x_C) \quad (1.21)$$

$$p(x_A, x_B | x_C) = p(x_A, x_B, x_C) / p(x_C) = g(x_A, x_C)h(x_B, x_C) / g'(x_C)h'(x_C) \quad (1.22)$$

□

Another equivalent condition for conditional independence $X_A \perp\!\!\!\perp X_B | X_C$ is that for all values x_A, x_B, x_C , we have

$$p(x_A, x_B, x_C) = \frac{p(x_A, x_C)p(x_B, x_C)}{p(x_C)} \quad (1.23)$$

Still another equivalent condition for conditional independence $X_A \perp\!\!\!\perp X_B | X_C$ is that for all values x_A, x_B, x_C , we have

$$p(x_A | x_B, x_C) = p(x_A | x_C) \quad (1.24)$$

We will be interested in deductions over independence relationships. For example, given a sequence of triples $\{(A_i, B_i, C_i)\}_i$ and distinct triple (A, B, C) , if $X_{A_i} \perp\!\!\!\perp X_{B_i} | X_{C_i}$, can we infer $X_A \perp\!\!\!\perp X_B | X_C$? As an example, it is true that $\{X_1, X_2\} \perp\!\!\!\perp Y | Z \Rightarrow \{X_1\} \perp\!\!\!\perp Y | Z$.

What happens when the sets are not disjoint? Consider if A and C are not disjoint. For example, what does $\{X_1, X_2\} \perp\!\!\!\perp Y | \{Z, X_2\}$ mean? Since

$$p(x_1, x_2 | z, x_2) = \frac{p(x_1, x_2, z, x_2)}{p(z, x_2)} = \frac{p(x_1, x_2, z)}{p(z, x_2)} = p(x_1 | x_2, z)$$

this is the same as $X_1 \perp\!\!\!\perp Y | \{Z, X_2\}$ since

$$\begin{aligned} p(x_1, y | z, x_2) &= p(x_1, x_2, y | z, x_2) = p(x_1, x_2 | z, x_2)p(y | z, x_2) \\ &= p(x_1 | z, x_2)p(y | z, x_2) \end{aligned}$$

What if A and B are not disjoint? For example, does $\{X_1, X_2\} \perp\!\!\!\perp X_2$ make sense? What this would imply is that

$$p(x_1, x_2) = p(x_1, x_2, x_2) = p(x_1, x_2)p(x_2) \Rightarrow p(x_2) = p(x_2)p(x_2)$$

which can only happen when X_2 is a constant random variable (only one value with non-zero probability). To give this intuitive meaning, if X_2 is a random variable with no marginal uncertainty, then knowing its value does not further reduce the uncertainty since the value is already known.

There are many properties of conditional independence, the first four of these, defined on sets of random variables W, X, Y, Z as follows:

- Property C1: $X \perp\!\!\!\perp Y | Z \Rightarrow Y \perp\!\!\!\perp X | Z$. In other words, multiplication (influence) is commutative.

- Property C2: $X \perp\!\!\!\perp Y|Z$ and $U = h(X) \Rightarrow U \perp\!\!\!\perp Y|Z$. Intuition is that U has less information than X , so it is not going to tell us any more about Y than X is, conditional or unconditional on Z .

This applies to subsets as well. I.e., it immediately follows from this property that if $A' \subseteq A$ and $B' \subseteq B$ then $X_{A'} \perp\!\!\!\perp X_{B'}|X_C$ whenever $X_A \perp\!\!\!\perp X_B|X_C$.

- Property C3: $X \perp\!\!\!\perp Y|Z$ and $U = h(X) \Rightarrow X \perp\!\!\!\perp Y|(Z, U)$. Knowing part of X or some function of X (i.e., $h(X)$) in addition to Z will not tell us any more about Y given Z than X already tells us about Y .

As an example, suppose $Y' \subset Y$, then we get that $X \perp\!\!\!\perp Y|(Z, Y')$ whenever $X \perp\!\!\!\perp Y|(Y)$.

This property corresponds to the fact that you can't get something from nothing, or may be seen as a consequence of the information processing inequality [97], where additional processing of random variables can only lose information that is contained in the original source.

This also relates to the example above where we discussed $\{X_1, X_2\} \perp\!\!\!\perp Y|\{Z, X_2\}$. In other words, $\{X_1, X_2\} \perp\!\!\!\perp Y|Z \Rightarrow \{X_1, X_2\} \perp\!\!\!\perp Y|\{Z, X_2\} \Rightarrow X_1 \perp\!\!\!\perp Y|\{Z, X_2\}$. This means that if $X \perp\!\!\!\perp \{Y, Z\}$ then $X \perp\!\!\!\perp Y|Z$ and $X \perp\!\!\!\perp Z|Y$.

- Property C4: $X \perp\!\!\!\perp Y|Z$ and $X \perp\!\!\!\perp W|\{Y, Z\} \Rightarrow X \perp\!\!\!\perp \{W, Y\}|Z$. Marginal case, $X \perp\!\!\!\perp Y$ and $X \perp\!\!\!\perp W|Y \Rightarrow X \perp\!\!\!\perp \{W, Y\}$.

This is just like the chain rule of conditional mutual information, where $I(X; W, Y|Z) = I(X; Y|Z) + I(X; W|Y, Z)$. Since mutual information is non-negative, if it is the case that $I(X; Y|Z) = I(X; W|Y, Z) = 0$, then so must $I(X; W, Y|Z)$.

A graph (which we have not yet spoken about) helps to visualize this property.

Note that C2, C3, and C4 together mean that $X \perp\!\!\!\perp \{W, Y\} \iff X \perp\!\!\!\perp Y$ and $X \perp\!\!\!\perp W|Y$. And conditioned on Z , we have that $X \perp\!\!\!\perp \{W, Y\}|Z \iff X \perp\!\!\!\perp Y|Z$ and $X \perp\!\!\!\perp W|\{Y, Z\} \iff X \perp\!\!\!\perp W|Z$ and $X \perp\!\!\!\perp Y|\{W, Z\}$.

Another property *sometimes holds* (i.e., when the distributions are positive) but not always. This property, C5, is sort of like a converse to C3.

- Property C5 (general form): $X \perp\!\!\!\perp Y|\{Z, U\}$ and $X \perp\!\!\!\perp Z|\{Y, U\} \Rightarrow X \perp\!\!\!\perp \{Y, Z\}|U$
- Property C5 (simpler form): $X \perp\!\!\!\perp Y|Z$ and $X \perp\!\!\!\perp Z|Y \Rightarrow X \perp\!\!\!\perp \{Y, Z\}$

Since C5 holds only in the case of positive distributions, we can find a counter example where it fails when distributions are sparse. Here is such an example when C5 does not hold.

Example 5 (C5 does not hold). Let $X = Y = Z$ and $p(X = 1) = p(X = 0) = 1/2$, implying that $p(X = Y = Z = 1) = p(X = Y = Z = 0) = 1/2$, and all other assignments have probability 0. But then, we have

$$p(x|z) = p(x, z)/p(z) = (1/2)\mathbf{1}\{x = z\}/(1/2) = \mathbf{1}\{x = z\}$$

and similarly $p(y|z) = \mathbf{1}\{y = z\}$. We also have that

$$p(x, y|z) = p(x, y, z)/p(z) = (1/2)\mathbf{1}\{x = y = z\}/(1/2) = \mathbf{1}\{x = y = z\}$$

But we also have that

$$p(x, y|z) = \mathbf{1}\{x = y = z\} = \mathbf{1}\{x = y\}\mathbf{1}\{y = z\} = p(x|z)p(y|z)$$

so $X \perp\!\!\!\perp Y|Z$ (and similarly $X \perp\!\!\!\perp Z|Y$). But we do not have that $X \perp\!\!\!\perp \{Y, Z\}$ since X is determined given either Y , Z , or both.

Theorem 6. When a density is strictly positive (i.e., $p > 0$), then property C5 holds.

Proof. Partition the random variables into three sets X, Y, Z . We have that $p(x, y, z) > 0$ and $X \perp\!\!\!\perp Y | Z$, and $X \perp\!\!\!\perp Z | Y$. Then \exists functions k, ℓ, g, h such that:

$$p(x, y, z) = k(x, z)\ell(y, z) = g(x, y)h(y, z)$$

where k, ℓ, g, h are strictly positive. This means that:

$$g(x, y) = \frac{k(x, z)\ell(y, z)}{h(y, z)}$$

We fix $z = z_0$ where z_0 is arbitrary. Then $g(x, y) = \pi(x)f(y)$ where $\pi(x) = k(x, z_0)$ and $f(y) = \ell(y, z_0)/h(y, z_0) \forall z_0$. Then, $p(x, y, z) = g(x, y)h(y, z) = \pi(x)(f(y)h(y, z))$, or that $X \perp\!\!\!\perp \{Y, Z\}$. \square

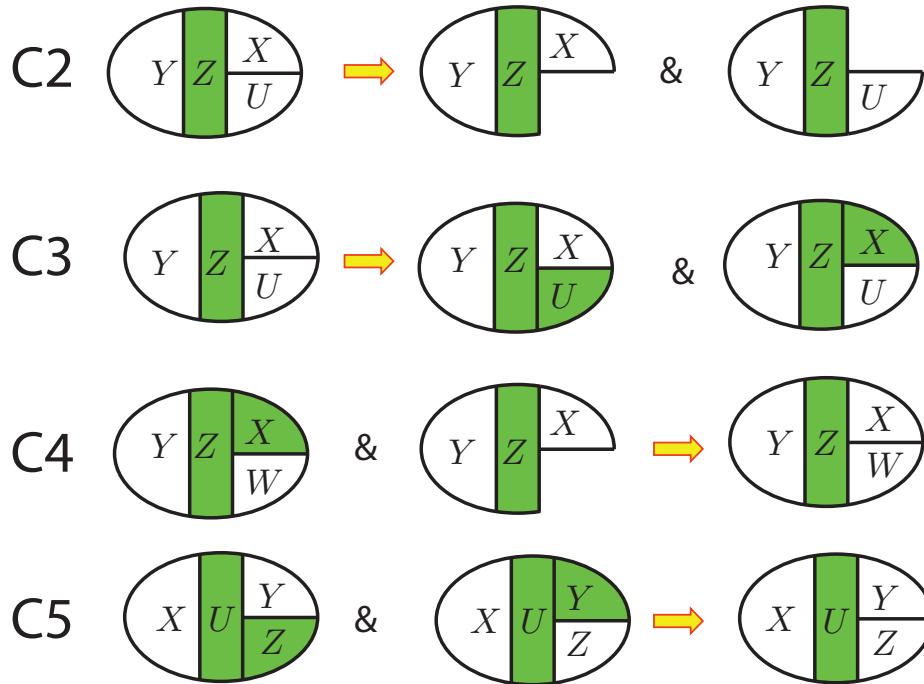


Figure 1.2: Pearl's eggs - summarizing the 5 deductive properties of conditional independence

The 5 deductive properties of conditional independence is nicely summarized by Pearl's eggs (Figure 1.2).

TODO: give more description here.

1.3 Examples of Graphical Models

There are many types of graphical models. Some require a directed graph, some require an undirected graph, some a bipartite graph, and some require a graph that might be both directed and undirected. Each type of graphical model has a set of rules. We'll be discussing these rules in later chapters but for now we just give a brief overview of different graphical models and description of what they represent.

A graphical model is a graph $G = (V, E)$ consisting of a set of nodes V and a set of edges $E \subseteq V \times V$, so each edge is a pair of nodes, i.e. $\forall e \in E$, there is an (i, j) pair corresponding to e . A graph represents

a family of probability distributions over a set of random variables where there is one random variable per node in the graph. Typically, a node in a graph represents a scalar random variable, and a vector random variable is represented by a set of graph nodes, but sometimes an individual graph node might itself represent a vector random variable (it will be clear from the context which is which).

There are many types of graphical model. This includes:

- Markov Random Fields: a form of undirected graphical model relatively simple to understand their semantics also, log-linear models, Gibbs distributions, Boltzmann distributions, many “exponential models”, conditional random fields (CRFs), etc.
- Bayesian networks: a form of directed graphical model originally developed to represent a form of causality, but not ideal for that (they still represent factorization) Semantics more interesting (but trickier) than MRFs
- Factor Graphs: the assembly language models for factorization properties came out of coding theory community (LDPC, Turbo codes).
- Chain graphs: Hybrid between Bayesian networks and MRFs A set of clusters of undirected nodes connected as directed links Not as widely used, but very powerful.
- Ancestral graphs are another type of directed model, distinct from Bayesian networks (although their semantics is such that they are, for all intents and purposes, identical to Markov random fields).

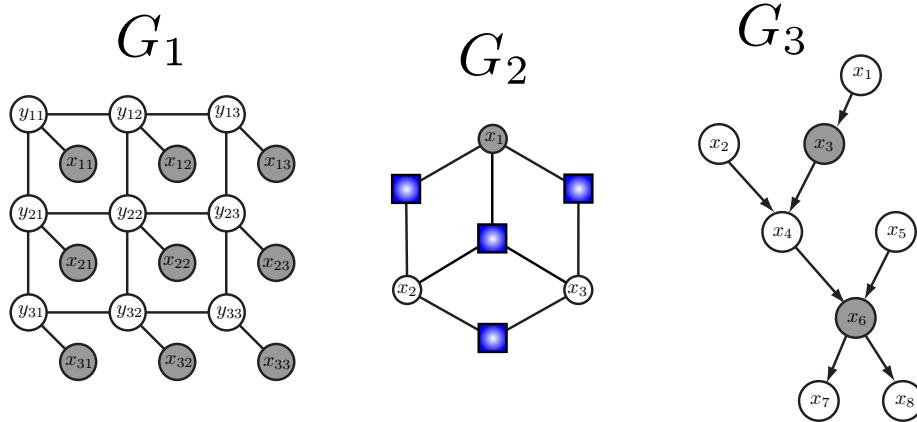


Figure 1.3: Three types of graphical models are shown here. On the left we have a Markov random field, which is an undirected model. In the middle we have a factor graph. The right shows a Bayesian networks. In each case, when the value of a random variable is known, the corresponding node in the graph is shaded, although sometimes different shadings can mean different things (e.g., middle graph).

There are also particular types of graphical model that apply to certain types of data, such temporal data. “Dynamic graphical models” describe the general family and this includes hidden Markov models, dynamic Bayesian networks, temporal conditional random fields, and so on. Such temporal models is our main topic but the subject of this chapter is general to any graphical model, temporal or otherwise. Other forms of models are template-based, and these includes things like probabilistic relational models, and Markov logic models. These will be discussed later as well.

What follows are a number of simple examples of specific graphical models. When possible, the corresponding statistical technique will also be listed.

1.3.1 Basic Models

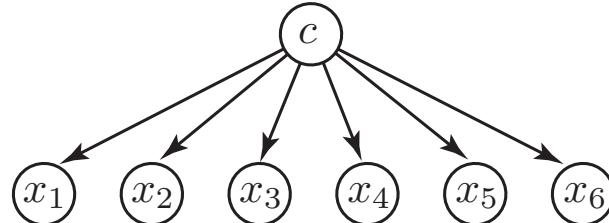
The simplest possible graphical model is a graph with a single node representing one scalar random variable. The graph of course does not tell us anything about the random variable other than that it exists. A single node graphical model also does not tell us if the random variable is a scalar variable, or if it is a vector random variable — an in this latter case, it would not tell us anything about the relationships between elements in the vector.

A vector variable can be described more explicitly with a graphical model as shown in the following figure.



On the left of the figure is a Bayesian network and on the right is a Markov random variable, both of which are graphical models that specify constraints on any distribution $p(x_1, \dots, x_5)$ they are supposed to represent. The specific constraints depend on the semantics of the graphical model — in this particular case, it turns out, the set of constraints are identical even though the two graphical models look somewhat different.

A slightly more complicated and realistic example is that of a Naïve Bayes classifier which is a joint model of $p(c, x_1, \dots, x_N)$ where there are N feature variables and a class variable c and where the features are independent given the class variable. Given a set of values for the features, the goal is to infer the class. The Bayesian network for a naïve Bayes classifier is shown next:



Even given the apparently catastrophic independence assumption made by this model, it does remarkably well in practice.

Another way we can slightly increase the complexity of a single random variable is to represent a mixture distribution. A mixture distribution is a model for $p(x)$ that consists of a convex weighted sum of mixture components. That is, a general mixture distribution is:

$$p(x) = \sum_i \alpha_i p(x|i) \quad (1.25)$$

where $\sum_i \alpha_i = 1$ and $\alpha_i \geq 0 \ \forall i$. This can easily be represented as a graphical model as follows:



We note here that the graph represents a joint distribution of $p(x, c)$ where c is a mixture variable, so that $p(x) = \sum_i p(c_i)p(x|c_i)$ and where we identify $\alpha_i = c_i$. Therefore, the mixture distribution above is the marginal over x form of any joint distribution over two variables x and c .

Conditional mixture generalizations, where x requires z , are quite easy to obtain using the graph $Z \rightarrow C \rightarrow X$, leading to the equation:

$$p(x|z) = \sum_i p(x, C = i, z) = \sum_i p(C = i|z)p(x|C = i)$$

Texts exist, such as [426, 298], that describe the properties of mixture distributions, most of which can be described using graphs in this way.

1.3.2 Principle Component Analysis and its generalizations

Our second example of graphical models includes techniques that are commonly used to transform feature vectors from their more raw format into something that might be more amenable for use in a pattern classifier. These include principle component analysis (PCA) (also called the Karhunen-Loéve or KL transform), factor analysis (FA), and independent component analysis (ICA). These techniques, in fact, themselves can be given a statistical interpretation. The PCA technique, for example, has historically been presented without any probabilistic interpretation. Interestingly, when given such an interpretation and seen as a graph, PCA has exactly the same graph structure as both FA and ICA

The graphs in this and the next section show nodes both for random variables and their parameters. For example, if X is Gaussian with mean μ , a μ node might be present as a parent of X . Parameter nodes will be indicated as shaded yellow rippled circles. For our purposes, these nodes constitute constant random variables whose probability score is not counted (they are conditional-only variables, always to the right of the conditioning bar in a probability equation). In a more general Bayesian setting [190, 189, 371], however, these nodes would be true random variables with their own distributions and hyper-parameters.

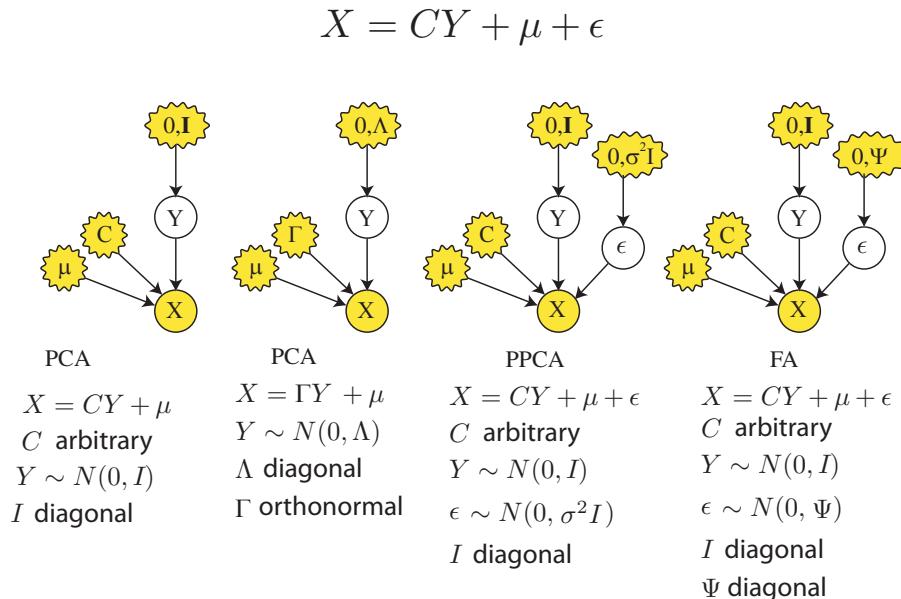


Figure 1.4: Left two graphs: two views of principle components analysis (PCA); middle: probabilistic PCA; right: factor analysis (FA). In general, the graph corresponds to the equation $X = CY + \mu + \epsilon$, where $Y \sim \mathcal{N}(0, \Lambda)$ and $\epsilon \sim \mathcal{N}(0, \Psi)$. X is a random conditional Gaussian with mean $CY + \mu$ and variance $CACT^T + \Psi$. With PCA, $\Psi = 0$ so that $\epsilon = 0$ with probability 1. Also, either (far left) $\Lambda = I$ is the identity matrix and C is general, or (second from left) Λ is diagonal and $C = \Gamma$ is orthonormal. With PPCA, $\Psi = \sigma^2 I$ is a spherical covariance matrix, with diagonal terms σ^2 . With FA, Ψ is diagonal. Other generalizations are possible, but they can lead to an indeterminacy of the parameters.

Starting with PCA, observations of a d -dimensional random vector X are assumed to be Gaussian with mean μ and covariance Σ . The goal of PCA is to produce a vector Y that is a zero-mean uncorrelated linear transformation of X . The spectral decomposition theorem [404] yields the factorization $\Sigma = \Gamma \Lambda \Gamma^T$,

where Γ is an orthonormal rotation matrix (the columns of Γ are orthogonal eigenvectors, each having unit length), and Λ is a diagonal matrix containing the eigenvalues that correspond to the variances of the elements of X . A transformation achieving PCA's goal is $Y = \Gamma^T(X - \mu)$. This follows since $E[YY^T] = \Gamma^T E[(X - \mu)(X - \mu)^T] \Gamma = \Gamma^T \Sigma \Gamma = \Lambda$. Alternatively, a spherically distributed Y may be obtained by the following transformation: $Y = (\Lambda^{-1/2}\Gamma)^T(X - \mu) = C^T(X - \mu)$ with $C = \Lambda^{-1/2}\Gamma$.

Solving for X as a function of Y yields the following:

$$X = \Gamma Y + \mu$$

Slightly abusing notation, one can say that $X \sim \mathcal{N}(\Gamma Y + \mu, 0)$, meaning that X , conditioned on Y , is a linear-conditional constant "Gaussian" — i.e., a conditional-Gaussian random variable with mean $\Gamma Y + \mu$ and zero variance.³ In this view of PCA, Y consists of the latent or hidden "causes" of the observed vector X , where $Y \sim \mathcal{N}(0, \Lambda)$, or if the C -transformation is used above, $Y \sim \mathcal{N}(0, I)$ where I is the identity matrix. In either case, the variance in X is entirely explained by the variance within Y , as X is simply a linear transformation of these underlying causes. PCA transforms a given X to the most likely values of the hidden causes. This is equal to the conditional mean $E[Y|X] = \Gamma^T(X - \mu)$ since $p(y|x) \sim \mathcal{N}(\Gamma^T(x - \mu), 0)$.

The two left graphs of Figure 1.4 show the probabilistic interpretations of PCA as a graphical model, where the dependency implementations are all linear. The left graph corresponds to the case where Y is spherically distributed. The hidden causes Y are called the "principal components" of X . It is often the case that only the components (i.e., elements of Y) corresponding to the largest eigenvalues of Σ are used in the model, the other elements of Y are removed, so that Y is k -dimensional with $k < d$. There are many properties of PCA [295] — for example, using the principle k elements of Y leads to the smallest reconstruction error of X in a mean-squared sense. Another notable property (which motivates factor analysis below) is that PCA is not scale invariant — if the scale of X changes (say by converting from inches to centimeters), both Γ and Λ will also change, leading to different components Y . In this sense, PCA explains the variance in X using only variances found in the hidden causes Y .

Factor analysis (the right-most graph in Figure 1.4) is only a simple modification of PCA — a single random variable is added onto the PCA equation above, yielding:

$$X = CY + \mu + \epsilon$$

where $Y \sim \mathcal{N}(0, I)$, and $\epsilon \sim \mathcal{N}(0, \Psi)$ with Ψ a non-negative diagonal matrix. In factor analysis, C is the *factor loading* matrix and Y the *common factor* vector. Elements of the residual term $\epsilon = X - CY - \mu$, are called the *specific factors*, and account both for noise in the model and for the underlying variance in X . In other words, X possesses a non-zero variance, even conditional on Y , and Y is constrained to be unable to explain the variance in X since Y is forced to have I as a covariance matrix. C , on the other hand, is compelled to represent just the correlation between elements of X irrespective of its individual variance terms, since correlation can not be represented by ϵ . Therefore, unlike PCA, if the scale of an element of X changes, the resulting Y will not change as it is ϵ that will absorb the change in X 's variance. As in PCA, it can be seen that in FA X is being explained by underlying hidden causes Y , and the same graph (Figure 1.4) can describe both PCA and FA.

Probabilistic PCA (PPCA) (second from the right in Figure 1.4) [425, 372], while not as widely used as the others, is only a simple modification to FA, where $\Psi = \sigma^2 I$ is constrained so that ϵ is a spherically-distributed Gaussian.

In all of the models above, the hidden causes Y are uncorrelated Gaussians, and therefore are marginally independent. Any statistical dependence between elements of X exist only in how they are jointly dependent on one or more of the hidden causes Y . It is possible to use a GM to make this marginal Y dependence

³This of course corresponds to a degenerate Gaussian, as the covariance matrix is singular.

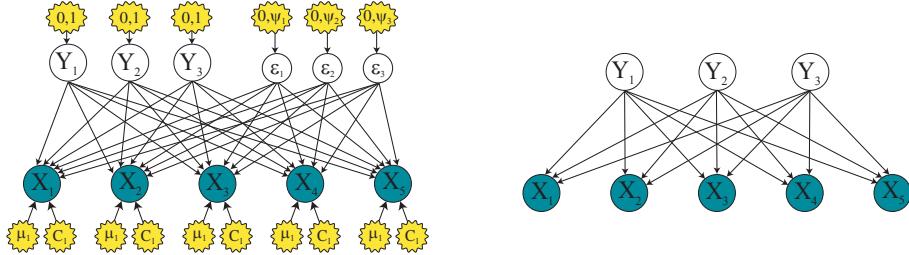


Figure 1.5: Left: A graph showing the explicit scalar variables (and therefore their statistical dependencies) for PCA, PPCA, and FA. The graph also shows the parameters for these models. In this case, the dependencies are linear and the random variables are all Gaussian. Right: The graph for PCA/PPCA/FA (parameters not shown) which is the same as the graph for ICA. For ICA, the implementation of the dependencies and the random variable distributions can be arbitrary, different implementations lead to different ICA algorithms. The key goal in all cases is to explain the observed vector X with a set of statistically independent causes Y .

explicit, as is provided on the left in Figure 1.5 where all nodes are now scalars. In this case, $Y_j \sim \mathcal{N}(0, 1)$, $\epsilon_j \sim \mathcal{N}(0, \psi_j)$, and $p(x_i) \sim \mathcal{N}(\sum_j C_{ij}y_j + \mu_i, \psi_i)$ where $\psi_j = 0$ for PCA.

The PCA/PPCA/FA models can be viewed without the parameter and noise nodes, as shown on the right in Figure 1.5. This, however, is the general model for independent component analysis (ICA) [24, 233], another method that explains data vectors X with independent hidden causes. Like PCA and FA, a goal of ICA is to first learn the parameters of the model that explain X . Once done, it is possible to find Y , the causes of X , that are as statistically independent as possible. Unlike PCA and FA, however, dependency implementations in ICA neither need to be linear nor Gaussian. Since the graph on the right in Figure 1.5 does not depict implementations, the vector Y can be any non-linear and/or non-Gaussian causes of X . The graph insists only that the elements of Y are marginally independent, leaving alone the operations needed to compute $E[Y|X]$. Therefore, ICA can be seen simply as supplying the mechanism for different implementation of the dependencies used to infer $E[Y|X]$. Inference can still be done using the standard graphical-model inference machinery, described in future chapters.

With graphs it is easy to understand all of these techniques, and simple structural or implementation changes can lead to dramatically different statistical procedures.

1.3.3 Discriminative models LDA/QDA/MDA/QMDA

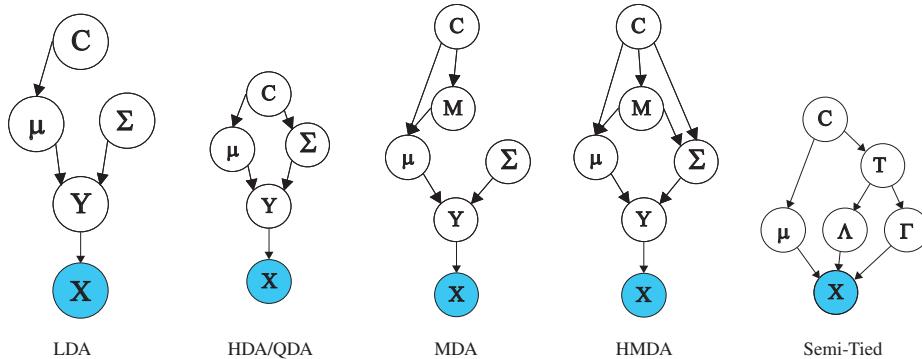


Figure 1.6: Linear discriminant analysis (left), and its generalizations.

When the goal is pattern classification [131] (deciding amongst a set of classes for X), it is often beneficial to first transform X to a space spanned neither by the principle nor the independent components, but rather to a space that best discriminatively represents the classes. This may be done either linearly (below) or non-linearly (not covered here). Let C be a variable that indicates the class of X , $|C|$ the cardinality of C . As above, a linear transformation can be used, but in this case it is created to maximize the between-class covariance while minimizing the within-class covariance in the transformed space. Specifically, the goal is to find the linear transformation matrix A to form $Y = AX$ that maximizes $\text{tr}(BW^{-1})$ [160] where

$$W = \sum_i p(C = i) E_{p(y|C=i)}[(Y - \mu_y^i)(Y - \mu_y^i)^T]$$

and

$$B = \sum_i p(C = i)(\mu_y^i - \mu_y)(\mu_y^i - \mu_y)^T$$

where μ_y^i is the class conditional mean and μ_y is the global mean in the transformed space. This is a multi-dimensional generalization of Fisher's original linear discriminant analysis (LDA) [146].

LDA can also be seen as a particular statistical modeling assumption about the way in which observation samples X are generated. In this case, it is assumed that the class conditional distributions in the transformed space $P(Y|C = i)$ are Gaussians having priors $P(C = i)$. Therefore, Y is a mixture model $p(y) = \sum_i p(C = i)p(y|C = i)$, and classification of y is optimally performed using the posterior:

$$p(C = i|y) = \frac{p(y|i)p(i)}{\sum_j p(y|j)p(j)}$$

For standard LDA, it is assumed that the Gaussian components $p(y|j) = \mathcal{N}(y; \mu_j, \Sigma)$ all have the same covariance matrix, and are distinguished only by their different means. Finally, it is assumed that there is a linear transform relating X to Y . The goal of LDA is to find the linear transformation that maximizes the likelihood $P(X)$ under the assumptions given by the model above. The statistical model behind LDA can therefore be graphically described as shown on the far left in Figure 1.6.

There is an intuitive way in which these two views of LDA (a statistical model or simply an optimizing linear transform) can be seen as identical. Consider two class-conditional Gaussian distributions with identical covariance matrices. In this case, the discriminant functions are linear, and effectively project any unknown sample down to an affine set⁴, in this case a line, that points in the direction of the difference between the two means [131]. It is possible to discriminate as well as possible by choosing a threshold along this line — the class of X is determined by the side of the threshold X 's projection lies.

More generally, consider the affine set spanned by the means of $|C|$ class-conditional Gaussians with identical covariance matrices. Assuming the means are distinct, this affine set has dimensionality $\min\{|C| - 1, \dim(X)\}$. Discriminability is captured entirely within this set since the decision regions are hyperplanes orthogonal to the lines containing pairs of means [131]. The linear projection of X onto the $|C| - 1$ dimensional affine set Y spanned by the means leads to no loss in classification accuracy, assuming Y indeed is perfectly described with such a mixture. If fewer than $|C| - 1$ dimensions are used for the projected space (as is often the case with LDA), this can lead to a dimensionality reduction algorithm that has a minimum loss in discriminative information. It is shown in [255] that the original formulation of LDA ($Y = AX$ above) is identical to the maximum likelihood linear transformation from the observations X to Y under the model described by the graph shown on the left in Figure 1.6.

When LDA is viewed as graphical model, it is easy to extend it to more general techniques. The simplest extension allows for different covariance matrices so that $p(x|i) = \mathcal{N}(x; \mu_i, \Sigma_i)$, leading to the GM second

⁴An affine set is simply a translated subspace [366].

from the left in Figure 1.6. This has been called quadratic discriminant analysis (QDA) [131, 299], because decision boundaries are quadratic rather than linear, or heteroscedastic discriminant analysis (HDA) [255], because covariances are not identical. In the latter case, it is assumed that only a portion of the mean vectors and covariance matrices are class specific — the remainder corresponds in the projected space to the dimensions that do not carry discriminative information.

Further generalizations to LDA are immediate. For example, if the class conditional distributions are Gaussians mixtures, every component sharing the same covariance matrix, then mixture discriminant analysis (MDA) [186] is obtained (3rd from the left in Figure 1.6). A further generalization yields what could be called be called heteroscedastic MDA, as described 2nd from the right in Figure 1.6. If non-linear dependencies are allowed between the hidden causes and the observed variables, then one may obtain non-linear discriminant analysis methods, similar to the neural-network feature preprocessing techniques [148, 238, 193] that have recently been used.

Taking note of the various factorizations one may perform on a positive-definite matrix [184], a concentration matrix K within a Gaussian distribution can be factored as $K = \Lambda^T \Gamma \Lambda$. Using such a factorization, each Gaussian component in a Gaussian mixture can use one each from a shared pool of Λ s and Γ s, leading to what are called semi-tied covariance matrices [165, 454]. Once again, this form of tying can be described by a GM as shown by the far right graph in Figure 1.6.

1.3.4 Grid models

Image processing requires dealing with a grid of pixels and one is often interested in statistical inference procedures over such grids. For example, an image-processing application might be to take an image and segment it into regions that correspond to different textures and where certain consistency constraints should be enforced over any segmentation considered. One possible model is shown in Figure 1.3 on the left. The graph shows two 2-dimensional grids, one which is indexed by nodes x_{ij} and the other y_{ij} . One possible interpretation of this graph is that of noise removal in high-ISO settings for digital cameras. In such case, the observations (i.e., the x_{ij} values) are the digital image, and the hidden variables (the y_{ij} values) are the noise-removed version of the image. The goal is to find the most likely assignment to all of the y_{ij} values given the x_{ij} values. Another possible application of this model is image segmentation where each y_{ij} can take on small integer values, where the inferred y_{ij} indicates the segment of the pixel ij in the image.

1.3.5 Dynamic models

Dynamic graphical models are graphical models for which there exists a static template of a fixed size and which can be expanded to fit any desired length T . The static templates are usually relatively simple, and have a finite number of parameters. A key characterization of most static models is that the number of model parameters is the same regardless of the duration model is expanded.

Such models include: HMMs, hierarchical HMMs, input-output HMMs, CRFs, and DBNs.

Much more discussion will be given on dynamic graphical models starting in Chapter .

Chapter 2

Graphical Models Overview

In this chapter, we give a brief but broad overview of graphical models, and in doing so begin to use the notation used in this document. Some of this material is redundant with the next few chapters, where we go into much more detail.

2.1 Families and sub-Families

We'll be talking about families of distributions quite a bit in this document, so to put first things first, let's talk a bit about families.

Suppose we have a collection of n random variables X_1, X_2, \dots, X_n that are governed by a probability distribution $p(x_1, x_2, \dots, x_n)$. This single distribution can be seen as a family of size one. It's more interesting to talk about families of larger size, so let's say that each of p_1, p_2, \dots, p_K are distributions over those n random variables. Then the set of distributions $\{p_1, p_2, \dots, p_K\}$ can be seen as a family (or a set) of distributions. Here we see that we have a countable number of distributions, so the set is finite and countable. Alternatively, we might have a countably infinite set $\{p_1, p_2, \dots\}$, or even an uncountable set — if $\{p_\alpha\}_{\alpha \in S}$ is a family distribution, where each p_α is a distribution parameterized by a continuous parameter α (possibly vector-valued), and S is an uncountable set.

Perhaps the easiest way to view a family is using Venn diagrams as shown in Figure 2.1. For a given fixed number of random variables n , we use circles (more generally an ellipse) to represent all probability distributions over those n random variables within the family. Sets A and B in the figure are such a family, and their intersection shows the area corresponding to members in both families. On the right of the figure, we see three families, A , B , and C , with different shaded regions corresponding to members in one or more of the families. A sub-family of a family is a subset of the set of distributions. For example, if $D \subseteq A$, then any member of D is also a member of A .

Families are not very interesting unless the members of the family share certain properties. One such property (used by graphical models) is that of factorization. For example, consider all distributions where all random variables are mutually independent, or the “all independent family.” That is, any p in the family must be such that $p(x_{1:n}) = \prod_{i=1}^n p(x_i)$ where for each i , $p(x_i) = \sum_{x_1, x_2, \dots, x_{i-1}, x_{i+1}, \dots, x_n} p(x_{1:n})$. This implies that for any $A \subseteq [n]$, have that $p(x_A) = \prod_{a \in A} p(x_a)$. Any distribution p that does not obey this factorization is not a member of the all independent family.

There are many more sophisticated factorization properties that can be specified as well, and this is what graphical models will help us with (as we see below). Let's say that \mathcal{C} is a clustering of the indices,

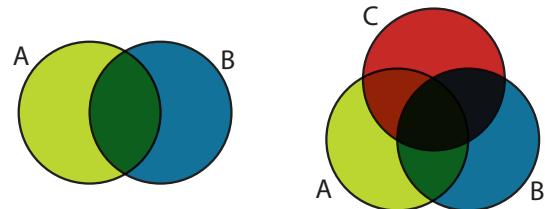


Figure 2.1: Venn diagrams of sets, in this case each set represents a family of distributions.

meaning that for all $C \in \mathcal{C}$, $C \subseteq [n]$ and that $\bigcup_{C \in \mathcal{C}} C = [n]$. Then it might be that for any p in a family, $p = \prod_{C \in \mathcal{C}} p(x_C)$ where $p(x_C) = \sum_{x_{V \setminus C}} p(x_{1:n})$. Like before, any distribution that does not factor in this way is not a member of the family.

It should be clear that a family is a set of probability distributions over n random variable that obey certain properties or rules.

Still other forms of properties, besides factorization, are also possible and that can define families. For example, suppose that the random variables are all binary ($\{0, 1\}$) valued. Then if $p(x_{1:n}) = \frac{1}{Z} \exp(-f(x_{1:n}))$ where f is a submodular function [159]. Submodular functions do not require any factorization properties at all (meaning that it need not be the case that f can be represented as a sum over smaller blocks). But such distributions also have special properties, have special efficiency properties, and hence constitute a family.

Of course, a restriction to family would not be useful if it was the case that being a member of a family didn't incur certain advantages or features. Graphical models are one way of specifying families and sub-families of sets of probability distributions. We will see that these properties take the form of *Markov properties*, meaning that the factorization properties required of any distribution is encoded by a graph, where the graph is mathematically precise as to what properties are required by the graph. By a set of “properties” we mean that a given graph has a formal semantics that states how precisely it can be determined whether or not a given p is a member of the family. While it is not possible to designate all possible useful families with graphical models (e.g., the submodular case above is not characterized by any graph), certainly a large class of them can be, and many of them are quite attractive in a number of ways.

2.2 What are graphical models anyway?

A picture is worth a thousand words, but how many words is a mathematical equation worth? Mathematical equations are like pictures — they compress a great deal of information into a precise, concise, unambiguous, and symbolic form. This is quite different than human language, where even good writing can contain ambiguities and contradictions. When using human language, avoiding ambiguities often requires a verbosity that becomes quite tedious (consider legal writing). Avoiding contradictions benefits from using a symbolic formalism that inherently discourages them. In the field of science, we often speak in the language of equations. The development of the universal mathematical language of equations has fostered almost every scientific and engineering advance humankind have seen thus far.

Perhaps then a picture is worth a thousand equations, but are they really comparable? Unlike equations, pictures are often only rough sketches, they are vague and do not typically have the precision and specificity of mathematical equations. A pictures can compress a great deal of information in order to communicate an idea rapidly, and they often help provide us with intuition about an otherwise complicated mathematical model, but in doing so they often make the details fuzzy. While this helps with general understanding, it might not help with detailed understanding. A picture can give us the idea of a concept, but pictures as they are often used do not have the formal semantics necessary to express properties with precision.

Suppose however that we formalize pictures so that they retain their expressiveness while also acquiring the property of precision that mathematical equations possess. With such a picture, perhaps it would be possible to compress thousands, if not millions (or more), of equations into an equally precise and yet easy to understand visual form. Indeed, how many equations would such a picture be worth?

One way of understanding a graphical model is that it is exactly such a formalism. A graphical model is a picture (a graph) that says something mathematical. A graphical model can summarize an enormous number of equations in a concise and precise way. A graphical model does not suffer accuracy loss as would some less formal general picture.

Graphical models are a formalism whereby a graph $G = (V, E)$ and a set of properties \mathcal{M} determine an

(often) infinite size family of probability distributions. There are many different types of graphs and many different types of properties, each one potentially determining a different family. Common amongst almost all forms of graphical models is the fact that one can develop algorithms that work for all members of a given family by observing only a graph and its properties. This is an amortization property of graphical models, and means that solving problems (such as spending a significant amount of computer time up front approximating an NP-complete optimization problem) might become worthwhile only because the solution can be applied many times over.

A graphical model consists of a graph $G = (V, E)$ (where V is a set of nodes and E is a set of edges) along with a set of Markov properties \mathcal{M} that determine a family of probability distributions. While such a family will of infinite size, **every** member of the family must satisfy a set of constraints, and the constraints are encoded by the Markov properties applied to the graph. Therefore, not all distributions lie within the family. There are many different types of graphs and many different types of Markov properties (henceforth we may refer to Markov properties as just “properties”).

Given a probability distribution $p(x_V)$ over a set of random variables $X_V = \{X_{v_1}, X_{v_2}, \dots, X_{v_N}\}$, where $N = |V|$, one often wishes to compute quantities of the form $p(x_S|x_U)$ where $S, U \subseteq V$ are disjoint. Such computations are known as *probabilistic inference*, or sometimes just *inference*. This is an important problem because computing such probabilistic quantities lie at the core of many important statistical inference and machine learning algorithms and their applications. For example, amortization means that if it is known that the distribution p is a member of a graphical model’s family, and if we already have an algorithm that does inference for any member of this family, we then automatically have a solution for $p(x_S|x_U)$.

There are many may different ways of performing inference, some of which are extremely slow and some of which can be much faster. Obviously we would like to find the fastest cheapest way of computing inference quantities, and moreover we would like to find this fastest way automatically (by computer). Unfortunately it can be shown that finding the fastest way of doing inference is an NP-complete optimization problem. We should note here that by “fastest” we mean in general both the fastest in terms of time, but also the best in terms of resource utilization (e.g., memory usage).

There are also ways of performing inference approximately. In general, there is an accuracy-memory-speed tradeoff when doing inference. I.e., you can do inference faster and with less memory if you are willing to sacrifice accuracy, and vice versa. In this section, we will primarily refer to doing inference exactly and even here there is a memory-speed tradeoff. Many of the command line options to GMTK allow one to explore this tradeoff in dynamic graphical models (to be defined in §8).

GMTK is meant for graphical models that are amenable to sequential data (and thus are called dynamic graphical models), which includes speech and language strings, biological sequences, and economic series. At the heart of dynamic models, however, lies “static” graphical models so we will briefly review these first.

2.2.1 Static Graphical Models

An overview of (static) graphical models is given in this documentation. For more detail, however, see the many excellent texts such as [258, 208, 249, 219].

Let $p = p(x_1, x_2, \dots, x_N)$ be a probability distribution over $N = |V|$ random variables and let \mathcal{U} be the unrestricted family (or set) of all such distributions. A graphical model consists of a graph $G = (V, E)$ and a set of properties \mathcal{M} (often called *Markov properties* [258]) that together determine a sub-family $\mathcal{F}(G, \mathcal{M}) \subseteq \mathcal{U}$ of probability distributions. There are many types of graphical model (such as Markov random fields (MRF) or Bayesian networks (BN), see [336, 207, 258, 361, 254] for further information), and each type of graphical model has its own *semantics* [45] that govern how the graph specifies a set of factorization constraints on multi-variate probability distributions. The type of graphical model is determined by both the set of allowable graphs (e.g., directed or undirected, acyclic or not, bipartite, etc.) and the set of properties.

One such simple graphical model is the Markov random field (MRF), where any undirected graph over N vertices is allowed (one vertex for each random variable), and p is a member of the family whenever p 's conditional independence properties obey the separation properties \mathcal{M}^{sep} in the graph. Given three sets of graph vertices $A, B, C \subset V(G)$, it is said that A is separated from B by C in G if all paths (i.e., sequences of adjacent vertices) from any node in A to any node in B must intersect some node in C , and in such case C is called a *separator*. If C is such a separator, then for any $p \in \mathcal{F}(G, \mathcal{M}^{\text{sep}})$ we must have that $p(x_A, x_B | x_C) = p(x_A | x_C)p(x_B | x_C)$ for all values x_A, x_B, x_C . This conditional independence property is denoted $X_A \perp\!\!\!\perp X_B | X_C$. Any $p \in \mathcal{F}(G, \mathcal{M}^{\text{sep}})$ must satisfy the conditional independence properties corresponding to all separation properties in G .

Another way to characterize MRFs is via factorization properties \mathcal{M}^{fac} [258]. We have that $p \in \mathcal{F}(G, \mathcal{M}^{\text{fac}})$ if p factorizes with respect to a set of factors, $\{\phi(\cdot)\}$, known as *potential functions*, defined on the maximal cliques¹ of G . Specifically, if $p \in \mathcal{F}(G, \mathcal{M}^{\text{fac}})$, then it must be the case that we can write p as $p(x) = \prod_{C \in \mathcal{C}} \phi_C(x_C)$ where $\phi_C(x_C)$ are functions, and for each $C \in \mathcal{C}$, there must be some clique (say \mathcal{C}) in G such that $C \subseteq \mathcal{C}$. It is always the case that $\mathcal{F}(G, \mathcal{M}^{\text{fac}}) \subseteq \mathcal{F}(G, \mathcal{M}^{\text{sep}})$. This means that if a given p factors with respect to a graph G , then it also necessarily obeys the separation property with respect to the same graph. The reverse is true ($\mathcal{F}(G, \mathcal{M}^{\text{fac}}) \supseteq \mathcal{F}(G, \mathcal{M}^{\text{sep}})$) in general only for strictly positive p [258], which is known as the Hammersley-Clifford theorem. An example of separation and factorization properties is depicted in Figure 2.2-left. Note that in this case, separation does not hold if we remove any node from the blocking set — i.e., conditional independence between X_2 and X_4 does not hold if we condition only on any strict subset of $\{X_1, X_3, X_5\}$.

In general, a lack of an edge between two nodes does not imply that the nodes are independent. The nodes might be able to influence each other indirectly via an indirect path. Moreover, the existence of an edge between two nodes does *not* imply that the two nodes are necessarily dependent for a particular instance of a probability distribution within the graph's family — the two nodes could still be independent for certain parameter values or under certain conditions. A graphical model guarantees only that the lack of an edge implies some conditional independence property, determined according to the graph's semantics. There can be distributions within a family that have more independence (or factorization) properties than what is required by a given graph. We can think of this as a GM requiring a set of properties, or rules, that must be obeyed by any family member, where by properties (rules) we mean the set of conditional independence (or as we see below, more generally factorization) properties expressed by the graph. A given probability distribution p must obey all the rules to be a member of the family. A distribution p that obeys all the rules may also obey additional rules and it is still a family member. But if p violates any of the GM's rules, it is not a member of the family.

It is therefore best, when discussing a given GM, to refer only to its (conditional) independence properties or its factorization properties rather than its dependence properties (which might not be true for a given instance). For example, it is more precise to say that there is an edge between A and B , or that there is an allowable direct interaction between A and B , than to say that A and B are dependent.

Note that in the literature when one has an edge of the form $A \rightarrow B$, it is often colloquially said that there is a “direct dependence” between A and B . $A \rightarrow B$ does not require, in a particular instance of the family of probability distributions $p(A, B)$, that A and B are dependent on each other. Rather, $A \rightarrow B$ allows for there to be such a dependence. In this document, then, we will use the term “ B may be directly dependent on A ” to suggest that such a dependence is allowed.

Why is this important? For a number of reasons. When training a model, say in GMTK, if one puts an edge $A \rightarrow B$, then in the resulting trained model might have a dependence between A and B . On the other hand, if in the true data generating source A and B are independent, it might not be the case that in the trained model A and B are independent. Due to sampling noise, dearth of training data, or other reasons

¹A clique is a fully connected set of vertices, and with a *maximal clique* (or *maxclique*), the vertex set is no longer fully connected if any additional vertex is added.

(e.g., bugs), one might get a spurious dependence between A and B even though there shouldn't be one. On the other hand, if the data is sufficiently large and well represents the underlying true distribution, and if A and B are in truth independent, then specifying the edge $A \rightarrow B$ might at best be wasteful. The resulting model, then, might be used under the assumption that A and B do directly influence each other, but actually in the model that is not the case. This might also lead to faulty hypotheses being made about the data. On the other hand, if one treats $A \rightarrow B$ as a possible or allowable dependence, hopefully one might treat each such allowance with some skepticism before further analysis is performed.

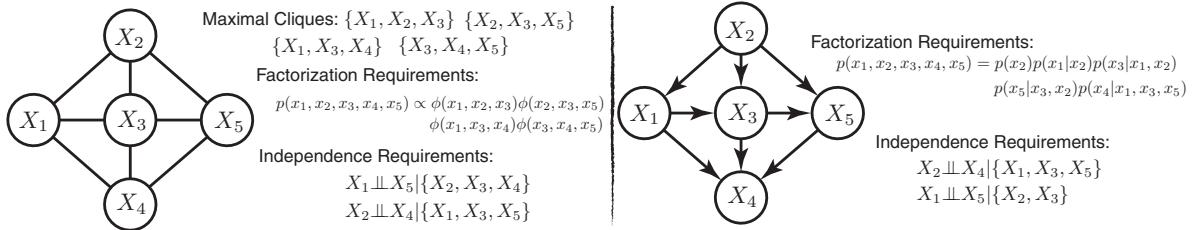


Figure 2.2: Left: A 4-cycle (X_1, X_2, X_4 , and X_5) with an extra node (X_3). There are two separation (and thus conditional independence) properties — namely, X_2 is separated from X_4 by X_1, X_3 , and X_5 , and also X_1 is separated from X_5 by X_2, X_3 , and X_4 . There are four maximal cliques (as indicated) and thus a minimum of four factors. Any p in the graph's family must obey these independence properties and (equivalently) must validly factorize as shown. We therefore say that any distribution that is a member of the family must (at least) satisfy the stated and necessary “factorization requirement.” It can have other factorization properties and still be in the family, but it can not have any less than what is required. Right: The edges are now directed as in a BN. The required factorization properties, as well as the conditional independence properties, are shown. Notice how the required properties have changed relative to the undirected (left) case.

A Bayesian network (BN) is another type of graphical model, where G is always directed and acyclic (a DAG). All edges are said to point from *parent* to *child*. Like before, a BN corresponds to a family of probability distributions $\mathcal{F}(G, \mathcal{M}^{\text{bn}})$ based on G and a set of properties \mathcal{M}^{bn} specific to BNs. BN properties are a bit more complicated than those of MRFs, but the essential concept is the same: if there is a form of separation encoded in the graph, then there must be a corresponding independence property. In the BN case, separation can be based on d-separation [336, 258].

Definition 7. A set of variables A is conditionally independent of a set B given a set C if A is **d-separated** from B by C . D-separation holds if and only if all paths that connect any node in A and any other node in B are blocked. A path is blocked if it has a node v with either: 1) the arrows along the path **do not** converge at v (i.e., serial or diverging at v) $v \in C$; or 2) the arrows along the path **do** converge at v , and neither v nor any descendant of v is in C . Note that C can be the empty set in which case d-separation encodes standard statistical independence.

From d-separation, one may compute a list of conditional independence statements made by a graph, and there might be quite a few of them (this is an example where the graph's picture is worth a thousand equations). This set of probability distributions for which this list of statements is true is precisely the set of distributions represented by the graph.

Graph properties equivalent to d-separation include the directed local Markov property [258] (a variable is conditionally independent of its non-descendants given its parents), and the Bayes-ball procedure [392] which is a simple algorithm that one can use to read conditional independence statements from graphs, and which is arguably simpler than d-separation. There is also an equivalent corresponding directed factorization property $\mathcal{M}^{\text{dfac}}$ [258], namely: if x_i is a variable, and π_i is the set of parents of i , then if $p \in \mathcal{F}(G, \mathcal{M}^{\text{dfac}})$, we may factor p as $p(x) = \prod_i p(x_i | x_{\pi_i})$. There are many additional (and provably equivalent) characterizations of BNs, including the notion of *d-separation* [336, 258].

In a Bayesian network, edges point from *parent* variable to *child* nodes, and as mentioned such graphs implicitly portray factorizations that are simplifications of the chain rule of probability, namely:

$$p(X_{1:N}) = \prod_i p(X_i|X_{1:i-1}) = \prod_i p(X_i|X_{\pi_i}). \quad (2.1)$$

The first equality is the probabilistic chain rule, and always holds regardless of the ordering of the variables. The second equality holds under a particular BN and for a particular order, where π_i designates node i 's parents according to the BN.

An example of a BN is depicted in Figure 2.2-right.

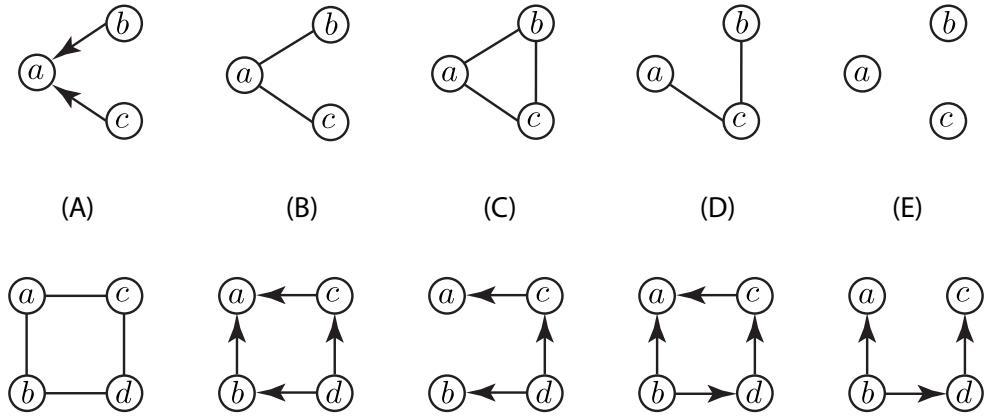


Figure 2.3: Example of a BN whose family cannot be represented by MRFs, and vice versa. On top left, we see a BN — none of the MRFs over three nodes on the right (positions B, C, D, and E) are identical to this BN. This construct is called a *V-structure*. On the bottom left, we see a MRF (a four cycle) that cannot be represented by a BN, and various failed attempts are made on the right (positions B, C, D, and E).

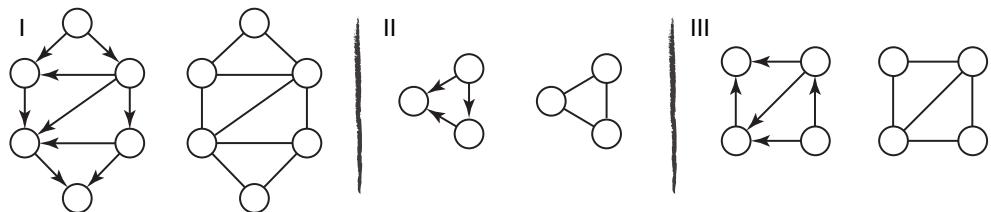


Figure 2.4: Examples of BNs and associated MRFs that correspond to precisely the same family. In each case (I, II, and III), on the left is a BN and the right is a MRF that specifies exactly the same family.

Note that self-loops do not make sense for a BN. Firstly, a BN is defined using a DAG, so there are no self-loops. A self-loop would, if it was allowed, correspond to a factor of the form $p(x_i|x_{\pi_i}, x_i)$ which does not correspond to any factor in the chain rule expansion $p(x_{1:n}) = \prod_i p(x_i|x_{1:i-1})$ for any ordering of the variables. That is, there is no ordering $\sigma = (\sigma_1, \sigma_2, \dots, \sigma_n)$ such that $p(x_{1:n}) = \prod_i p(x_{\sigma_i}|x_{\sigma_1}, \dots, x_{\sigma_{i-1}})$ and where $\sigma_i \in \{\sigma_1, \dots, \sigma_{i-1}\}$.

Bayesian networks and MRFs correspond to different families. That is, there are certain MRFs that can not be *precisely represented* by Bayesian networks, and vice versa. Note that being precisely represented is quite different than being covered. For example, a clique is an MRF that represents all distributions, so if G is a clique then $p \in \mathcal{F}(G, \mathcal{M})$. Similarly, a directed acyclic clique G' (a graph where all nodes are connected to each other but where there are no cycles) is a Bayesian network, and $p \in \mathcal{F}(G', \mathcal{M}^{\text{dfac}})$

for any p . Hence, both MRFs and Bayesian networks can, with the right graph, “cover” or “include” all distributions, given enough edges in the corresponding graph. In other words, the clique graphs in each case have no restrictions and hence do not restrict any distribution from being a member of the family. This is not interesting, though, since there are many peculiar (and useful) properties of certain p that cannot be expressed by a certain graphical models. That is, for such p , the properties might not exclude it from the clique graph family for either MRFs or Bayesian networks, but then there might be no graph which allow membership to be exclusive to those p with those peculiarities. We will call this the *specificity* of a graphical model.

To make this more concrete, Figure 4.9 shows several examples of BNs (respectively MRFs) that correspond to model families that cannot be represented by MRFs (respectively BNs). For example, the V-structure is of course covered by the clique MRF, but there is no MRF that asks precisely for those peculiarities of the V-structure, namely that it is required for $b \perp\!\!\!\perp c$ but not required for $b \perp\!\!\!\perp c|a$. Likewise, there is no Bayesian network that requires the 4-cycle, i.e., **only** that $c \perp\!\!\!\perp b|a, d$ and $a \perp\!\!\!\perp d|c, b$ but does not require any further independence properties. There are distributions that satisfy either V-structure or the 4-cycle of course, since the corresponding families, respectively for both, are not empty.

The families corresponding to BNs and MRFs have an non-empty intersection, however. This means that the properties that characterize p can be precisely characterized by either a Bayesian network or an MRF, and there is in general no benefit to using one over the other (other than one might be easier for a particular application). Figure 2.4 shows a few examples of a BN and MRF that correspond to the same family of models and these corresponds to what are known as the decomposable models [258].

A benefit of graphical models is that it is possible to automatically develop algorithms that can compute probabilistic quantities of interest (exactly or approximately) based on knowledge only of (G, \mathcal{M}) and without having knowledge of a particular instance $p \in \mathcal{F}(G, \mathcal{M})$. Given such an algorithm, it is then applicable to any member of the family, thereby amortizing the algorithm’s development cost over all $p \in \mathcal{F}(G, \mathcal{M})$. The algorithm can of course also be applied to any $p \in \mathcal{F}(G', \mathcal{M}')$ where $\mathcal{F}(G', \mathcal{M}') \subseteq \mathcal{F}(G, \mathcal{M})$.

One simple such algorithm for computing probabilistic quantities exactly is belief propagation and its generalization [336, 208] the junction tree algorithm. We give a brief overview of this algorithm here. First, the algorithm operates on MRFs rather than directly on BNs. Therefore, for any BN, it must be possible to find a MRF that includes all members of its family. That is, given that G is a directed graph, we need an operation (let’s call it $m(G)$) that transforms the BN G into an MRF $m(G)$ such that $\mathcal{F}(G, \mathcal{M}^{\text{bn}}) \subseteq \mathcal{F}(m(G), \mathcal{M}^{\text{sep}})$ holds. Then, any inference algorithm developed for $\mathcal{F}(m(G), \mathcal{M}^{\text{sep}})$ will be valid for any member of $\mathcal{F}(G, \mathcal{M}^{\text{bn}})$. We also wish $m(G)$ to be minimal, in that performing general inference for a member of $\mathcal{F}(m(G), \mathcal{M}^{\text{sep}})$ should not be significantly (or any) more costly than inference for members of $\mathcal{F}(G, \mathcal{M}^{\text{bn}})$.

The operation $m(G)$ (called moralization) is a simple graph-theoretic operation of converting a DAG into an undirected graph and proceeds as follows:²

Definition 8 (moralization). *Connect with an undirected edge the unconnected parents of any child, and then drop all remaining edge directions. The reason it is called moralization is that any unconnected (unmarried) parents of a child node are connected with undirected edges (i.e., married, or made moral).*

The moralization operation ensures that any factor in the original BN has a containing clique (a fully connected set of nodes) in the resulting MRF, and thus is capable of respecting the original factorization property in the BN. The operation of moralization is shown graphically in Figure 4.3.

Henceforth, we assume the graph is an MRF.

²As nodes do not specify a gender, however, the degree to which such a graph is considered “moral” might still depend on one’s geographic domicile or political persuasion. Moreover, before moralization a node might have more than two parents. Hence, it might be said that Bayesian networks are quite progressive.

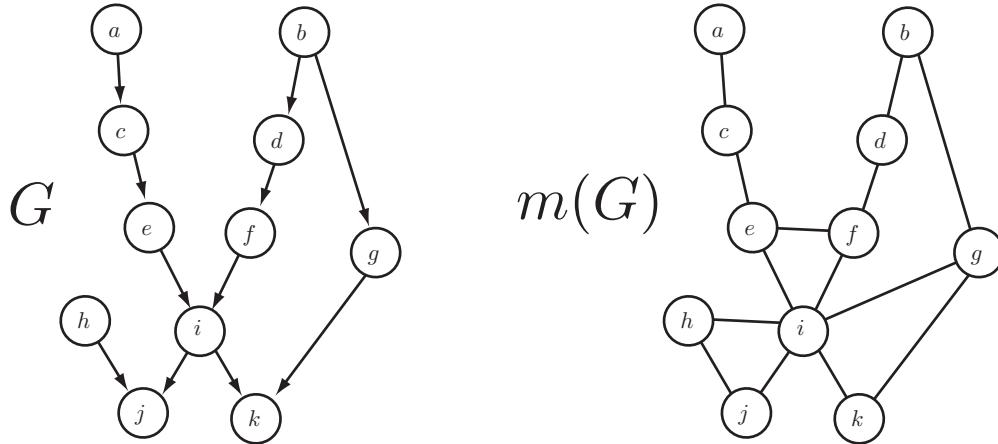


Figure 2.5: Examples of a DAG G and its moralized version $m(G)$. Note that any factor $p(v|\text{pa}(v))$ becomes a clique in the moralized graph so that if p is such that directed factorization property holds in the Bayesian network, then p is also such that the global Markov property holds in $m(G)$. For example, $p(j|h, i)$ becomes a clique factor $\phi_{j,h,i}$ after moralization, so the factor $p(j|h, i)$ lives within some clique in the resulting undirected graph. Also, note that the V-structure $h \rightarrow j \leftarrow i$ which expresses that $h \perp\!\!\!\perp i$ is lost in the undirected graph — this is not surprising since V-structures are precisely the construct that MRFs are unable to represent.

If the graph G is a tree (meaning, all pairs of nodes are connected by exactly one unique path) or a forest (meaning, all pairs of nodes are connected by at most one unique path), then we are guaranteed that there are at least two random variables (i.e., leaf nodes) that have the property that they are involved in a factor with only one additional node. Such random variables can then be marginalized away (the corresponding nodes have been *eliminated*, and the graph-theoretic counterpart of belief propagation is known as the *elimination algorithm*). This leaves us with a residual graph that is still a tree, meaning that once again there must be at least two nodes with the desired leaf property. Performing such marginalizations repeatedly, but only on leaf nodes, ensures that computing any node marginal, or any marginal along two neighboring nodes connected by a tree edge, is computationally inexpensive. Each such marginalization can be seen as a message passed between two nodes along a connecting edge. The message passing equations are summarized in Figure 2.6 I. Thus we can see that marginalizing away all nodes corresponds to sending messages starting at the leaves, and towards a designated root of the tree. In fact, these marginalization messages can be passed along both directions of each edge, and since there are $N - 1$ edges, after $2(N - 1)$ messages the state of the tree will reach the point where each node (and edge) holds the marginal distribution for that node (edge). This is sufficient for the majority of probabilistic queries needed on a tree. For example, most learning algorithms do not need marginals over sets of variables larger than those that may directly interact (e.g., via a parameterized factor) in the graph.

If the graph $G = (V, E)$ is not a tree, then we can make use of a generalization of belief propagation known as the *junction tree algorithm*. A junction tree is a particular kind of tree in which the nodes are overlapping clusters of the original nodes V . Let $\mathcal{T} = (\mathcal{C}, \mathcal{E})$ be this tree, where for each $C \in \mathcal{C}$, $C \subseteq V$. The edge set \mathcal{E} is such that \mathcal{T} is a tree with subsets of V as nodes. The tree of clusters can then be treated exactly like the tree of nodes discussed in our presentation of belief propagation, where we eliminate leaf nodes by marginalizing them away, thereby producing messages that are passed on the junction tree. In this case, however, marginalizing away a leaf node means summing over more than just one variable. In fact, multiple variables are summed over simultaneously. Essentially, the cluster of nodes C can be treated as a single node with a large enough domain size. That is, the variable, say Y_u , can represent the set of variables

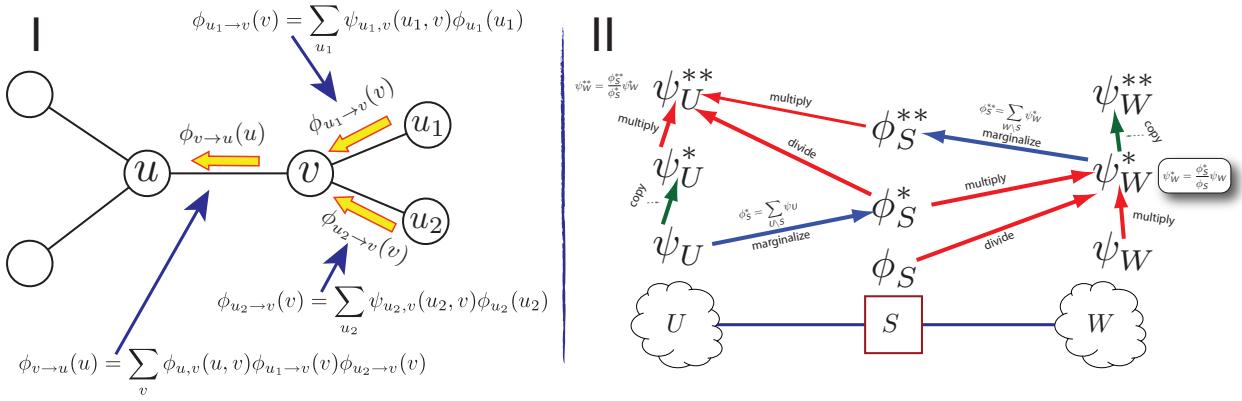


Figure 2.6: I: standard belief propagation messages on a tree. Each yellow arrow corresponds to a message, which is defined by the associated equation. II: Messages sent on a clustered graph, where each cloud shape is a cluster of variables in the original graph, and the messages are in the Hugin [208] style. The clique potentials ψ_U and ψ_W are initialized to the product of factors that are contained within cliques U and W respectively. The separator potential ϕ_S is initialized to unity. The messages are shown in data-flow fashion, so that the arrows show message dependency. Green arrows are message copies. Blue arrows are marginalization operations, and red arrows are message multiplies or divides. The cost of this process is exponential in the size of the cluster. More details may be found in [208, 249, 218].

X_C if Y_u has as many possible values as the Cartesian product $\times_{c \in C} |D_{X_c}|$ where D_{X_v} is the domain of X_v and $|D_{X_v}|$ its size.

A tree of cluster nodes is not a junction tree unless it possesses certain properties. Those properties are as follows: first, the original graph must be *triangulated* (see the next ¶ and Figure 2.7). If the graph is not triangulated, then one must add edges until it is triangulated. Only the class of triangulated graphs can be clustered into a junction tree. Second, the tree of clusters must be such that each tree node cluster corresponds to a clique in the (triangulated) graph. Third, for every two clusters in the tree, and for the necessarily unique path between these two clusters, the nodes that lie in the intersection of these extremal cliques must exist within every cluster along this path. This is called the *running intersection property*. If the above three properties are true, then eliminating leaf clusters one at a time will produce a correct inference algorithm for the original graph. We note, however, that the cost of marginalizing away the nodes in a cluster is exponential in the cluster's size, so it is imperative in general to have as small a cluster size as possible. When the distribution is very sparse (meaning many random variable assignments have zero probability), however, then in some cases larger cliques can actually be faster [19], and this is indeed something that is important for GMTK.

A graph is triangulated whenever any cycle in the graph has an edge (called a *chord*) connecting two non-consecutive nodes in that cycle. In other words, choose any cycle in the graph. Once that cycle is chosen, there are original graph edges that are part of the cycle, and there might be original graph edges that are not be part of the cycle. In order for a graph to be triangulated, it must be that for any chosen cycle, there has to be at least two non-adjacent nodes along that cycle (non-adjacent meaning that they are not connected by any edge in the currently chosen cycle) that are connected by a non-cycle edge (chord). If there is any such chosen cycle where no two non-adjacent nodes along that cycle are connected, the graph is not triangulated.

Running the elimination algorithm (using any node order) on the original graph is one way to produce a triangulated graph. In this case, however, the elimination algorithm must be modified: to eliminate a node v with more than one neighbor, we first connect together all of v 's neighbors in the current graph and then we remove v and its immediately adjacent edges from the graph. This step is called *node elimination*,

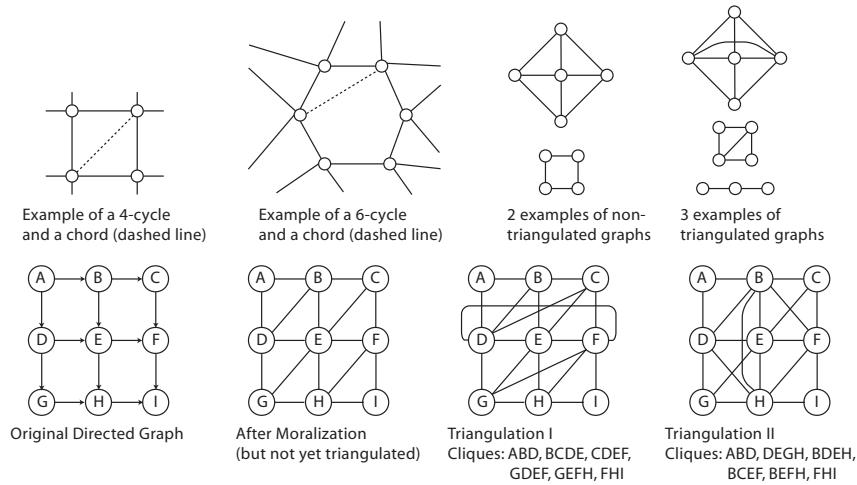


Figure 2.7: A brief summary of several graphical model concepts used in this document. A *chord* of a cycle is any two non-consecutive nodes in that cycle that are connected. A *triangulated* graph is one that has no chordless cycles. A *clique* in a graph is a set of nodes that are jointly pairwise connected, and where no proper superset of this set has that property. The process of *moralization* of a directed graph connects any unconnected parents of a node and then drops all edge directions (a procedure necessary for correct probabilistic inference). The bottom two right-most figures show two possible triangulations (and the corresponding clique set) for the graph on the bottom left.

and any new edges are called *fill-in edges* (see Figure 2.8). Given a graph $G = (V, E)$ we can depict its triangulation as $G^{\text{tr}} = (V, E \cup F)$ where F are the fill-in edges. To triangulate the graph, one way is to eliminate all of the nodes, and then “reconstitute” all nodes along with any edges that were added along the way. Once done, we are guaranteed that the graph is triangulated. Since a triangulation can only add edges, $\mathcal{F}(G, \mathcal{M}^{\text{sep}}) \subseteq \mathcal{F}(G^{\text{tr}}, \mathcal{M}^{\text{sep}})$, so any inference algorithm for all members of $\mathcal{F}(G^{\text{tr}}, \mathcal{M}^{\text{sep}})$ will work for any member of $\mathcal{F}(G, \mathcal{M}^{\text{sep}})$. Any of the $N!$ orders in which nodes might be eliminated (each known as an *elimination order*) will yield a triangulated graph. The graphical elimination procedure corresponds to summing out a variable from a factored probability distribution.

One very simple triangulated graph has all variables clustered into a single large clique, but this is typically not useful (or interesting, as mentioned above) since it entails a cost that is exponential in the

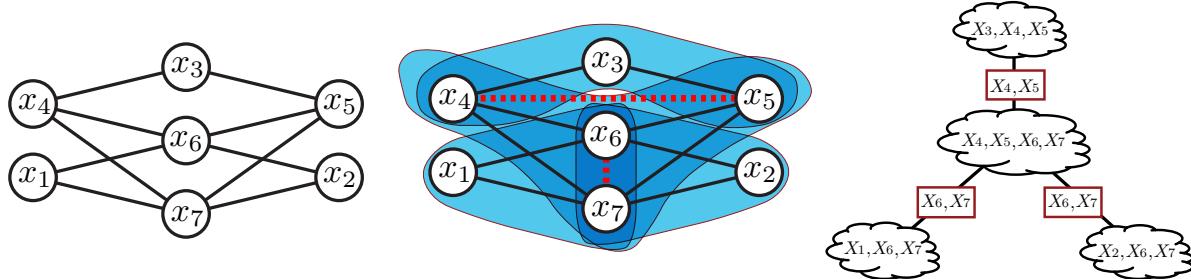


Figure 2.8: A graph with seven nodes is eliminated in node order (x_1, x_2, \dots) to produce the middle triangulated graph. The fill-in edges are shown as dotted red, and the cliques (node clusters) are shown as shaded blue. Right: the resulting junction tree. In the junction tree, the “nodes” (which are clusters of original graph nodes) are shown as cloud shapes, and the intersection between two adjacent clouds are shown as a square shape. Notice that the running intersection property holds for any two cliques.

number of variables N . In general, the goal is to find the triangulation that minimizes the size of the largest maximal clique. This problem is identical to finding the inherent treewidth of the original graph (a quantity that results with the inherent best-case complexity of the graph). In general, if each random variable can have r values, then the cost of doing inference will be $O(r^{k+1})$ where $k+1$ is the size of the largest cluster in the graph. Note that k is in the exponent so even reducing k by one can have an enormous different in inference time. Unfortunately, finding the clustering that results in the lowest k is itself an NP-complete optimization problem, so heuristics are often used [258, 208, 249, 218].

In general, the clusters in a triangulated graph consist of all of the maximal cliques and possibly other cliques that are subsets of those maximal cliques. Given a junction tree (representing a triangulated graph), one can form another junction tree (representing the same triangulated graph) by taking any clique C , forming a subset of that clique (say $C' \subseteq C$) and attaching C' in the tree with an edge directly between C' and C . In such case, the running intersection property will still hold (since C contains everything that is contained in C'). We'll call this the *pendant clique property*. While this might process might seem to be redundant or unnecessary, we will see that in some cases in practice, doing this operation is useful since it allows one to specify a given marginal probability distribution over the variables C' that one is interested in computing for an application during the message passing procedure. In fact, GMTK can make use of the pendant clique property in order to efficiently compute desired marginals (see §TODO: cross reference section)

In the context of sparse models, it is important to note as mentioned above, that elimination orders do not produce all possible triangulations of a graph. An easy way to see this is that for most graphs, there is no elimination order that will produce one large clique (i.e., **all** nodes connected to each other), even though such a large clique is a valid triangulated graph (as there are no chordless cycles). For most graphs, there are many other examples of such “large” cliques that cannot be produced by any elimination order. Normally one does not wish to have large cliques, as mentioned above. When the model is sparse (meaning that there are cases where variables might be deterministic functions of others) it can be quite beneficial to produce junction trees with large cliques. This is described more fully in [19]. GMTK's triangulation engine, (a program called `gmtkTriangulate`), supports many heuristics for producing such large cliques, or are they are more formally called, *non-minimal triangulations*.

The general process for using a graphical model is depicted in Figure 6.1. Here, a user designs a graph G according to some Markov properties \mathcal{M} . The graph is fed to a module that produces an inference procedure which then gets used many times, once for each probabilistic query

In some cases, once the graph is clustered and even if it is clustered optimally (meeting the treewidth), performing inference is still prohibitively costly (the treewidth is just too high). There are enumerable techniques to perform approximate inference in such graphs, see for example [58, 434, 249]. Also, in some special cases (for example grid graphs used in image processing and image segmentation where the potentials are submodular) simple min-cut algorithms can often find optimal (or near optimal) solutions, even if the tree width is high, as long as certain restrictions are placed on the factors.

2.2.2 Gaussian Graphical Models

The most widely used density for acoustic modeling in speech recognition systems has been the multi-dimensional Gaussian. Gaussians are also quite useful in many other applications as well. Hence, spend a bit of time describing Gaussians as graphical models (both undirected and directed), as this will be quite useful later when describing how GMTK works with Gaussians.

The Gaussian density has a deceptively simple mathematical description that does not disclose many of the useful properties this density possesses (such as that first and second moments completely characterize the distribution). Often, many important properties can be described by graphical models, meaning that a multivariate Gaussian density p lives within the family $\mathcal{F}(G, \mathcal{M})$ of a given graphical model.

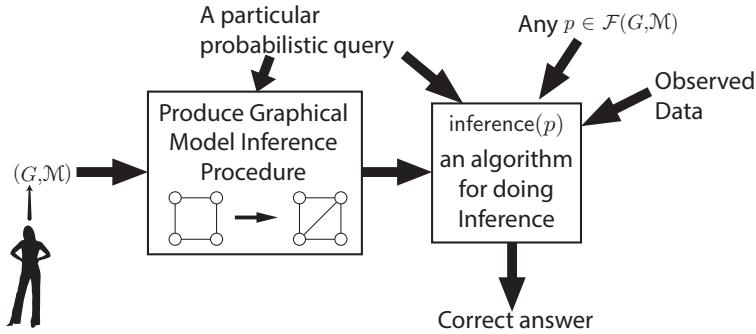


Figure 2.9: Graphical model *inference* means that we wish to take a graph G and a set of Markov properties \mathcal{M} defining the graphical model type, and produce an algorithm *inference* so that for any $p \in \mathcal{F}(G, \mathcal{M})$, $\text{inference}(p)$ will produce the desired correct quantity. An essential component for this is that we may wish to run $\text{inference}(p)$ for many different $p \in \mathcal{F}(G, \mathcal{M})$ so that the cost of producing $\text{inference}(p)$ starting from (G, \mathcal{M}) is amortized over the many times $\text{inference}(p)$ is run. With GMTK, a user first designs a model that includes specifying a graph and a set of parameters (which might be learnt from another GMTK module). The module (on the left) that produces the graphical model inference procedure (and whose cost is amortized over the different $p \in \mathcal{F}(G, \mathcal{M})$ for which a probabilistic query is needed) does several things. This includes triangulating the graph (pictured here in the case of chording the 4-cycle), deciding on message passing orders and/or variable summation orders within cliques, assigning factors and constraints to cliques, and a number of other bookkeeping items, all of which can have a real practical impact on inference time and memory cost.

An N -dimensional Gaussian density has the form:

$$p(x) = p(x_{1:N}) = \mathcal{N}(x_{1:N}; \mu, \Sigma) = |2\pi\Sigma|^{-1/2} e^{-\frac{1}{2}(x-\mu)^T \Sigma^{-1} (x-\mu)} \quad (2.2)$$

where μ is an N -dimensional mean vector, and Σ is an $N \times N$ covariance matrix. The Gaussian is defined by its first μ and second Σ moments, that is the first moment $EX = \mu$, and second moment $\text{cov}(X) = E[(x - \mu)(x - \mu)^T]$. A Gaussian is unusual in this regard in that it can be completely parameterized via only its first and second moments (a Gaussian also has higher order moments but once we have the first and second moments, the higher order moments provide no additional information about the distribution). Hence, that these two moments constitute sufficient statistics for the Gaussian.

A more formal definition of a Gaussian is as follows: X has a d -variate joint normal (Gaussian) distribution iff $a^T x$ is a univariate normal \forall fixed d -vectors a , where a univariate normal distribution is defined as $p(x|\mu, \sigma^2) = \frac{1}{\sqrt{2\pi}\sigma^2} e^{-\frac{1}{2}(x-\mu)^2/\sigma^2}$.

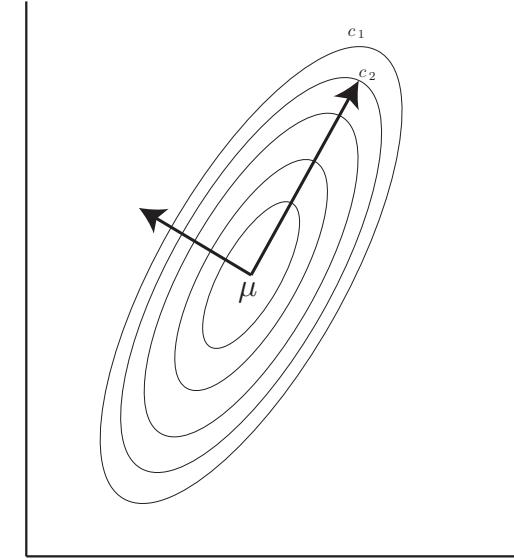
Note that the argument $f(x) = \frac{1}{2}(x - \mu)^T \Sigma^{-1} (x - \mu)$ is the *Mahalanobis distance* between x and μ , where anything along the contours is the same distance from the center (and contours are ellipses). This is shown in Figure 2.10.

It is often useful to represent (or specify) a Gaussian using the inverse of the covariance matrix. Typically, $K = \Sigma^{-1}$ refers to the inverse covariance matrix of the density and is often called the *concentration matrix* or the *precision matrix*, the reason being that the larger the diagonal of the concentration matrix, the more “concentrated” the distribution is.

There are many useful properties of Gaussians, including:

1. If $x \sim \mathcal{N}(\mu, \Sigma)$ and $y = Ax + c$ then $y \sim \mathcal{N}(A\mu + c, A\Sigma A^T)$.
2. If $x \sim \mathcal{N}(\mu, \Sigma)$, and Σ is positive definite and $y = \Sigma^{1/2}(x - \mu)$, where $\Sigma = \Sigma^{1/2}\Sigma^{1/2}$, then $y \sim \mathcal{N}(0, I)$.

Figure 2.10: On the right, we see elliptical contours of a 2-D Gaussian with mean μ . The arrows point in the major directions (if we sort eigenvalues from largest to smallest, then the arrows point in directions of corresponding eigenvectors). The set of points $\{x : f(x) = c\}$ for constant c define the ellipsoids at Mahalanobis distance c away from μ . If the covariance matrix was the identity matrix, the contours would be circular and if it was diagonal, the contours would be axis-parallel ellipses.



3. If $x_1 \sim \mathcal{N}(\mu_1, \Sigma_1)$ and $x_2 \sim \mathcal{N}(\mu_2, \Sigma_2)$, and $X_1 \perp\!\!\!\perp X_2$, then $x_1 + x_2 \sim \mathcal{N}(\mu_1 + \mu_2, \Sigma_1 + \Sigma_2)$. If the variables are not independent, in general we need to compute the convolution of distributions to get distribution of the sum, but it will still be Gaussian.
4. If x_1 and x_2 are each Gaussian with zero mean, and $E[X_1 X_2] = E[X_1]E[X_2]$, then $X_1 \perp\!\!\!\perp X_2$.

In general, we are interested here in how Gaussians relate to Graphical models and their independence statements. What properties of multivariate Gaussian, say, will lead to certain variables being (conditionally) independent of others? GMTK, for the most part, treats Gaussians using their Bayesian network description. This makes it easy to use the same mechanism to express both standard Gaussians (including diagonal covariance Gaussians and full covariance Gaussians and sparse variants that lie in between), and conditional Gaussians (see §2.2.2.3) and their sparse variants. It is thus important to understand all aspects of Gaussians, including their undirected graphical model perspective, which we cover next. Lets start with marginal independence.

2.2.2.1 Gaussians and Independence

It will be useful to form partitions of an $n \times 1$ vector x into a number of parts. For example, a bi-partition of x ,

$$x = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \quad (2.3)$$

may be formed, where x_1 is a $p \times 1$ vector and x_2 is a $q \times 1$ vector are sub-vectors of x [174], and where the sum of the dimensions of x_1 and x_2 equals n ($p + q = n$). Similarly, the mean vector μ can be partitioned as

$$\mu = \begin{bmatrix} \mu_1 \\ \mu_2 \end{bmatrix} \quad (2.4)$$

and the covariance and concentration matrices can be block partitioned as

$$\Sigma = \begin{bmatrix} \Sigma_{11} & \Sigma_{12} \\ \Sigma_{21} & \Sigma_{22} \end{bmatrix} \quad \text{and} \quad K = \begin{pmatrix} K_{11} & K_{12} \\ K_{21} & K_{22} \end{pmatrix}. \quad (2.5)$$

where, Σ_{11} is $p \times p$, Σ_{21} is $p \times q$, Σ_{22} is $q \times q$, and so on.

We can then write the Gaussian as:

$$p(x|\mu, \Sigma) = \frac{1}{(2\pi)^{(p+q)/2} |\Sigma|^{1/2}} \exp \left[-\frac{1}{2} \begin{pmatrix} x_1 - \mu_1 \\ x_2 - \mu_2 \end{pmatrix}^T \begin{pmatrix} \Sigma_{11} & \Sigma_{12} \\ \Sigma_{21} & \Sigma_{22} \end{pmatrix}^{-1} \begin{pmatrix} x_1 - \mu_1 \\ x_2 - \mu_2 \end{pmatrix} \right] \quad (2.6)$$

By convention, we have that $\Sigma_{11}^{-1} = (\Sigma_{11})^{-1}$, so that the sub-matrix operator takes precedence over the matrix inversion operator.

A well known property of Gaussians is that if $\Sigma_{12} = 0$ then x_1 and x_2 are marginally independent ($x_1 \perp\!\!\!\perp x_2$).

Theorem 9. Let X be partitioned as above. Then $X_1 \perp\!\!\!\perp X_2$ iff $\Sigma_{12} = 0 = \Sigma_{21}^\top$.

Proof. We first factorize the matrix as follows:

$$|\Sigma| = \begin{vmatrix} \Sigma_{11} & 0 \\ 0 & \Sigma_{22} \end{vmatrix} = |\Sigma_{11}| |\Sigma_{22}| \quad (2.7)$$

Also, we have that:

$$\begin{pmatrix} \Sigma_{11} & 0 \\ 0 & \Sigma_{22} \end{pmatrix}^{-1} = \begin{pmatrix} \Sigma_{11}^{-1} & 0 \\ 0 & \Sigma_{22}^{-1} \end{pmatrix} \quad (2.8)$$

This allows us to express the distribution as follows:

$$p(x|\mu, \Sigma) = \frac{1}{(2\pi)^{(p+q)/2} |\Sigma|^{1/2}} \exp \left[-\frac{1}{2} \begin{pmatrix} x_1 - \mu_1 \\ x_2 - \mu_2 \end{pmatrix}^T \begin{pmatrix} \Sigma_{11} & \Sigma_{12} \\ \Sigma_{21} & \Sigma_{22} \end{pmatrix}^{-1} \begin{pmatrix} x_1 - \mu_1 \\ x_2 - \mu_2 \end{pmatrix} \right] \quad (2.9)$$

$$= \frac{1}{(2\pi)^{p/2} |\Sigma_{11}|^{1/2} (2\pi)^{q/2} |\Sigma_{22}|^{1/2}} \exp \left[-\frac{1}{2} ((x_1 - \mu_1)^\top \Sigma_{11}^{-1} (x_1 - \mu_1) + (x_2 - \mu_2)^\top \Sigma_{22}^{-1} (x_2 - \mu_2)) \right] \quad (2.10)$$

$$= p(x_1|\mu_1, \Sigma_{11}) p(x_2|\mu_2, \Sigma_{22}) \quad (2.11)$$

which means that $x_1 \perp\!\!\!\perp x_2$ since factorization holds. \square

But this is just marginal independence. We would like conditional independence, meaning if we partition $x = [x_1^\top \ x_2^\top \ x_3^\top]^\top$, when is it the case that $X_1 \perp\!\!\!\perp X_2 | X_3$ under X being jointly Gaussian? We will relate this to K , the concentration matrix.

The property of a Gaussian we wish to show is that for a given tri-partition $x = [x_1 \ x_2 \ x_3]$ of x , and corresponding tri-partitions of μ and K , then $X_1 \perp\!\!\!\perp X_2 | X_3$ if and only if, in the corresponding tri-partition of K , we have that $K_{12} = 0$. Now this might seem a bit unusual (or at best unimportant), but bear with us for the moment, as soon you will have your reward by seeing that this corresponds to a very simple property of a corresponding undirected graph.

2.2.2.2 Partitioned Matrices and the Schur Complement

We take a brief tour of the Schur complement, which will allow us to express the inverse of a matrix semi-analytically via an expression involving inverses of the blocks of the original matrix.

First, consider a general matrix M partitioned into four blocks as follows:

$$M = \begin{bmatrix} E & F \\ G & H \end{bmatrix} \quad (2.12)$$

We assume that E^{-1} , H^{-1} , and M^{-1} all exist.

We need to find a simple formula to invert M in terms of these blocks. We “block diagonalize” the matrix (i.e., make all off-diagonal blocks zero) using the following two steps:

1. premultiply 2nd block column by row vector $[I \ - FH^{-1}]$
2. postmultiply 1st block row by column vector $[I \ - H^{-1}G]^\top$.

The preceding steps can be expressed via matrix operations as in the following:

$$\begin{bmatrix} I & -FH^{-1} \\ 0 & I \end{bmatrix} \begin{bmatrix} E & F \\ G & H \end{bmatrix} \begin{bmatrix} I & 0 \\ -H^{-1}G & I \end{bmatrix} \quad (2.13)$$

$$= \begin{bmatrix} E - FH^{-1}G & 0 \\ G & H \end{bmatrix} \begin{bmatrix} I & 0 \\ -H^{-1}G & I \end{bmatrix} \quad (2.14)$$

$$= \begin{bmatrix} E - FH^{-1}G & 0 \\ 0 & H \end{bmatrix} \quad (2.15)$$

From this, we get the Schur complement of M w.r.t. H .

$$M/H \triangleq E - FH^{-1}G \quad (2.16)$$

The notation M/H for the Schur complement (where the notation is “sort of” like dividing M by H) is used for a reason, namely that from Equations (2.13) and (2.15), it can be seen that $|M| = |M/H||H|$.

Now, because of Equation (2.13), we have that $(M/H)^{-1}$ exists. This follows since $|M| > 0$ and $|H| > 0$, and each of the first and third matrix factors in Equation (2.13) are invertible, so therefore $|M/H| > 0$.

Also note, if $X M Z = W$ for matrices X , M , Z , and W , then $Z^{-1} M^{-1} X^{-1} = W^{-1}$, which gives $M^{-1} = Z W^{-1} X$. In other words, we have

$$\begin{bmatrix} E & F \\ G & H \end{bmatrix}^{-1} = \begin{bmatrix} I & 0 \\ -H^{-1}G & I \end{bmatrix} \begin{bmatrix} (M/H)^{-1} & 0 \\ 0 & H^{-1} \end{bmatrix} \begin{bmatrix} I & -FH^{-1} \\ 0 & I \end{bmatrix} \quad (2.17)$$

$$= \begin{bmatrix} (M/H)^{-1} & -(M/H)^{-1}FH^{-1} \\ -H^{-1}G(M/H)^{-1} & H^{-1} + H^{-1}G(M/H)^{-1}FH^{-1} \end{bmatrix} \quad (2.18)$$

Equation (2.18) then gives our expression for the inverse of M . This might look complicated, but we will soon see that this will greatly simplify the description of conditional independence in Gaussian models.

2.2.2.3 Gaussians and Conditional Independence

We are now set to prove conditional independence for Gaussians. Recall, we have a tri-partition of x into x_1 , x_2 , and x_3 . We block partition the covariance matrix again as:

$$\Sigma = \begin{bmatrix} \Sigma_{11} & \Sigma_{12} \\ \Sigma_{21} & \Sigma_{22} \end{bmatrix} \quad (2.19)$$

The Schur complement formula above (Equation (2.17)) can be applied giving

$$\Sigma^{-1} = \begin{bmatrix} I & 0 \\ -\Sigma_{22}^{-1}\Sigma_{21} & I \end{bmatrix} \begin{bmatrix} (\Sigma/\Sigma_{22})^{-1} & 0 \\ 0 & \Sigma_{22}^{-1} \end{bmatrix} \begin{bmatrix} I & -\Sigma_{12}\Sigma_{22}^{-1} \\ 0 & I \end{bmatrix}$$

We next replace Σ^{-1} in the argument of the \exp function in the Gaussian $e^{-\frac{1}{2}(x-\mu)\Sigma^{-1}(x-\mu)}$ with this expression:

$$\exp \left\{ -\frac{1}{2} \begin{pmatrix} x_1 - \mu_1 \\ x_2 - \mu_2 \end{pmatrix}^\top \begin{bmatrix} I & 0 \\ -\Sigma_{22}^{-1}\Sigma_{21} & I \end{bmatrix} \begin{bmatrix} (\Sigma/\Sigma_{22})^{-1} & 0 \\ 0 & \Sigma_{22}^{-1} \end{bmatrix} \begin{bmatrix} I & -\Sigma_{12}\Sigma_{22}^{-1} \\ 0 & I \end{bmatrix} \begin{pmatrix} x_1 - \mu_1 \\ x_2 - \mu_2 \end{pmatrix} \right\}$$

$$= \exp \left\{ -\frac{1}{2} (x_1 - \mu_1 - \Sigma_{12}\Sigma_{22}^{-1}(x_2 - \mu_2))^T (\Sigma/\Sigma_{22})^{-1} (\text{same}) \right\} \exp \left\{ -\frac{1}{2} (x_2 - \mu_2)^T \Sigma_{22}^{-1} (x_2 - \mu_2) \right\}$$

where “same” is just shorthand for $(x_1 - \mu_1 - \Sigma_{12}\Sigma_{22}^{-1}(x_2 - \mu_2))$. Also, from the above, we have that $|\Sigma| = |\Sigma/\Sigma_{22}| |\Sigma_{22}|$.

Thus, we have factored the normal distribution into a product of marginal and conditional distributions $p(x) = p(x_1, x_2) = p(x_1|x_2)p(x_2)$. That is, the marginal Gaussian is over X_2 :

$$p(x_2) = \frac{1}{(2\pi)^{p/2} |\Sigma_{22}|^{1/2}} \exp \left\{ -\frac{1}{2} (x_2 - \mu_2)^T \Sigma_{22}^{-1} (x_2 - \mu_2) \right\} \quad (2.20)$$

And the conditional Gaussian is over X_1 given that $X_2 = x_2$:

$$p(x_1|x_2) = \frac{1}{(2\pi)^{q/2} |\Sigma/\Sigma_{22}|^{1/2}} \exp \left\{ -\frac{1}{2} (x_1 - \mu_1 - \Sigma_{12}\Sigma_{22}^{-1}(x_2 - \mu_2))^T (\Sigma/\Sigma_{22})^{-1} (\text{same}) \right\} \quad (2.21)$$

Now, the marginal over X_2 is just a standard Gaussian with mean μ_2 and covariance matrix Σ_{22} . Moreover, μ_2 is just one block of the previously partitioned version of μ in Equation (2.4) and Σ_{22} is the corresponding block of the partitioned original covariance matrix Σ in Equation (2.5). Lets give it some new names, using m for marginal:

$$\mu_2^m = \mu_2 \quad (2.22)$$

$$\Sigma_2^m = \Sigma_{22} \quad (2.23)$$

The conditional Gaussian is more interesting in that it will give us the conditions for conditional independence. The form of the conditional Gaussian is that the mean vector is a function of x_2 — that is, we center x_2 (by subtracting off its mean), then scale the result by $\Sigma_{12}\Sigma_{22}^{-1}$. Intuitively, the first scale to be applied is Σ_{22}^{-1} which in some sense transforms the distance $(x_2 - \mu_2)$ into “normalized” units, and then the second scale Σ_{12} in some sense transforms units of variations in dimension set 2 into units of variation into dimension set 1. We then add the result to μ_1 to get the new mean $\mu_{1|2}^c$ defined below in Equation (2.24). Note also that the covariance matrix is fixed, but it is a function of a partition of original covariance matrix. This means that the covariance stays the same regardless of the value of x_2 . We can give the conditional mean and conditional covariance names as in:

$$\mu_{1|2}^c = \mu_1 + \Sigma_{12}\Sigma_{22}^{-1}(x_2 - \mu_2) \quad (2.24)$$

$$\Sigma_{1|2}^c = \Sigma_{11} - \Sigma_{12}\Sigma_{22}^{-1}\Sigma_{21} \quad (2.25)$$

We now, at least, have the tools to show that conditional independence holds when there are zeros in the concentration matrix. The concentration matrix again is as follows where we also have block partitioned it using the same dimensions as in Equation (2.5).

$$K = \Sigma^{-1} = \begin{bmatrix} K_{11} & K_{12} \\ K_{21} & K_{22} \end{bmatrix} \quad (2.26)$$

Applying the Schur matrix inversion formula from Equation (2.18), we get

$$\begin{aligned} \Sigma^{-1} &= \begin{bmatrix} \Sigma_{11} & \Sigma_{12} \\ \Sigma_{21} & \Sigma_{22} \end{bmatrix}^{-1} \\ &= \begin{pmatrix} (\Sigma/\Sigma_{22})^{-1} & -(\Sigma/\Sigma_{22})^{-1}\Sigma_{12}\Sigma_{22}^{-1} \\ -\Sigma_{22}^{-1}\Sigma_{21}(\Sigma/\Sigma_{22})^{-1} & \Sigma_{22}^{-1} + \Sigma_{22}^{-1}\Sigma_{21}(\Sigma/\Sigma_{22})^{-1}\Sigma_{12}\Sigma_{22}^{-1} \end{pmatrix} \end{aligned}$$

This means we have that:

$$(\Sigma^{-1})_{11} = (\Sigma/\Sigma_{22})^{-1} = (\Sigma_{11} - \Sigma_{12}\Sigma_{22}^{-1}\Sigma_{21})^{-1} = K_{11} \quad (2.27)$$

and also that

$$K_{12} = -K_{11}\Sigma_{12}\Sigma_{22}^{-1} \quad \text{and} \quad K_{11}^{-1}K_{12} = -\Sigma_{12}\Sigma_{22}^{-1}. \quad (2.28)$$

Plugging these into the conditional mean and covariance formulas from above, we get:

$$\mu_{1|2}^c = \mu_1 + \Sigma_{12}\Sigma_{22}^{-1}(x_2 - \mu_2) = \mu_1 - K_{11}^{-1}K_{12}(x_2 - \mu_2) \quad (2.29)$$

and

$$\Sigma_{1|2}^c = \Sigma_{11} - \Sigma_{12}\Sigma_{22}^{-1}\Sigma_{21} = K_{11}^{-1} = \Sigma/\Sigma_{22} \quad (2.30)$$

We're almost done. Next, we further subpartition the vector x_1 as follows:

$$x_1 = \begin{bmatrix} x_{1,1} \\ x_{1,2} \end{bmatrix}, \quad \mu_1 = \begin{bmatrix} \mu_{1,1} \\ \mu_{1,2} \end{bmatrix}, \quad \text{and} \quad x = \begin{bmatrix} x_{1,1} \\ x_{1,2} \\ x_2 \end{bmatrix}. \quad (2.31)$$

This means that we have:

$$K_{11} = \begin{bmatrix} K_{11,11} & K_{11,12} \\ K_{11,21} & K_{11,22} \end{bmatrix} \quad (2.32)$$

and

$$\Sigma_{1|2}^c = \begin{bmatrix} \Sigma_{1|2,11}^c & \Sigma_{1|2,12}^c \\ \Sigma_{1|2,21}^c & \Sigma_{1|2,22}^c \end{bmatrix} = K_{11}^{-1}. \quad (2.33)$$

Conditional dependence is similar to the marginal independence — with marginal independence, we need zeros in the covariance matrix while in conditional dependence we need zeros in the conditional covariance matrix. This means that conditional factorization is analogous to marginal factorization. Conditional independence of the form $X_{1,1} \perp\!\!\!\perp X_{1,2}|X_2$ corresponds to factorization in the conditional distribution which means that $p(x_{1,1}, x_{1,2}|x_2) = p(x_{1,1}|x_2)p(x_{1,2}|x_2)$. When we consider $p(x_{1,1}, x_{1,2}|x_2)$ as in Equation (2.21), and compare that with $p(x)$ as in Equation (2.6), we are reminded that the thing that allowed $p(x)$ to be factored was zeros in Σ (namely that $\Sigma_{12} = 0$). But we can compare the Gaussian in Equation (2.6) with the conditional Gaussian in Equation (2.21). We see that zeros in Σ in Equation (2.6) is analogous to zeros in Σ/Σ_{22} within Equation (2.21). The only difference really is that in Equation (2.21) we are also conditioning on x_2 but we see that Σ/Σ_{22} is the same for any value of x_2 so the factorization condition of $p(x_{1,1}, x_{1,2}|x_2)$ does not depend on x_2 at all. That is, we would have conditional independence $X_{1,1} \perp\!\!\!\perp X_{1,2}|X_2$ if we have $\Sigma_{1|2,1,2}^c = 0$ in Equation (2.33). But again from Equation (2.33), this is equivalent to $K_{11,12} = 0$. Hence, we have proven the following theorem

Theorem 10. *Let $X \sim \mathcal{N}(\mu, \Sigma)$, Σ non-singular, then for the tri-partition $X = [X_1^\top \ X_2^\top \ X_3^\top]^\top$ of the variables, with $\mu = [\mu_1^\top \ \mu_2^\top \ \mu_3^\top]^\top$, and with:*

$$\Sigma = \begin{bmatrix} \Sigma_{11} & \Sigma_{12} & \Sigma_{13} \\ \Sigma_{21} & \Sigma_{22} & \Sigma_{23} \\ \Sigma_{31} & \Sigma_{32} & \Sigma_{33} \end{bmatrix} \quad (2.34)$$

and

$$K = \begin{bmatrix} K_{11} & K_{12} & K_{13} \\ K_{21} & K_{22} & K_{23} \\ K_{31} & K_{32} & K_{33} \end{bmatrix} \quad (2.35)$$

then $X_1 \perp\!\!\!\perp X_2 | X_3$ iff $K_{12} = 0$.

Thus, zeros in Σ correspond to marginal independence, and zeros in K correspond to conditional independence.

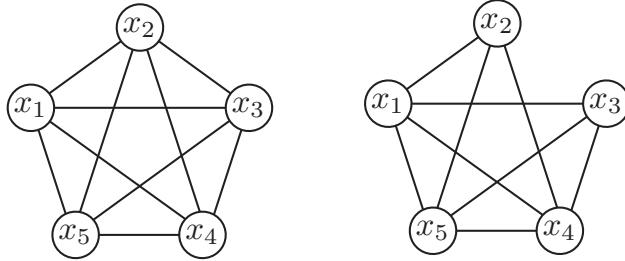


Figure 2.11: Left: a 5-clique (five notes all connected) is a graph whose family includes a 5D Gaussian with a concentration matrix with no zeros. Right: the missing edge says that any member of this graph's family must have a conditional independence property, namely that $X_2 \perp\!\!\!\perp X_3 | \{X_1, X_4, X_5\}$. The 5D Gaussian with a concentration matrix having $K_{2,3}$ is a member of this family.

The following is an example that illustrates this point. Consider $d = 5$ (a 5-dimensional Gaussian). If K has no zeros, the MRF corresponds to a clique as shown in Figure 2.11-Left. Suppose $K_{23} = 0 \Leftrightarrow X_2 \perp\!\!\!\perp X_3 | \{X_1, X_4, X_5\}$. The MRF for this is shown in Figure 2.11-Right.

In general, we see that the zeros in K correspond precisely to graphs that obey a Markov property (the pairwise Markov property in particular) associated with MRFs [258]. Moreover, since a Gaussian is everywhere positive, we see by the Hammersley-Clifford theorem that zeros in K correspond to missing edges in a MRF — a Gaussian p with zeros in K will be in the MRF family with a graph G having missing edges corresponding to the zeros in K . In other words, a graph that has edges (i, j) only where $K_{i,j} \neq 0$, and missing edges whenever the corresponding coefficient in K is zero, then this graph will contain the Gaussian as a member of its family. More discussion on this is given in [258].

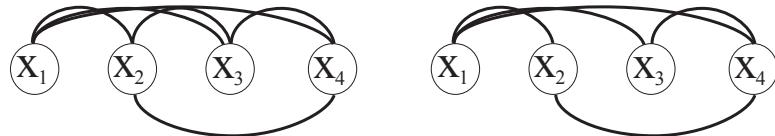


Figure 2.12: A Gaussian viewed as an MRF. On the left, there are no independence assumptions. On the right, $X_2 \perp\!\!\!\perp X_3 | \{X_1, X_4\}$.

2.2.2.4 Gaussians and Bayesian Networks

A Gaussian may also be viewed as a BN, and in fact many BNs. Unlike with a MRF, to form a Gaussian Bayesian network a specific variable ordering must first be chosen, the same ordering used to factor the joint distribution with the chain rule of probability. A Gaussian can be factored

$$p(x_{1:N}) = \prod_i p(x_i | x_{i+1:N})$$

according to some fixed but arbitrarily chosen variable ordering. Each factor is a Gaussian with conditional mean

$$\mu_{i|i+1:N} = \mu_i + \Sigma_{i,i+1:N}(\Sigma_{i+1:N,i+1:N})^{-1}(x_{i+1:N} - \mu_{i+1:N})$$

and conditional covariance

$$\Sigma_{i|i+1:N} = \Sigma_{ii} - \Sigma_{i,i+1:N}(\Sigma_{i+1:N,i+1:N})^{-1}\Sigma_{i+1:N,i}.$$

both of which are unique for a given ordering (these are an application of the conditional Gaussian formulas above, but with A and B set to the specific values $\{i\}$ and $\{(i+1):N\}$ respectively). Therefore, the chain rule expansion can be written:

$$p(x_{1:N}) = \prod_i (2\pi\Sigma_{i|i+1:N})^{-1/2} e^{-\frac{1}{2}(x_i - \mu_{i|i+1:N})^2 \Sigma_{i|i+1:N}^{-1}} \quad (2.36)$$

An identical decomposition of this Gaussian can be produced in a different way. Every concentration matrix K has a unique factorization $K = U^T D U$ where U is a unit upper-triangular matrix and D is diagonal [295, 184]. A unit triangular matrix is a triangular matrix that has ones on the diagonal, and so has a unity determinant (so is non-singular), therefore, $|K| = |D|$. This corresponds to a form of Cholesky factorization $K = R^T R$, where R is upper triangular, $D^{1/2} = \text{diag}(R)$ is the diagonal portion of R , and $R = D^{1/2}U$. A Gaussian density can therefore be represented as:

$$p(x) = (2\pi)^{-d/2} |D|^{1/2} e^{-\frac{1}{2}(x - \mu)^T U^T D U (x - \mu)}$$

The unit triangular matrices, however, can be “brought” inside the squared linear terms by considering the argument within the exponential

$$\begin{aligned} (x - \mu)^T U^T D U (x - \mu) &= (U(x - \mu))^T D(U(x - \mu)) \\ &= (Ux - \tilde{\mu})^T D(Ux - \tilde{\mu}) \\ &= ((I - B)x - \tilde{\mu})^T D((I - B)x - \tilde{\mu}) \\ &= (x - Bx - \tilde{\mu})^T D(x - Bx - \tilde{\mu}) \end{aligned}$$

where $U = I - B$, I is the identity matrix, B is an upper triangular matrix with zeros along the diagonal, and $\tilde{\mu} = U\mu$ is a new mean. Again, this transformation is unique for a given Gaussian and variable ordering. This process exchanges K for a diagonal matrix D , and produces a linear auto-regression of x onto itself, all while not changing the Gaussian normalization factor contained in D . Therefore, a full-covariance Gaussian can be represented as a conditional Gaussian with a regression on x itself, yielding the following:

$$p(x_{1:N}) = (2\pi)^{-d/2} |D|^{1/2} e^{-\frac{1}{2}(x - Bx - \tilde{\mu})^T D(x - Bx - \tilde{\mu})}. \quad (2.37)$$

In this form the Gaussian can be factored where the i^{th} factor uses only the i^{th} row of B :

$$p(x_{1:N}) = \prod_i (2\pi)^{-1/2} D_{ii}^{1/2} e^{-\frac{1}{2}(x_i - B_{i,i+1:N}x_{i+1:N} - \tilde{\mu}_i)^2 D_{ii}} \quad (2.38)$$

When this is equated with Equation (2.36), and note is taken of the uniqueness of both transformations, it is the case that

$$B_{i,i+1:N} = \Sigma_{i,i+1:N}(\Sigma_{i+1:N,i+1:N})^{-1}.$$

and that $\tilde{\mu}_i = \mu_i - B_{i,i+1:N}\mu_{i+1:N}$. This implies that the regression coefficients within B are a simple function of the original covariance matrix. Since the quantities in the exponents are identical for each factor (which are each an appropriately normalized Gaussian), the variance terms D_{ii} must satisfy:

$$D_{ii} = \Sigma_{i|i+1:N}^{-1}$$

meaning that the D_{ii} values are conditional variances.

Using these equations we can now show how a Gaussian can be viewed as a BN. The directed local Markov property of BNs [258] states that the joint distribution may be factorized as follows:

$$p(x_{1:N}) = \prod_i p(x_i | x_{\pi_i})$$

where $\pi_i \subseteq \{(i+1):N\}$ are parents of the variable x_i . When this is considered in terms of Equation (2.38), an inductive argument implies that the non-zero entries of $B_{i,(i+1):N}$ correspond to the set of parents of node i , and the zero entries correspond to missing edges. In other words (under a given variable ordering) the B matrix determines the conditional independence statements for a Gaussians when viewed as a Bayesian network, namely we have the following theorem:

Theorem 11. *In a Gaussian distribution, $X_i \perp\!\!\!\perp X_{\{(i+1):N\} \setminus \pi_i} | X_{\pi_i}$ if and only if the entries $B_{i,\{(i+1):N\} \setminus \pi_i}$ are zero.³*

It is important to realize that these results depend on a particular ordering of the variables $X_{1:N}$. A different ordering might yield a different B matrix, possibly implying different independence statements (depending on if the graphs are *Markov equivalent*, mean in they express identical conditional independence properties). Moreover, a B matrix can be sparse for one ordering, but for a different ordering the B matrix can be dense, and zeros in B might or might not yield zeros in $K = (I - B)^T D(I - B)$ or $\Sigma = K^{-1}$, and vice versa.

This means that a full covariance Gaussian with $N(N + 1)/2$ non-zero covariance parameters might actually employ fewer than $N(N + 1)/2$ parameters, since it is in the directed domain where sparse patterns of independence occur. For example, consider a 4-dimensional Gaussian with a B matrix such that $B_{12} = B_{13} = B_{14} = B_{24} = B_{34} = 1$, and along with the other entries of B that are normally zero (since it is an upper triangular matrix), also take $B_{23} = 0$ (see Figure 2.13). For this B matrix and when $D = I$, neither the concentration nor the covariance matrix has any zeros, although they are both full rank and it is true that $X_2 \perp\!\!\!\perp X_3 | X_4$. Another way of looking at this is that an MRF is unable to express the V-structure that exists in the Bayesian network view of this Gaussian. It must be that K possesses redundancy in some way, but in the undirected formalism it is impossible to encode this independence statement and one is forced to generalize the model family and to use a model that possesses no independence properties.

The opposite can occur as well, where zeros exist in K or Σ , and less sparsity exists in the B matrix. Take, for example the matrix

$$K = \begin{pmatrix} 5 & 2 & 0 & 4 \\ 2 & 9 & 1 & 0 \\ 0 & 1 & 5 & 3 \\ 4 & 0 & 3 & 6 \end{pmatrix}$$

with corresponding MRF

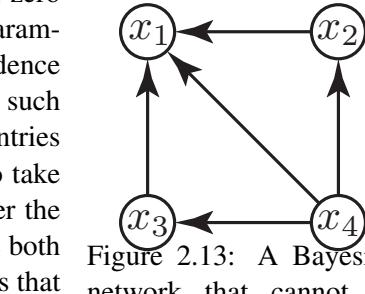
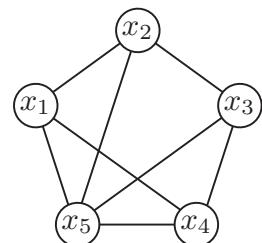


Figure 2.13: A Bayesian network that cannot be represented as an MRF due to V-structure $2 \rightarrow 1 \leftarrow 3$.



This concentration matrix states that $X_1 \perp\!\!\!\perp X_3 | \{X_2, X_4, X_5\}$ and $X_2 \perp\!\!\!\perp X_4 | \{X_1, X_3, X_5\}$, but the corresponding B matrix (using the standard order) has only a single zero in its upper portion reflecting only the first independence statement. This is because of the subgraph of the graph corresponding to the unchorded 4-cycle $x_1 - x_2 - x_3 - x_4 - x_1$

³Standard notation is used here, where if A and B are sets, $A \setminus B$ is the set of elements in A that are not in B .

It was mentioned earlier that MRFs and Bayesian networks represent different families of probability distributions, and this is reflected in the Gaussian case above by a reduction in sparsity when moving between certain B and K matrices. It is interesting to note that Gaussians are able to represent any of the dependency structures captured either in a Bayesian network (via an appropriate order of the variables and zeros in the B matrix) or a MRF (with appropriately placed zeros in the concentration matrix K). Therefore, Gaussians, along with many other interesting and desirable theoretical properties, are quite general in terms of their ability to possess conditional independence relationships. Another way of saying that is that for any G , we can find a Gaussian p such that $p \in \mathcal{F}(G, \mathcal{M})$ but that $p \notin \mathcal{F}(G', \mathcal{M})$ where G' is any spanning subgraph of G (i.e., G' has one or more edges missing relative to G).

The question then becomes what form of Gaussian should be used, a Bayesian network or a MRF, and if a Bayesian network, in what variable order. A common goal is to minimize the total number of free parameters. If this is the case, the Gaussian should be represented in a “natural” domain [28], where the least degree of parameter redundancy exists. Sparse matrices often provide the answer, assuming no additional cost exists to represent sparse matrices, since the sparse pattern itself might be considered a parameter needing a representation. This was exploited in [46], where the natural directed Gaussian representation was solicited from data, and where a negligible penalty in word error performance was obtained with a factored sparse covariance matrix having significantly fewer parameters.

Lastly, it is important to realize that while all MRF or Bayesian network dependency structures can be realized by a Gaussian, the implementations in each case are still only linear and the random residual terms are only univariate Gaussian. A much greater family of distributions, other than just a Gaussian, are included in either the MRF’s or the Bayesian network’s family.

Exercise 12. *Given a Bayesian network that contains V-structures (so it is not exactly representable with an MRF), and given a Gaussian distribution that has been set up to exactly and only represent the independence statements made by this Bayesian network, what happens to the corresponding MRF when we look at the same Gaussian parameters in MRF form? Conversely, given a Gaussian MRF that contains an unchorded 4-cycle (so it is not exactly representable with a Bayesian network), what happens when we convert it to Bayesian network form with matrix B ?*

2.2.3 Gaussian Parameterization and Sufficient Statistics

As mentioned earlier, a Gaussian distribution may be parameterized (and has sufficient statistics) considering of a mean and covariance matrix μ, Σ . In light of Equation (2.37), which we repeat again here

$$p(x_{1:N}) = (2\pi)^{-d/2} |D|^{1/2} e^{-\frac{1}{2}(x-Bx-\tilde{\mu})^T D(x-Bx-\tilde{\mu})}, \quad (2.39)$$

we see that a Gaussian can also be expressed via (non-redundant) sufficient statistics consisting of:

1. A conditional mean μ ,
2. A vector of conditional variances values $\text{diag}(D)$ (meaning that we take the vector consisting of the diagonal entries of D which of course are the only possible non-zero entries), and
3. the conditional regression values B .

GMTK uses this parameterization of a Gaussian (μ, D, B) rather than the pair (μ, Σ) for several reasons.

1. It is easy to express full covariance matrices (the upper triangular portion of B has no zeros), diagonal covariance matrices ($B = 0$), and sparse variants where, as mentioned above, the sparse patterns within B correspond to the missing edges in a Bayesian network view of the Gaussian. It is recently becoming apparent in the machine learning community that sparse models have many advantages.

2. Within the same framework, it is possible to express conditional Gaussians of the form $p(x|z)$ where x is a standard set of features (such as MFCCs for speech recognition) and z are some auxiliary features that are to be used only conditionally. Alternatively, it might be that we want a model of the form $p(x|z)p(z)$ where the sparsity pattern is quite different between these two factors. In any event, using the tripartite representation (μ, D, B) makes this easy.
3. It offers a rich set of parameter sharing. I.e., as we will see later, in GMTK it is possible to share any element of (μ, D, B) for one Gaussian with the corresponding element in another Gaussian. While this significantly complicates EM training (as described in [49]), it does offer greater flexibility than the case where we only may share either means or covariance matrices.

It might be asked why was it not the case that the MRF view of Gaussians was not also included in GMTK so that sparse Gaussians corresponding to the 4-cycle could also be exactly represented. While this might happen in the future, one issue is that with an MRF it is easy to get into situations where the Gaussian no longer has a closed form update for maximum likelihood training [258]. That is, the MRF corresponding to the Gaussian might first need to be triangulated before being able to use it during training (so that the parameter bundles decompose into groups that may be updated in closed-form).

We will discuss Gaussians again in the context of GMTK's use of them for observation modeling (see §).

2.3 What families are not representable using graphs?

As mentioned in §2.1, a graphical model with its set of Markov properties specifies a family $\mathcal{F}(G, \mathcal{M})$. Given a graph G with enough edges, then for any p , we have $p \in \mathcal{F}(G, \mathcal{M})$, but it might be (as we saw in §2.2.1) that p has unique properties not expressible by a graphical model. Are there useful families that can not be represented by any graph?

The answer of course depends on the type of graph. Pure “graphical model” research involves developing new formal graph theoretic mechanisms that can describe families of probability distributions. Graphs themselves are rich, and are able to be annotated in an unlimited number of ways. Hence, it might be that one could develop a graphical model that has the specificity to either include or exclude any given p with whatever peculiarities it has. Right now, however, the types of graphical models that exist (e.g., Bayesian networks, MRFs, factor graphs, chain graphs ancestral graphs, value specific independence, sparse models, etc.) all correspond to some form of factorization property. As mentioned above, there are other ways of restricting a given p that are not describable via current graphical model technology. These include the log supermodular/ferromagnetic distributions used in image processing and statistical physics, and also log submodular distributions such as determinantal point processes.

Another example is when the number of variables n is not fixed, and where the graphical model might need to expand in response to a given sample of data. One such example is dynamic graphical models where the graph may expand in one dimension, but other examples exist as well (cf. the template models in §8.1). Indeed, our focus in this document is on GMTK which operates on dynamic graphical models to which we next turn our attention.

Chapter 3

Undirected Graphical Models

In this chapter, we start discussing and defining graphical models and their semantics. In particular, we define a type of undirected graphical model (i.e., the graph edges do not have directions associated with it) called a Markov random field (or MRF). We do this *before* moving on to directed graphical models (such as Bayesian networks) since MRFs have a much easier semantics than do their directed counterpart. MRFs are sufficient to understand the ideas of a family and the constraints that a graphical model specifies and how that relates to both generality and specificity of a class of models. With a firm grounding of UGMs under our belt, we will in the following chapter discuss Bayesian networks (Chapter ??). We'll move on to dynamic models in later chapters.

We will first discuss traditional “Markov random fields” and will follow that up with factor graphs. As we will see, factor graphs can also be seen as a form of undirected model although they are different in a number of ways as well (factor graphs are best considered as undirected bipartite graphs). Thus, while a factor graph is an UGM, not all UGMs are factor graphs.

We will be somewhat loose on the distinction between distributions, mass functions, and densities in this chapter (which would allow us to more formally discuss mixed continuous/discrete distributions). That is, like before, we have that $p(x) = p(x_{1:N}) = p(x_1, \dots, x_N)$ is a distribution over N random variables, and we will for the most part assume that they are discrete random variables (i.e., that the domain of values that random variable X may take on, i.e., D_X , is a finite discrete set).

3.1 What is a graphical model?

We explain the general concepts behind what a graphical model is in the context of defining a Markov random field. We note that the definition of a graphical model is a little bit subtle, so we ask the reader not to finalize in stone their own conclusions regarding their understanding of a graphical model until fully reading and understanding all of this chapter. Once grounded in these ideas, it will be easy to explain other types of graphical model (such as Bayesian networks and factor graphs).

What is a model? We will be using the word “model” quite a bit. We know that a model airplane is like a real airplane in some way, but it is much smaller and does not do all the things that a real airplane can do. Some model airplanes are more accurate than others — a model airplane for a 2-year old would certainly not have the same representational accuracy as one for a 17-year old, let alone a model for an adult model enthusiast. Each model is accurate enough for its purpose (the 2-year old would get the idea of a model airplane even if it is only cartoon like), and not too accurate for its purpose (the 2-year old would not only be unappreciative of the enthusiast model, but would probably end up breaking it).

What is a mathematical or statistical model? Like a model airplane, a statistical model is meant to look like a real physical system. A model might have certain properties that are appropriate for the task at hand, be that the proof of a theorem, or the use of a model as a practical tool. Given a (say physical)

process for which we might have complete or partial information, a model of that process is a mathematical description of certain important properties of that process, where the meaning of “important” depends on the context. Example processes include weather patterns on earth, the way that a sequence of bases in a DNA string change from one generation to the next, or the relationship between an acoustic speech waveform and the words that comprise it. The reason for needing such a model is many fold. As was mentioned in the introductory chapter, we may wish to make predictions about that process in the future, or we may wish to predict aspects of the process that we can not directly observe. A statistical model is a model where it is acknowledged that we are unable to know elements of the process with certainty and instead those elements are represented with controlled randomness. The ideal statistical model would be such that it is computationally possible to know all of the elements with certainty, but this is almost never the case. Randomness buys us several things: it allows us to represent and learn about the process without observing all of it (there might be missing or hidden information), and it allows us in many cases to compute statistical properties of that process (such as expectations) computationally efficiently, something that, without the expression of uncertainty, would not be possible. Randomness is also our expression of uncertainty. We acknowledge and utilize the inherent uncertainty to help us make decisions. An example is the notion of volatility of an economic indicator: even if we predict that the indicator will rapidly rise, we might not wish to bet in favor of that indicator if the volatility (or variance) of our prediction is also high.

A statistical model itself is a probability distribution. That is, there are a set of discrete or continuous random variables that are related in such a way that for any assignment to those variables we can perform probabilistic queries (such as, in the discrete case, asking for the probability of a set of variables).

A graphical model, like any model, is an expression of a process, but unlike a typical model, a graphical model represents a multitude (or family) of distributions. This is a key point so we restate it more formally below.

There are many different types of graphical models. Each type of graphical model has rules that: 1) say what kinds of graphs are valid within that type of graphical model, and 2) for any valid graph within, what that graphical model means. These rules are formal precise mathematical statements that are derived from the graph, so it is not possible to speak precisely about a graphical model without knowing the rules that apply to that particular type of graphical model. As an example, Bayesian networks (which are directed) and UGMs have different rules that define what they mean. There are different types of directed graphs (e.g., Bayesian networks and ancestral graphs) which use different rules. Sometimes we might have two very different looking rules, but it can be proven that the rules yield the same thing.

We are discussing undirected models, and in particular, Markov random fields (MRF). First, we are given an undirected graph $G = (V, E)$ consisting of a set of nodes V and a set of edges $E \subseteq V \times V$, so each edge is a pair of nodes. Recall, we can recover the nodes from G using notation $V(G)$ (and edges using $E(G)$) treating V (and E) as functions on graphs. For an MRF, there are no restrictions on the undirected graph so all undirected graphs are allowed. This includes a graph that is fully connected (or a clique, meaning that all pairs of nodes are connected with an edge) to the other extreme where no two nodes are connected.

A key property in any graphical model is that properties of a graph help us to infer aspects of families of probability distributions. I.e., the “graphical” part of graphical model must mean that the graph itself delineates a family of probability models. A family \mathcal{F} of models is seen a set of distributions, so we will be able to talk about set inclusion, intersection, and so on. We say that two families of models \mathcal{F}_1 and \mathcal{F}_2 are identical if the sets are equal, $\mathcal{F}_1 = \mathcal{F}_2$.

It is almost always the case that the nodes of a graph $V(G)$ correspond to random variables. That is, for each $v \in V$, we have a random variable X_v . The graph in one way or another informs us about properties of $p(X_V)$, the distribution of the vector random variable X_V . One could conceivably also define a graphical model so that each edge $e \in E$ corresponds to a random variable, but this is typically not done. A graphical model corresponds to a family of distributions over X_V , and this family can't be defined unless there is an accompanying set of rules that come along with the graph.

Since it is not possible to define a graphical model without a set of rules, we formally define an undirected graphical model as a graph $G = (V, E)$ and a set of rules that precisely define what that graph means. I.e., a graphical model is a pair $(G, R) = ((V, E), R)$ where R are a set of rules that map from graph $G = (V, E)$ to statements about probability distributions, where for each probability distribution p there is a one-to-one relationship between the nodes in G and the random variables described by p . An individual rule $r \in R$ is a predicate (truth statement) that takes a graph and a probability distribution, and evaluates to either true or false.

For example, a given rule $r \in R$ is such that for a given distribution p and graph G , $r(p, G)$ is either true or false. Crucially, a graphical model corresponds to a **set** or a **family** of probability distributions. Each member of this family is a distribution over random variables, where there is a random variable corresponding to each node of the graph. Let X_V correspond to this set of random variables, i.e., $X_V = \{X_v : v \in V\}$ where X_v is an individual random variable.

Notationally, we can and will refer both to a node in a graph and its corresponding random variable. That is, $v \in V(G)$ is a node in the graph $G = (V, E)$, and X_v is the random variable corresponding to node v and a possible value for this random variable might be x_v or \bar{x}_v , so we can say things like $p(x_v) \triangleq p(X_v = x_v)$.

We can define a family function \mathcal{F} that, implicitly using a set of rules, maps from a graph to a family of probability distributions over random variables.¹

$$\mathcal{F}(G, R) = \{p : p \text{ is a distribution over } X_V \text{ and } r(p, G) = \text{true}, \forall r \in R\} \quad (3.1)$$

I.e., $\mathcal{F}(G, R) \subseteq \mathcal{U}$ is the set of all probability distributions over $|V|$ random variables such that every rule in R applied to the distributions in \mathcal{F} evaluates to true.

As can be seen, this is a formal statement about how to express a family of distributions given a graph G and a set of rules R .

What do the rules actually mean? The rules in R state what **must** is required (i.e., what must be true) of all members of the family. If a given p obeys all of the rules, then it is a member of the family. A given member might also obey additional rules that are not specified by the rule set, but they are still part of the family. You can think of the rule set as a filter over the set of all $p \in \mathcal{U}$, and R filters out those $p \in \mathcal{U}$ that do not satisfy the rules, but R does not filter out p that satisfies rules that are not specified within R . I.e., there could be other rules $r \notin R$ such that r is true of a $p \in \mathcal{F}(G, R)$. There might be other ways of specifying a family, for example, members might satisfy the given set of rules R and no other rules, but this would be more restrictive. Those p that satisfy only R and nothing else constitute a sub-family of the family.

Figure 3.1 shows a Venn-diagram explaining this idea. The graph, as will become known as the 4-cycle, along with a set of rules, defines the sub-family such that all members of the family obey the rules.

Before moving on, it may be worth giving an example of the rules. Lets say that a rule $r \in R$ states that “if there are two nodes $u, v \in V$ that are neither directly nor indirectly connected in G (i.e., there no path leading from u to v in G) then the corresponding random variables in p are independent”. Given such a rule, we can see that if a graph corresponds to two nodes that are not connected, then this graphical model will describe the family of all probability distributions over two independent random variables.

This immediately leads to the following set of questions:

- For a given type of graphical model, can the rule set R be listed in finite space and computed efficiently? (answer, yes).
- For a given type of graphical model, are there more than one set of rules that define a family? In other words, are there rule sets R_1 and R_2 such that $\mathcal{F}(G, R_1) = \mathcal{F}(G, R_2)$ for all G ? The answer is yes. In fact, this is such an important point, we will be discussing a variety of rule sets that for MRFs are

¹Note we are now notationally using \mathcal{F} to refer to a function that maps from a graph and a set of rules to the family of distributions that obey those rules for the graph.

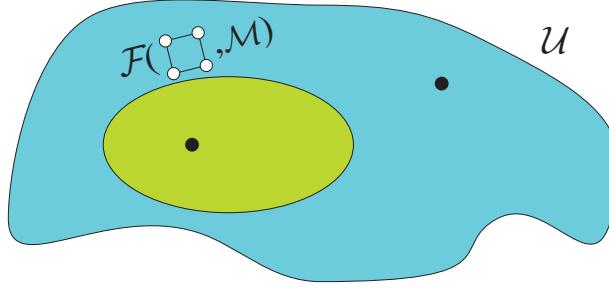


Figure 3.1: The set \mathcal{U} is the family of all possible probability distributions over N random variables $p(x_1, \dots, x_N)$ and $\mathcal{F}(G, R) \subset \mathcal{U}$ is a subset of distributions. Every $p \in \mathcal{F}(G, R)$ obeys all of the rules given in R when applied along with graph G .

all mathematically equivalent in that they define the same family of graphical models. The reason that this is useful as sometimes it is easier to infer certain properties of the family using one rule set over another.

- Is there a smallest rule set? In other words, are there rules sets $R_1 \subset R_2$ such that $\mathcal{F}(G, R_1) = \mathcal{F}(G, R_2)$, and can we compute the smallest rule set R' so that $\mathcal{F}(G, R') = \mathcal{F}(G, R)$ where $|R'|$ is minimal?
- Are there rule sets that are non-equivalent? I.e. R_1 and R_2 such that $\mathcal{F}(G, R_1) \neq \mathcal{F}(G, R_2)$ for some G ? Answer, yes.

Now $\mathcal{F}(G, R)$ is the family of probability distributions over N variables that obey rules R for the specific graph G . Let \mathcal{G}_N be the set of all undirected graphs over N nodes (there are $2^{\binom{N}{2}}$ such graphs). Then consider

$$\mathcal{F}_N(R) = \bigcup_{G \in \mathcal{G}_N} \mathcal{F}(G, R) \quad (3.2)$$

is the set of all graphs over N nodes that obey rules R , and

$$\mathcal{F}(R) = \bigcup_{N=1}^{\infty} \mathcal{F}_N(R) \quad (3.3)$$

is the family of all distributions over any number of random variables that obeys rules R .

Then the type of graphical model is determined by R . That is, let $R^{(\text{mrf})}$ be a set of rules that define a MRF. Then $\mathcal{F}(R^{(\text{mrf})})$ is the models specifiable by Markov random fields. Let $R^{(\text{bn})}$ be the set of rules that define a Bayesian network. Then $\mathcal{F}(R^{(\text{bn})})$ is the set of models specifiable by a Bayesian network. In the MRF case, the graphs G that we union over above is the set of all undirected graphs. In the BN case, the set of graphs that we union over is the set of all directed acyclic graphs.

We have said that there are different types of graphical models. What this really means is that each type of graphical model has a different set of rules, and a different allowable set of graphs. Given two rule sets R^A and R^B we could have any possible relationship between the families (for example, any of $\mathcal{F}(R^A) \subset \mathcal{F}(R^B)$, $\mathcal{F}(R^B) \subset \mathcal{F}(R^A)$, $\mathcal{F}(R^B) = \mathcal{F}(R^A)$, etc.). Different types of graphical models would not be truly different if it was the case that their families were identical. In fact, much of graphical model research is to show that for a certain rule set and/or model time, we can show certain set theoretic relationships between the corresponding families. As an example, when we say that Bayesian networks are not the same as Markov random fields, what we mean is that

$$\mathcal{F}(R^{(\text{mrf})}) \neq \mathcal{F}(R^{(\text{bn})}) \quad (3.4)$$

and more over, we have that neither $\mathcal{F}(R^{(\text{mrf})}) \subseteq \mathcal{F}(R^{(\text{bn})})$ nor $\mathcal{F}(R^{(\text{bn})}) \subseteq \mathcal{F}(R^{(\text{mrf})})$, so there are MRFs that are not BNs and vice versa. Moreover, the intersection of the two $\mathcal{F}(R^{(\text{mrf})}) \cap \mathcal{F}(R^{(\text{bn})})$ has some interesting properties.

Note that it should be clear in the above that the family of graphs is also important. I.e., for a given set of rules R , any $r \in R$ takes only a certain type of graphs. For example, if R is the rules that define MRFs, and $r \in R$, then $r(p, G)$ will only work on a graph G that is undirected over N variables. The rule $r(p, G)$ will not work on a directed model. We do not explicitly note this and assume that when we define constructs like $\mathcal{F}_N(R)$ we are taking the union only over appropriate graph classes (directed, undirected, bipartite, etc.).

To summarize, different types of graphical model are determined based on both the rule set and the valid graph types. We can talk about the family of distributions corresponding to a given graph and set of rules (which define the meaning of a particular graph) and we may also talk about all of the distributions that correspond to a particular type of graphical model. We may formally reason about how these families relate to each other.

To make these ideas concrete, we start out talking specifically about the rules that define the family of distributions that like within MRFs.

3.2 Markov Random Fields - a form of undirected model

In this section, we define MRFs using a variety of rule sets and proof that they are equivalent to each other (i.e., lead to the same family).

Markov random fields (MRFs) are a class of probability models that came from the field of statistical physics and have traditionally been used for image processing problems. The basic idea is that there are a collection of random variables that are related to each other via some form of interaction function, and that the score of these interactions are then summed together, exponentiated, and then normalized to produce a valid probability model.

For example, the Ising model from statistical physics defines how the spin of certain atomic particles relate to each other. Let $W = [w_{ij}]_{ij}$ be a matrix of weights. We expect that many of these weights will be zero. Let $S = [s_i]_i$ be a vector of binary random variables, $s_i \in \{-1, +1\}$. Define a quantity called the “energy” as

$$E = - \sum_{ij} s_i s_j w_{ij} \quad (3.5)$$

Given this, we can define a joint distribution over the vector of random variables S as follows:

$$p(s) = \frac{1}{Z} \exp(-E/T) \quad (3.6)$$

where T is called the temperature of the model. Most often, the vector S corresponds to a grid (i.e., S is really a matrix or 3D-matrix) and w_{ij} determines the interaction style between the two variables S_i and S_j . For example, if $w_{ij} = 0$ then there is no interaction. If $w_{ij} > 0$ then more probable for $s_i = s_j = \pm 1$. If $w_{ij} < 0$, then it is more probable for $s_i \neq s_j$. To find the most probable configuration of S , we are unable to use the local interactions between any two pairs of variables, but rather all variables have to be considered together (since changing a value of one variable will cause a cascade of indirect consequences throughout the entire set of other variables). We think of matrix W and vector S as a graph, $G = (V, E)$ where the variables S correspond to V , and W corresponds to E ($(i, j) = e \in E$ only when $w_{ij} \neq 0$). For example, if the graph in Figure 3.2 corresponded to an Ising model, $w_{ij} = 0$ for all (i, j) except when $(i, j) \in \{(1, 2), (1, 3), (2, 3), (2, 4), (3, 4), (2, 5)\}$ which constitute the edges of this graph. As will be seen, we will find that for any Ising model p , $p \in \mathcal{F}(G, R^{(\text{mrf})})$ for the appropriately defined MRF rules. This means that any Ising model is an MRF.

More generally, we wish to define a set of rules that define the family of MRF models. What we will do is define a number of different rule sets and see how they relate to each other, and in doing so, the definition of MRFs will be given at the end of this section.

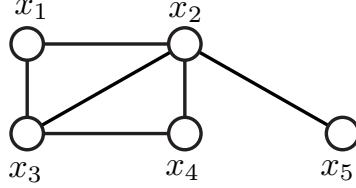


Figure 3.2: In this figure, we see an example of the rules of clique factorization specified in Equation (4.2). On the left we see a graph G over the variables x_1, \dots, x_5 . The cliques in the graph are $\mathcal{C} = \{1, 2, 3, 4, 5, \{1, 2\}, \{1, 3\}, \{2, 3\}, \{1, 2, 3\}, \{3, 4\}, \{2, 4\}, \{2, 3, 4\}, \{2, 5\}\}$. For any $p \in \mathcal{F}(G, \mathcal{M}^{(cf)})$, it must be possible to represent p as a product of factors over variables corresponding to these cliques, as shown in Equation (3.10). The maxcliques of this graph are $\{\{1, 2, 3\}, \{2, 3, 4\}, \{2, 5\}\}$ and for any $p \in \mathcal{F}(G, \mathcal{M}^{(mcf)})$ we must be able to write p as in Equation (??).

The first rule set we define is called *clique factorization*. Recall, a clique in a graph $G = (V, E)$ is any subset of nodes $C \subseteq V$ such that for all pairs $u, v \in C$, u and v are connected (i.e., C is a clique in G if $\forall u, v \in C, (u, v) \in E$). Let $\mathcal{C}(G)$ be the set of all cliques in the graph G , and let $C \in \mathcal{C}(G)$ be a particular clique in the graph. Consider a probability distribution that can be represented as follows:

$$p(x_V) = \prod_{C \in \mathcal{C}} \phi_C(x_C) \quad (3.7)$$

where $\phi_C(x_C)$ is a non-negative function of only the random variables X_C . We see that such a distribution may be represented by a product of a set of functions, where each function operates only on the variables in the local clique. Note that sometimes there might be a normalizing constant out in front of the form $1/Z$ where

$$Z = \sum_{x_V} \prod_{C \in \mathcal{C}} \phi_C(x_C) \quad (3.8)$$

but we can always place this constant into one of the clique functions without loss of generality. For the immediate discussion, we make this assumption.

Consider the following family:

$$\mathcal{F}(G, \mathcal{M}^{(cf)}) = \left\{ p : \forall C \in \mathcal{C}(G), \exists \psi_C : D_{X_C} \rightarrow \mathbb{R}^+ \text{ and } p(x_V) = \prod_{C \in \mathcal{C}} \psi_C(x_C) \right\} \quad (3.9)$$

In other words, the family comprises all probability distributions such that there exists local non-negative clique functions (defined on the appropriate domain so $\psi_C : D_{X_C} \rightarrow \mathbb{R}^+$), and where these local clique functions define the distribution when multiplied together. The above says nothing about uniqueness, in fact we can easily multiply one ϕ factor for one clique by a positive constant and divide any other factor by the same constant and we would get another set of functions yielding the identical distribution. Some of the clique functions might always evaluate to a constant. What matters in this definition is only the factorization property and that there exists functions of the corresponding variable arguments in terms of cliques.

Note also that the rule $R^{(cf)}$ is implicitly used in the above — the rule set contains a single rule $r(p, G)$ that, to evaluate to true, says that there must exist functions for p that correspond to (or “respect” the) cliques in graph G in such a way that the distribution p may be written as a product of these clique functions. Often the rules will be used like above, only implicitly.

An example graph is given in Figure 3.2 which corresponds to all distributions $p(x)$ that can be factored as follows:

$$\begin{aligned} p(x_{1:N}) &= \frac{1}{Z} \phi_1(x_1) \phi_2(x_2) \phi_3(x_3) \phi_4(x_4) \phi_5(x_5) \phi_{1,2}(x_1, x_2) \phi_{1,3}(x_1, x_3) \phi_{2,3}(x_2, x_3) \\ &\quad \phi_{1,2,3}(x_1, x_2, x_3) \phi_{3,4}(x_3, x_4) \phi_{2,4}(x_2, x_4) \phi_{2,3,4}(x_2, x_3, x_4) \phi_{2,5}(x_2, x_5) \end{aligned} \quad (3.10)$$

Note that in the equation, there is in some sense a certain redundancy in that, for example, $\phi_1(x_1)$ can be absorbed into $\phi_{1,2}(x_1, x_2)$. That is, given any factorization like the above, we can define a new factorization with new factor $\phi'_{1,2}(x_1, x_2) = \phi_{1,2}(x_1, x_2)\phi_1(x_1)\phi_2(x_2)$ and $\phi'_1(x_1) = \phi_1(x_1) = 1$ and we would have the same distribution. For these rules, we do not mind this redundancy, we only insist that such a factorization must be possible for p to be a member of $\mathcal{F}(G, \mathcal{M}^{(cf)})$.

The next rule set we will use is called *maxclique factorization*, or $R^{(mcf)}$. We distinguish between a clique in a graph (as defined above) and a maximal clique (or *maxclique*) where the latter is defined as a clique that cannot be enlarged with any node while remaining a clique (i.e., if any node is added to a maxclique then it is no longer a clique). There are never more maxcliques in a graph than there are cliques. There are only three maxcliques in Figure 3.2 which are $\{\{1, 2, 3\}, \{2, 3, 4\}, \{2, 5\}\}$.

As an aside, we will at some point refer to a “maximum clique” which is the largest possible maxclique. A maximum clique is necessarily a maxclique but a maxclique is not necessarily a maximum clique.

The rule $R^{(mcf)}$ says that there must exist maxclique functions with respect to the graph G in such a way that the distribution may be written as a product of the maxclique functions. That is, let $\mathcal{C}^{(mc)}(G)$ be the set of maximal cliques in graph G . Then we have the following family:

$$\mathcal{F}(G, R^{(mcf)}) = \left\{ p : \forall C \in \mathcal{C}^{(mc)}(G), \exists \psi_C(x_C), \text{ and } p(x_V) = \prod_{C \in \mathcal{C}^{(mc)}} \psi_C(x_C) \right\} \quad (3.11)$$

It should be clear that the above two families are identical, i.e., for any graph G , we have that $\mathcal{F}(G, R^{(mcf)}) = \mathcal{F}(G, R^{(cf)})$. The reason is that all cliques in a graph are properly contained in at least one maxclique and therefore we can assign every clique to one maxclique and define every maxclique function as the product of its corresponding clique functions, leading to the same resulting families. Going the other way starting from the maxcliques, we can always define additional unity functions over the subsets of maxcliques for any non-maxclique clique in the graph.

As an example, consider the graph G in Figure 3.2. Any distribution $p \in \mathcal{F}(G, R^{(cf)})$ can be written as in Equation (3.10). There are three maxcliques in the graph and we can absorb any factor over a clique that is not a maxclique into one of the maxclique factors that contain it. That is, we can re-write the factorization in Equation (3.10) as:

$$\begin{aligned} p(x_{1:N}) &= \frac{1}{Z} \underbrace{\phi_1(x_1) \phi_2(x_2) \phi_3(x_3) \phi_{1,2}(x_1, x_2) \phi_{1,3}(x_1, x_3) \phi_{2,3}(x_2, x_3) \phi_{1,2,3}(x_1, x_2, x_3)}_{\phi'_{1,2,3}(x_1, x_2, x_3)} \\ &\quad \underbrace{\phi_4(x_4) \phi_{3,4}(x_3, x_4) \phi_{2,4}(x_2, x_4) \phi_{2,3,4}(x_2, x_3, x_4)}_{\phi'_{2,3,4}(x_2, x_3, x_4)} \underbrace{\phi_5(x_5) \phi_{2,5}(x_2, x_5)}_{\phi'_{2,5}(x_2, x_5)} \end{aligned} \quad (3.12)$$

$$= \phi'_{1,2,3}(x_1, x_2, x_3) \phi'_{2,3,4}(x_2, x_3, x_4) \phi'_{2,5}(x_2, x_5) \quad (3.13)$$

by defining new factors $\phi'_{1,2,3}(x_1, x_2, x_3)$, $\phi'_{2,3,4}(x_2, x_3, x_4)$, and $\phi'_{2,5}(x_2, x_5)$, over the maxcliques. Going the other way, we can always add additional factors over subsets of maxcliques by defining the factors to be identically unity. For example, given the maxclique function $\phi'_{2,5}(x_2, x_5)$, we can write this as $\phi'_{2,5}(x_2, x_5)\phi''_5(x_5)$ where $\phi''_5(x_5) = 1$ for all x_5 .

Therefore, we have the following theorem.

Theorem 13. For all undirected graphs $G = (V, E)$, we have $\mathcal{F}(G, R^{(mcf)}) = \mathcal{F}(G, R^{(cf)})$.

Since these rules are identical, we will refer to them as the *factorization* rule, or $R^{(f)}$, and the corresponding family for graph G as $\mathcal{F}(G, \mathcal{M}^{(f)})$.

The point of initially defining these two rule sets is to point out that one might obtain the family with different rule sets. Most often, however, it is not so obvious that two different rule sets yield the same families and in such cases the equivalence needs to be formally proven. In many cases, different rule sets will not yield the same family. We touch on this issue again when we discuss generality and specificity (Section 3.4).

Recall that if $A, B, C \subset V$ are three disjoint index sets of a set of V random variables, then $X_A \perp\!\!\!\perp X_B | X_C$ iff $p(x_A, x_B | x_C) = p(x_A | x_C)p(x_B | x_C)$ for all x_A, x_B, x_C .

Conditional independence and separation in MRFs are very related in this next rule. Recall, separation in an undirected graph means that if we have three disjoint sets of nodes in a graph, A, B, C , we say that A is separated from B by C if it is the case that all paths from any node in A to any node in B must go through some node in C . We will use $A \perp\!\!\!\perp B | C$ as notation to indicate such graph separation in graph G .

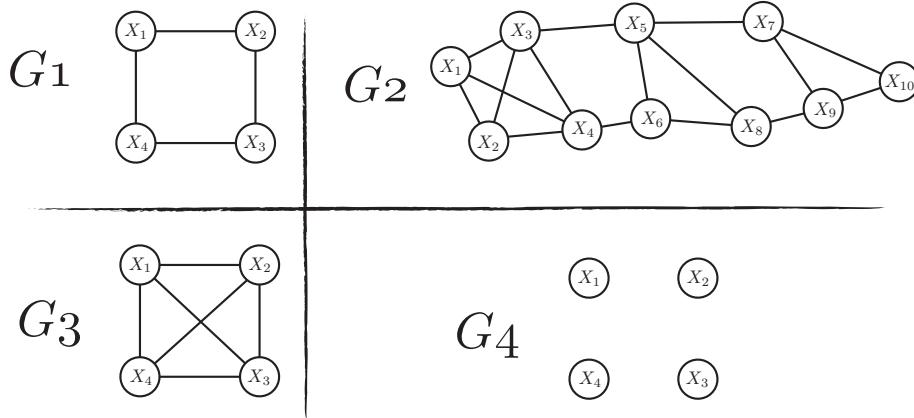


Figure 3.3: Four examples of undirected graphs.

The next rule we define is called the “global Markov property” rule. This rule states that for all sets A, B, C of nodes such that C separates A from B , an abiding p should have the following conditional independence statement $X_A \perp\!\!\!\perp X_B | X_C$. We can state the corresponding family as:

$$\mathcal{F}(G, R^{(g)}) = \{p : X_A \perp\!\!\!\perp X_B | X_C \text{ in } p \text{ whenever } A \perp\!\!\!\perp B | C \text{ in } G\} \quad (3.14)$$

It is crucial to realize that on the one hand, $X_A \perp\!\!\!\perp X_B | X_C$ is a statement about (or perhaps better, a constraint on) the corresponding set of probability distributions that live in the family, and that $A \perp\!\!\!\perp B | C$ is a statement about separation in a graph. The graph is what determines, via separation, the conditional independence statements that must be true in any member of the family.

As an example, consider G_1 in Figure 3.3. Since nodes x_1, x_3 separate x_2, x_4 we have that $X_2 \perp\!\!\!\perp X_4 | \{X_1, X_3\}$ in any abiding distribution and since nodes x_2, x_4 separate x_1, x_3 we have that $X_1 \perp\!\!\!\perp X_3 | \{X_2, X_4\}$. There are no other separation properties in this graph, so any p over four variables that satisfies these two conditional independence statements are a member of the family $\mathcal{F}(G_1, \mathcal{M}^{(g)})$.

Note that members of the family can have more true statements than just what is implied by the graph. For example, the distribution of variables that are all independent is one such trivial case. This means that for the graphs in Figure 3.3, for any $p \in \mathcal{F}(G_4, \mathcal{M}^{(g)})$, we have $p \in \mathcal{F}(G_1, \mathcal{M}^{(g)})$. Also, for any

$p \in \mathcal{F}(G_1, \mathcal{M}^{(g)})$, $p \in \mathcal{F}(G_3, \mathcal{M}^{(g)})$. $\mathcal{F}(G_3, \mathcal{M}^{(g)})$ contains all distributions over four random variables since there are no separators in the graph, so there are no conditional independence statements. No member of the family can violate any statement — there is no member of the family that can lack a conditional independence when the corresponding separation property exists in the graph.

The global Markov rule also captures the notion of marginal independence. In such case, if there is a graph with two connected components A and B (i.e., the two components are not connected to each other) then any variable in A is independent of any variable in B — conditioned on nothing, there is no path between any node in A and any node in B . Consider, for example, G_4 in Figure 3.3. For any p , this graph requires that, conditional on nothing, $X_1 \perp\!\!\!\perp X_2$, and in fact any set of variables are independent of any disjoint set of other variables unconditionally. The graph being disconnected states that this must be the case. Any distribution p where any variable is not independent of some other variable is not a member of $\mathcal{F}(G_4, \mathcal{M}^{(g)})$. Anytime we are given a graph that is connected (i.e., there is at least one path between any two nodes so it has one connected component), then this corresponds to a family distributions that are not required to have marginal independence properties at all (although they of course can if they wish).

We pause for a moment to point out that most rules regarding graphical models, and how they define a corresponding family of probability distributions, utilize graph separation in some way or another. What graph separation means, however, might be different depending on the type of graph — Bayesian networks, for example, use a completely different notion of separation (see Chapter XXX).

Our next rule is called the “local Markov property” rule. Recall, for a given node $v \in V$, we define we define $\text{cl}(v)$ (closure) to be the set consisting of v and all of v ’s immediate neighbors, and we define $\text{bd}(v)$ (boundary) to be $\text{cl}(v) \setminus \{v\}$. Note that $\text{bd}(v)$ is sometimes called the *Markov blanket* of v since the blanket of v shields v from any variable that can only indirectly influence v . We define the following family:

$$\mathcal{F}(G, \mathcal{M}^{(l)}) = \{p : X_v \perp\!\!\!\perp X_{V \setminus \text{cl}(v)} | X_{\text{bd}(v)} \text{ in } p \text{ for all } v \in G\} \quad (3.15)$$

This family is defined so that all abiding distributions must be such that a variable is conditionally independent of all else given its Markov blanket.

For example, in G_2 in Figure 3.3, the Markov blanket of x_1 is $\{x_2, x_3, x_4\}$ implying that any abiding distribution must have $X_1 \perp\!\!\!\perp \{X_5, X_6, X_7, X_8, X_9, X_{10}\} | \{X_2, X_3, X_4\}$. The Markov blanket of x_6 is $\{x_4, x_5, x_8\}$ leading to required independence property $X_6 \perp\!\!\!\perp \{X_1, X_2, X_3, X_7, X_9, X_{10}\} | \{X_4, X_5, X_8\}$. In G_3 , the Markov blanket of any node is \emptyset so this means that for any node X_i , and $X_{\neg i}$ indicating all nodes but node i , we have $X_i \perp\!\!\!\perp X_{\neg i} | \emptyset \equiv X_i \perp\!\!\!\perp X_{\neg i}$.

Our next rule is $R^{(p)}$ called the pairwise Markov property. This family may be defined as follows:

$$\mathcal{F}(G, \mathcal{M}^{(p)}) = \{p : X_u \perp\!\!\!\perp X_v | X_{V \setminus \{u, v\}} \text{ in } p \text{ for all non-adjacent pairs } u, v \in V(G)\} \quad (3.16)$$

This property says that if u and v are pairs of nodes in G that are *not* directly connected, then all abiding distributions must be such that the two corresponding random variables are conditionally independent given all variables other than those two. For example, in G_2 in Figure 3.3, x_4 is not directly connected to x_5 so any abiding distribution must have $X_4 \perp\!\!\!\perp X_5 | \{X_1, X_2, X_3, X_6, X_7, X_8, X_9, X_{10}\}$. The same is true for X_2 and X_{10} , or any pair of nodes without an adjoining edge. In G_1 , the two pairs that are not connected are (x_1, x_3) and (x_2, x_4) , so these pairs being conditionally independent given everything else give, resp. the statements $X_1 \perp\!\!\!\perp X_3 | \{X_2, X_4\}$ and $X_2 \perp\!\!\!\perp X_4 | \{X_1, X_3\}$, exactly the same as the global Markov rule.

Note that in the first two of the above families, the family is defined by a form of factorization with respect to the graph. In the latter three families, the families is defined by conditional independence, which is also a form of factorization, with respect to the graph. In all cases, the common aspect of each of these rules is that when edges are missing from the graph, then there is some form of factorization in every member of the corresponding family of distributions. For this reason, it is better to think of *missing edges* as defining independence or factorization properties of a graph that *must hold true*. The edges that are present, on the

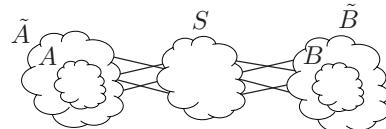
other hand, are not associated with the constraints that define the model — a present edge means only that a factorization or conditional independence property is not required in that case, although it might still be true (recall, the family might contain p 's that have more than what is asked for by the graph G and the set of rules R). An example of this last point is that given any $p \in \mathcal{F}(G_4, \mathcal{M}^{(g)})$, then both $p \in \mathcal{F}(G_1, \mathcal{M}^{(g)})$ and $p \in \mathcal{F}(G_3, \mathcal{M}^{(g)})$.

Our next theorem starts to relate the above families to each other. It is called “alphabetical” because the order of containment is alphabetical in the superscript.

Theorem 14 (The Alphabetical Theorem). *For any undirected graph $G = (V, E)$, the corresponding families have the following inclusion relationship:*

$$\mathcal{F}(G, \mathcal{M}^{(f)}) \subseteq \mathcal{F}(G, \mathcal{M}^{(g)}) \subseteq \mathcal{F}(G, \mathcal{M}^{(l)}) \subseteq \mathcal{F}(G, \mathcal{M}^{(p)}) \quad (3.17)$$

Proof. **1) $\mathcal{F}(G, R^{(f)}) \subseteq \mathcal{F}(G, R^{(g)})$:** Let $A, B, C \subset V(G)$ be a triple of disjoint subsets of $V(G)$ where S separates A from B in G . Since S is a separator, S separates G into a number of connected components. Let \tilde{A} be the nodes of only the connected components that intersect A and \tilde{B} be the nodes of the remaining components (i.e., $\tilde{B} = V \setminus (\tilde{A} \cup S)$). Thus $B \subseteq \tilde{B}$.



Given a maxclique C in G , it is a subset of either $\tilde{A} \cup S$ or $\tilde{B} \cup S$ or both. Note that if it is a subset of both, it must only be due to S , as if C intersected both \tilde{A} and \tilde{B} , S would not be a separator.

Let $\mathcal{C}_{\tilde{A}}$ be the cliques in $\tilde{A} \cup S$ and let $\mathcal{C}_{\tilde{B}} = \mathcal{C} \setminus \mathcal{C}_A$, where \mathcal{C} are the cliques in G .

Consider any $p \in \mathcal{F}(G, R^{(f)})$. Therefore, p may be written:

$$p(x) = \prod_{C \in \mathcal{C}} \psi_C(x) = \prod_{C \in \mathcal{C}_{\tilde{A}}} \psi_C(x) \prod_{C \in \mathcal{C}_{\tilde{B}}} \psi_C(x) \quad (3.18)$$

$$= h(x_{\tilde{A} \cup S}) k(x_{\tilde{B} \cup S}) \quad (3.19)$$

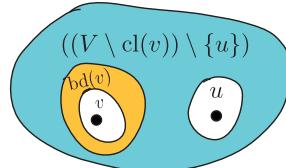
$$\Rightarrow \tilde{A} \perp\!\!\!\perp \tilde{B} | S \text{ by how we chose } p \quad (3.20)$$

$$\Rightarrow A \perp\!\!\!\perp B | S \text{ by C2} \quad (3.21)$$

Note, if \mathcal{C}_B contains no elements from S , then define $k(x_{\tilde{B} \cup S}) = k_1(x_{\tilde{B}})\mathbf{1}(x_S)$. Therefore, $p \in \mathcal{F}(G, R^{(g)})$.

2) $\mathcal{F}(G, R^{(g)}) \subseteq \mathcal{F}(G, R^{(l)})$: Consider any $p \in \mathcal{F}(G, R^{(g)})$, and take any $v \in G$. Since $\text{bd}(v)$ separates v from $V \setminus \text{cl}(v)$ in G , we have that $X_v \perp\!\!\!\perp X_{V \setminus \text{cl}(v)} | X_{\text{bd}(v)}$ in p , and therefore $p \in \mathcal{F}(G, R^{(l)})$.

3) $\mathcal{F}(G, R^{(l)}) \subseteq \mathcal{F}(G, R^{(p)})$: Consider any $p \in \mathcal{F}(G, R^{(l)})$ and pick any $v \in V(G)$ and $u \in V(G) \setminus \text{cl}(v)$. Therefore, $v, u \notin E(G)$ (they are non-adjacent). Also note that $\text{bd}(v) \cup ((V \setminus \text{cl}(v)) \setminus \{u\}) = V \setminus \{u, v\}$.



We thus have that $v \perp\!\!\!\perp V \setminus \text{cl}(v) | \text{bd}(v)$ in p . Since $(V \setminus \text{cl}(v)) \setminus \{u\} \subset V \setminus \text{cl}(v)$, we have by C3 that $v \perp\!\!\!\perp V \setminus \text{cl}(v) | V \setminus \{u, v\}$ in p , and since $u \in V \setminus \text{cl}(v)$, from C2 we have that $v \perp\!\!\!\perp u | V \setminus \{u, v\}$ in p , or that $p \in \mathcal{F}(G, R^{(p)})$.

□

Ideally it would be the case that the above four families are identical. If such was the case, then we would be able to jump between the semantics of each family in order to best obtain an understanding of what the graphical model is saying precise. Unfortunately, it is not always the case that the above four families are equivalent.

Exercise 15. Find an example of a distribution p over 4 variables such that $p \in \mathcal{F}(G, R^{(l)})$ but $p \notin \mathcal{F}(G, R^{(f)})$. Hint, p will need to be not everywhere positive.

Exercise 16. Show that if the general form of C5 is true, then we have $\mathcal{F}(G, R^{(g)}) = \mathcal{F}(G, R^{(l)}) = \mathcal{F}(G, R^{(p)})$

When we restrict the family of distributions to be everywhere positive (such as with a multivariate Gaussian, or any member of the exponential family of distributions where the statistics are defined over only the non-extended real numbers, which does not include $\pm\infty$), then we can show the equivalence of the above four families. This will give us liberty to reason about Markov random fields using any of the three main Markov properties on a graph, and also the factorization property, and switch back and forth between the various views using whatever property is most useful for the current task. This also gives us a deeper understanding of what a given graph means under the MRF semantics.

First we will need the Möbius inversion lemma which is then used to prove the Hammersley-Clifford theorem which states that for positive distributions we have that the above four are equivalent.

Lemma 17. (Möbius Inversion Lemma) Let ψ and ϕ be functions defined on the set of all subsets of a finite set V , taking values in an Abelian group (i.e., a group (closure, associativity, identity, and inverse) for which the elements also commute, the real numbers being just one example). The following two equations imply each other.

$$\forall A \subseteq V : \psi(A) = \sum_{B: B \subseteq A} \phi(B) \quad (3.22)$$

$$\forall A \subseteq V : \phi(A) = \sum_{B: B \subseteq A} (-1)^{|A \setminus B|} \psi(B) \quad (3.23)$$

Proof. We first plug equation 3.23 into the r.h.s. of equation 3.22 and then show that after expansion, we recover $\psi(A)$.

$$\sum_{B: B \subseteq A} \phi(B) = \sum_{B: B \subseteq A} \sum_{C: C \subseteq B} (-1)^{|B \setminus C|} \psi(C) \quad \text{B is sandwiched between } C \subseteq A \text{ and } A \quad (3.24)$$

$$= \sum_{C: C \subseteq A} \sum_{B: C \subseteq B \subseteq A} \psi(C) (-1)^{|B \setminus C|} \quad \text{we rearrange the order of summation} \quad (3.25)$$

$$= \sum_{C: C \subseteq A} \psi(C) \sum_{B: C \subseteq B \subseteq A} (-1)^{|B \setminus C|} \quad (3.26)$$

$$= \sum_{C: C \subseteq A} \psi(C) \sum_{H: H \subseteq A \setminus C} (-1)^{|H|} \quad \text{rename } B \setminus C \text{ to } H \quad (3.27)$$

$$(3.28)$$

The last step follows because the set of sets $\{B \setminus C : C \subseteq B \subseteq A\}$ is identical to the subsets of $A \setminus C$.

Also, note that for some set D ,

$$\sum_{H: H \subseteq D} (-1)^{|H|} = \sum_{i=0}^{|D|} \binom{|D|}{i} (-1)^i = \sum_{i=0}^{|D|} \binom{|D|}{i} (-1)^i (1)^{|D|-i} = (1-1)^{|D|} = \begin{cases} 1 & \text{if } |D| = 0 \\ 0 & \text{otherwise} \end{cases} \quad (3.29)$$

which follows from the binomial expansion. Therefore,

$$\sum_{H:H \subseteq A \setminus C} (-1)^{|H|} = \begin{cases} 1 & \text{if } A = C \\ 0 & \text{otherwise} \end{cases} \quad (3.30)$$

yielding

$$\sum_{B:B \subseteq A} \phi(B) = \sum_{C:C \subseteq A} \psi(C) \mathbf{1}\{A = C\} = \psi(A) \quad (3.31)$$

thus proving the theorem. The other direction is very similar. \square

Exercise 18. Prove the other direction.

It may seem like we have pulled this lemma out of a hat. Indeed, the reason why we have stated this theorem might not be immediately apparent. Rest assured, the theorem is quite general, very useful, and is extremely related to the principle of inclusion-exclusion. This principle, in general, is a property from elementary enumerative combinatorics and occurs when in the process of measuring the size of certain objects (like sizes of sets, or masses of probability), and at the high level states that when we wish to compute such a measure, we may do so by first measuring an overestimate of the quantity (inclusion), then subtract off another quantity (exclusion) yielding an underestimate, and then add on yet another quantity yielding once again an overestimate (exclusion), subtract off still yet another quantity yielding an underestimate (exclusion), and so on. Each quantity that we add or subtract gets smaller and smaller until at some point we converge to the exact quantity we wish to measure. This general idea can be applied in a variety of contexts, but it is perhaps best exemplified in set-theoretic terms, where the goal is to count the size of the union of a set of sets.

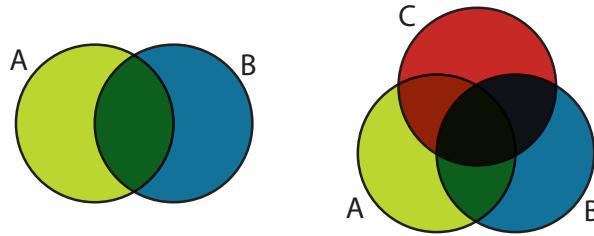


Figure 3.4: Venn Sets.

We all know that if we take two sets $A, B \subseteq V$ which are both subsets of some set V , and we wish to count $|A \cup B|$, we may do this as $|A \cup B| = |A| + |B| - |A \cap B|$. The first two terms $|A| + |B|$ may be an overestimate so we subtract off a correction factor. If there were three such sets $A, B, C \subseteq V$, then $|A \cup B \cup C| = |A| + |B| + |C| - |A \cap B| - |A \cap C| - |B \cap C| + |A \cap B \cap C|$. We start with $|A| + |B| + |C|$ which might be an overestimate, so we correct by subtracting $|A \cap B| - |A \cap C| - |B \cap C|$ which could yield an underestimate, and then add a final potential correction $|A \cap B \cap C|$ yielding the correct amount. Both of these examples are illustrated in Figure 3.4. How does this relate to the lemma above? It is related in that the general form of the inclusion-exclusion formula for set cardinality is as follows. Let A_1, A_2, \dots, A_n be subsets of V . Then we have that:

$$|\cup_{i=1}^n A_i| = \sum_{j=1}^n (-1)^{j-1} \sum_{1 \leq i_1 < i_2 < \dots < i_j \leq n} |A_{i_1} \cap A_{i_2} \cap \dots \cap A_{i_j}| \quad (3.32)$$

Note that the multiplicand on the inner sum is ± 1 and the inner sum is over all values of j variables, depending on what j is as controlled by the outer sum. The notation $\{1 \leq i_1 < i_2 < \dots < i_j \leq n\}$ means

that the j index variables, i_1, i_2, \dots, i_j , range over all j -element subsets of $\{1, 2, \dots, n\}$. This, therefore, directly generalizes the $n = 2$ and $n = 3$ cases we considered above. We note that this is called the sieve formula in combinatorics since we are taking an initial estimate and refining it by repeatedly “filtering” out elements that should not be in the final answer.

It turns out that the inclusion exclusion formula for set cardinality is just a special case of the Möbius inversion lemma.

Exercise 19. Show that this is the case.

Indeed, while the Möbius inversion lemma indeed has many applications, we will use it to prove an important theorem that allows us to equate, in certain cases, the various families of graphical models

Theorem 20. (Hammersley and Clifford) Let \mathcal{F}^+ be the family of distributions with positive (and continuous in the continuous case) density (i.e., $p(x) > 0$ for all $p \in \mathcal{F}^+$). Then $\mathcal{F}^+ \cap \mathcal{F}(G, R^{(f)}) = \mathcal{F}^+ \cap \mathcal{F}(G, R^{(p)})$. In other words, the four above rules define the same family of distributions over a graph G when the distributions are restricted to be everywhere positive.

Proof. We simplify notation by defining $\mathcal{F}^+(G, R) = \mathcal{F}^+ \cap \mathcal{F}(G, R)$ for any rule set R . Given Theorem 14, it suffices to prove only $\mathcal{F}^+(G, R^{(p)}) \subseteq \mathcal{F}^+(G, R^{(f)})$.

Given $p \in \mathcal{F}^+(G, R^{(p)})$. Choose a fixed but arbitrary assignment x^* to the set of random variables X . For all $A \subseteq V$, define

$$h_A(x) = \log p(x_A, x_{A^c}^*) \quad (3.33)$$

where $A^c = V \setminus A$, and where $(x_A, x_{A^c}^*)$ indicates an assignment of values to random vector X where the variables with index set A^c take values coming from x^* and the remaining variables have values from x . Thus, since x^* is a fixed constant, $h_A(x)$ is a function of x only via the values x_A and not via x_{A^c} . Note that positivity ensures that there is never a problem taking logarithms.

For all $A \subseteq V$, define:

$$\phi_A(x) = \sum_{B: B \subseteq A} (-1)^{|A \setminus B|} h_B(x) \quad (3.34)$$

Since $\phi_A(x)$ depends on x only via $h_B(x)$ for $B \subseteq A$, then $\phi_A(x)$ also depends on x only through x_A . Next, apply the Möbius inversion lemma to obtain that

$$\log p(x) = h_V(x) = \sum_{A: A \subseteq V} \phi_A(x). \quad (3.35)$$

While the lemma says that this is also true for all subsets of V , we need only the complete case here. This means that p can be represented as a product of functions of subsets of V . We will be done if we can show that that $\forall x$, $\phi_A(x) = 0$ whenever A is not a clique in G , equivalently whenever there exists non-adjacent vertices $\alpha, \beta \in A$. If we can do this, it will mean that p factorizes w.r.t. the cliques in G (our goal).

Suppose there exists non-adjacent vertices $\alpha, \beta \in A$. Define the set $C = A \setminus \{\alpha, \beta\}$. Then we have:

$$\phi_A(x) = \sum_{B:B \subseteq A} (-1)^{|A \setminus B|} h_B(x) \quad (3.36)$$

$$= \sum_{B:B \subseteq (C \cup \{\alpha, \beta\})} (-1)^{|A \setminus B|} h_B(x) \quad (3.37)$$

$$= \sum_{B:B \subseteq C} (-1)^{|A \setminus B|} h_B(x) + \sum_{B:B \subseteq C} (-1)^{|A \setminus (B \cup \{\alpha\})|} h_{B \cup \{\alpha\}}(x) + \sum_{B:B \subseteq C} (-1)^{|A \setminus (B \cup \{\beta\})|} h_{B \cup \{\beta\}}(x) \quad (3.38)$$

$$+ \sum_{B:B \subseteq C} (-1)^{|A \setminus (B \cup \{\alpha, \beta\})|} h_{B \cup \{\alpha, \beta\}}(x)$$

$$= \sum_{B:B \subseteq C} (-1)^{|A \setminus B|} h_B(x) - \sum_{B:B \subseteq C} (-1)^{|A \setminus B|} h_{B \cup \{\alpha\}}(x) - \sum_{B:B \subseteq C} (-1)^{|A \setminus B|} h_{B \cup \{\beta\}}(x) + \sum_{B:B \subseteq C} (-1)^{|A \setminus B|} h_{B \cup \{\alpha, \beta\}}(x) \quad (3.39)$$

$$= \sum_{B:B \subseteq C} (-1)^{|A \setminus B|} (h_B(x) - h_{B \cup \{\alpha\}}(x) - h_{B \cup \{\beta\}}(x) + h_{B \cup \{\alpha, \beta\}}(x)) \quad (3.40)$$

$$= \sum_{B:B \subseteq C} (-1)^{|C \setminus B|} (h_B(x) - h_{B \cup \{\alpha\}}(x) - h_{B \cup \{\beta\}}(x) + h_{B \cup \{\alpha, \beta\}}(x)) \quad (3.41)$$

So our job is done if we show that the quantity

$$(h_B(x) - h_{B \cup \{\alpha\}}(x) - h_{B \cup \{\beta\}}(x) + h_{B \cup \{\alpha, \beta\}}(x)) \quad (3.42)$$

is zero. We do this as follows. First, for notational simplicity, define $D = V \setminus \{\alpha, \beta\}$. Then the following equations follow where we use positivity and continuity of the distributions:

$$h_{B \cup \{\alpha, \beta\}}(x) - h_{B \cup \{\alpha\}}(x) = \log \frac{p(x_B, x_\alpha, x_\beta, x_{D \setminus B}^*)}{p(x_B, x_\alpha, x_\beta^*, x_{D \setminus B}^*)} \quad (3.43)$$

$$= \log \frac{p(x_\alpha | x_\beta, x_B, x_{D \setminus B}^*) p(x_\beta, x_B, x_{D \setminus B}^*)}{p(x_\alpha | x_\beta^*, x_B, x_{D \setminus B}^*) p(x_\beta^*, x_B, x_{D \setminus B}^*)} \quad x_\beta \text{ and } x_\beta^* \text{ on the l.h.s. can be removed ...} \quad (3.44)$$

$$= \log \frac{p(x_\alpha | x_B, x_{D \setminus B}^*) p(x_\beta, x_B, x_{D \setminus B}^*)}{p(x_\alpha | x_B, x_{D \setminus B}^*) p(x_\beta^*, x_B, x_{D \setminus B}^*)} \quad \dots \text{ by the pairwise Markov property.} \quad (3.45)$$

$$= \log \frac{p(x_\alpha^* | x_B, x_{D \setminus B}^*) p(x_\beta, x_B, x_{D \setminus B}^*)}{p(x_\alpha^* | x_B, x_{D \setminus B}^*) p(x_\beta^*, x_B, x_{D \setminus B}^*)} \quad \text{Since the first ratios is just unity} \quad (3.46)$$

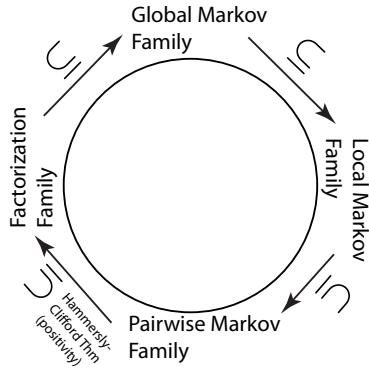
$$= \log \frac{p(x_B, x_\beta, x_\alpha^*, x_{D \setminus B}^*)}{p(x_B, x_\alpha^*, x_\beta^*, x_{D \setminus B}^*)} \quad \text{by pairwise Markov property and chain rule} \quad (3.47)$$

$$= h_{B \cup \{\beta\}}(x) - h_B(x) \quad (3.48)$$

Therefore, when A is not a complete set in G , $\phi_A(x) = 0$ and the distribution factors according to the cliques in the graph, or $p \in \mathcal{F}^+(G, R^{(f)})$. \square

Note that we have not discussed any domain restrictions of the random variables. Suppose that random variable $X_v, v \in V$ takes values on the domain D_{X_v} so that $p(x_v \in D_{X_v}) = 1$. We can form the vector domain as $D_{X_V} = D_{v_1} \times D_{v_2} \times \dots \times D_{v_{|V|}}$ so we have $p(x_V \in D_{X_V}) = 1$. Next, suppose we have subsets $D_{X_v}' \subseteq D_{X_v}$ such that $p(X_v \in D_{X_v}') = 1$, and we form D_{X_V}' accordingly. Note that means that there are zeros in the distribution somewhere, but we are forming the sub-domains to be those random variable values that are not zero, but that contain all the probability (they still sum to one). If D_{X_V}' is such that $p(x_V) > 0$ for all $x_V \in D_{X_V}'$ and if $\sum_{x_V \in D_{X_V}'} p(x_V) = 1$, then we see that even with this domain restriction, the Hammersley-Clifford theorem still holds (as long as we choose values x^* and x from \mathcal{D}'). Thus, we have equivalence of the Markov properties on all sub-domains whenever positivity holds which can be seen as strengthening our result.

The following image summarizes the above results.



We conclude this section by saying that we have defined the primary rules for MRFs and that, in the positive distribution case, they are identical. Therefore, for those distributions, it is possible to jump between rule type with impunity. One may choose which rule set is most applicable for a given purpose, and may freely jump between rules. Moreover, for those distributions that are sparse, we may still assume equivalence for the restricted domains where the random variable values have positive probability.

In the next section, we introduce another type of undirected graphical model.

3.3 Factor Graphs

A factor graph is another type of graphical model that utilizes a different type of graph altogether, and of course a different rule set. A factor graph $G = ((V, F), E)$ is a bipartite graph with two sets of nodes V and F and a set of edges $E \subseteq V \times F$. This means that any two nodes $v_1, v_2 \in V$ or $f_1, f_2 \in F$ are non-adjacent.

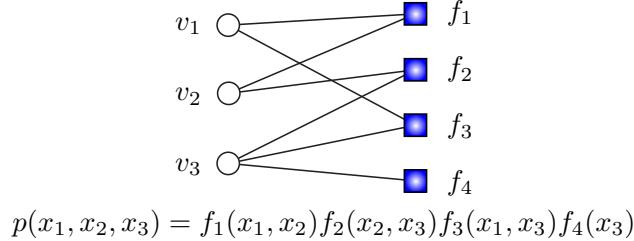
For any $S \subseteq V$, define the neighbors function as $\Gamma(S) = \{f \in F : \exists s \in S \text{ with } (s, v) \in E\}$. Thus, for $v \in V$, $\Gamma(v)$ are v 's neighbors in F . We define the symmetric case as well for any $B \subseteq F$. Thus, for $f \in F$, $\Gamma(f)$ are the neighbors of f in V .

Factor graphs have a different semantics than other graphical models. Here, like before, a set of random variables $\{X_v : v \in V\}$ is identified with the nodes $V(G)$, and a set of factor functions $\{\psi_f : f \in F\}$ is identified with the nodes $F(G)$. Each factor function $f \in F$ operates on the variables corresponding to its neighbors, so we have factors of the form $f(x_{\Gamma(f)})$.

For a given bipartite graph G we define a factor graph, and a set of factor-graph rules $R^{(fg)}$ as the following family of distributions.

$$\mathcal{F}(G, R^{(fg)}) = \left\{ p : \forall f \in F, \exists \psi_f(x_{\Gamma(f)}), \text{ and } p(x_V) = \prod_{f \in F} \psi_f(x_{\Gamma(f)}) \right\} \quad (3.49)$$

In other words, $\mathcal{F}(G, R^{(\text{fg})})$ corresponds to all distributions that can be written as a product of factors, where the factors are formed based on the nodes in F and the factors take arguments based on the factor nodes neighbors. The next figure shows how this is the case.



As can be imagined, this is quite a general form of graphical model, as any p that can be written as a product of factors has a corresponding bipartite graph.

Like in the above UGM case, there are algorithms that can be defined on bipartite graphs that correspond to probability inference. This is not our concern in this chapter, where our goal is to understand the meaning of graphical models in terms of the families they define. As we have seen in the MRF case, if the graph G is one large clique over the nodes V , then all distributions are contained in $\mathcal{F}(G, \mathcal{M}^{(\text{mrf})})$, so there is noting an MRF is unable to include in its family. On the other hand, there may be times that we wish a graphical model to do more than include or not a collection of probability distributions. We instead might be interested in the granularity of such inclusion. That is, suppose we have two distributions p_1 and p_2 which we know can be included in the family of MRFs for some graph. The question we wish now to ask is, are there ways in an MRF (or some other graphical model type) to graphically distinguish between p_1 and p_2 ?

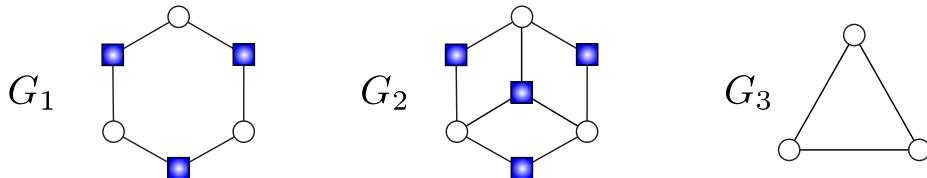
3.4 Generality and Specificity

We have defined a graphical model as a graph $G = (V, E)$ along with a set of rules that carve out, from the space of all distributions over random variables corresponding to $V(G)$, those that obey the rules.

As we will see in future chapters, this is extremely useful since then we can precisely refer to an entire family of models just by considering its graph. Moreover, it will be possible to deduce computational and statistical aspects of all member of a given family just by looking at graph-theoretic properties of the graph, without needing to do any numerical or statistical computation. For example, we will be able to compute probabilistic quantities of interest (which is important for pattern recognition, statistical machine learning, and autonomous decision making) using algorithms that have been derived based only on properties of the graph. Sometimes deriving these algorithms is itself computationally difficult (often times deriving the best algorithm is itself an NP-complete optimization problem). Nonetheless, once the algorithm has been specified, the cost associated with deriving that algorithm is amortized by the fact that the algorithm applies to all probability models in the graph's family for a given set of rules.

There are two aspects of a graphical model as seen as a family that are important to see, but that are perhaps a bit subtle to distinguish. We motivate these aspects with an example.

Consider the following three graphical models, the first two factor graphs and the third a MRF.



On the left, the graph refers to all distributions that can be written as:

$$p_1(x_1, x_2, x_3) = f_1(x_1, x_2)f_2(x_2, x_3)f_3(x_3, x_1) \quad (3.50)$$

The center corresponds to all distributions that can be written as:

$$p_2(x_1, x_2, x_3) = f_1(x_1, x_2)f_2(x_2, x_3)f_3(x_3, x_1)f_4(x_1, x_2, x_3) \quad (3.51)$$

Note that the second family is different in that there may be a factor that allows for the interaction of all three variables simultaneously. For example, following distribution over binary variables

$$\log p(x_1, x_2, x_3) = c + c_{12}x_1x_2 + c_{23}x_2x_3 + c_{13}x_1x_3 + c_{123}x_1x_2x_3 \quad (3.52)$$

can not be written without the tree-way interaction term.

The third figure corresponds to all distributions that can be written:

$$p_3(x_1, x_2, x_3) = \psi(x_1, x_2, x_3) \quad (3.53)$$

Now it is clear that we have $p_1, p_2, p_3 \in \mathcal{F}(G_2, R^{(fg)})$ and that $p_1 \in \mathcal{F}(G_1, R^{(fg)})$ but that $p_2, p_3 \notin \mathcal{F}(G_1, R^{(fg)})$. Moreover, it is clear that $p_1, p_2, p_3 \in \mathcal{F}(G_3, R^{(f)})$.

This suggests that there are two aspects of a type of graphical model that indicate its power, and that is: *generality*, can a graph be found such that its family includes a given distribution; and *specificity*, how discriminative is a given graph that represents a given distribution.

Generality refers to whether or not a given graphical model type can include distributions. For example, if we consider:

$$\mathcal{F}(R^{(f)}) = \bigcup_G \mathcal{F}(G, R^{(f)}) \quad (3.54)$$

then is it the case that all distributions p (over a given set of random variables) is a member of $\mathcal{F}(R^{(fg)})$. Clearly, the answer to this question is yes, since we can always use one large clique (all variables connected) and its family includes all distributions. But perhaps that is not a particularly interesting distribution (it certainly does not have useful properties, such as computational). So clearly it is the case that, at least for MRFs and Factor graphs, we have that:

$$\mathcal{F}(R^{(fg)}) = \mathcal{F}(R^{(f)}) \quad (3.55)$$

So the generality of a type of graphical model R refers to $\mathcal{F}(R)$. It corresponds to:

Definition 21 (Generality). *Given p over $|V|$ random variables, does \exists a G such that $p \in \mathcal{F}(G, R)$.*

We say that p is *covered* by $\mathcal{F}(G, R)$ if it is contained. Moreover, given R_1 and R_2 , if $\mathcal{F}(G, R_1) \subseteq \mathcal{F}(G, R_2)$ we say that R_2 covers R_1 .

A different question, however, might be, for a given p , can we find the simplest G that covers p . Or can we find a G such that its family contains a member that suitably approximates p . This is the structure learning problem in graphical models and will be dealt with later.

Relating to a graphical models semantics, however, there is another issue which we call a family's specificity. I.e., does there exist a graph that allows us to distinguish between two different distributions?

Definition 22 (Specificity). *Given $p_1 \neq p_2$, does \exists a G_1 such that $p_1 \in \mathcal{F}(G_1, R)$ but $p_2 \notin \mathcal{F}(G_1, R)$, and a G_2 such that $p_2 \in \mathcal{F}(G_2, R)$ but $p_1 \notin \mathcal{F}(G_2, R)$*

As we see from the above, in the MRF case, there is no G that allows us to distinguish between p_1, p_2, p_3 but in the factor graph case we can use a graph to specify that we specifically are referring to families such as either p_1 or all three p_1, p_2, p_3 (note that even a factor graph does not allow us to say specifically that there must be a non-zero 3-way interaction term, so we cannot force the family to include only those models that are not like p_1).

This ability of a graphical model to exclude certain models is also part of the power of methodology.

This issue comes up again when we discuss Bayesian networks, which have still different specificity than either MRFs or factor graphs.

Further, they show that probability distributions exist that satisfy the local chain Markov property and violate all conditional independence statements that are not of the form given by (GC)

Another way to think of specificity of a given family is as follows. Do probability distributions exist that abide by all of the properties that must be true of a given family member but also that violate properties that are not specified by the family. Such distributions would still be a member of the family (they violate no rules) but in some sense the existence of such distributions would mean that the family is perhaps not as tight as it could be in that it includes perhaps trivial members that do not respect the nature of the family. For example, a collection of all i.i.d. random variables would trivially be a member of all of the aforementioned families, but such a distribution would violate many of the properties that are left unspecified. We might be interested in another graphical model semantics that specifically precludes those cases.

We might ask the question, is there something special about those members that do not violate properties that are unspecified? Those members are in some sense perfect examples of the family.

3.5 Examples

TODO: Give an example of a MRF as typically used for image processing.

TODO: Give an example of a multivariate Gaussian and independence as specified by the concentration matrix.

TODO: Give an example of a sequential log-linear model as used for language modeling or natural language processing.

Chapter 4

Directed Graphical Models: Bayesian Networks

Now that we have a firm understanding if Markov random fields and factor graphs, we move on to directed graphical models which arguably have a semantics that is a bit more difficult to understand. On the other hand, there are certain distributions that neither MRFs nor Factor graphs can satisfactorily represent, as neither have sufficient specificity.

We define Bayesian networks in detail, and then briefly mention two other forms of directed graphical model, chain graphs and ancestral graphs.

4.1 Bayesian Networks

Like any graphical model, a Bayesian network defines a family of distributions each one of which must obey the rules that define the semantics of the network. In the case of MRFs, the rules generally stated that whenever some form of separation property existed in the graph, a corresponding conditional independence statement or a factorization property must be true of any member in the corresponding family. For Bayesian networks, this is no different, except that the rules for defining factorization in a distribution and separation in a graph are more complicated and involve the graph edge directions.

As will be seen, Bayesian networks correspond to a different family of distributions than do any of the UGMs we have seen so far, and this is particularly acute when one considers the specificity of the models.

We also note, it is important to realize that there is nothing inherently “Bayesian” in Bayesian networks, when one uses the pure statistical use of the term Bayesian. Bayesian statistical models correspond to the case where both random variables and any parameters governing those random variables are random, and there are probability distributions governing those random variables. The type of graphical model called “Bayesian networks” say nothing about this intrinsically. In fact, there are both Bayesian and non-Bayesian Bayesian networks. Some prominent researchers, in fact, would prefer to change the name of Bayesian networks to something more appropriate although other names have their problems as well (e.g., “directed graphical models” are inappropriate since there can be different types of directed graphical model, as we will see). We go with the term “Bayesian network” since that is what has historically been used.

We are given a directed acyclic graph (or DAG) $G = (V, E)$ where V is a set of nodes, and E is a set of *directed edges*. E can be seen as a set of *ordered* pairs of the elements of V , so for each $e \in E$, we have $e = (u, v)$ (equivalently, $u \rightarrow v$). We use the notation (u, v) to indicate an ordered list, where u is the first and v is the second element of the list (u, v) . Each edge $e = (u, v) \in E$ is in the direction from the *parent* variable u to the *child* variable v .

For $e = (u, v)$, we can recover the parent and child using the notation such that $\partial^+(e) = u$ and $\partial^-(e) = v$ — thus, we always have that $e = (\partial^+(e), \partial^-(e))$. We generalize this notation as well, so that

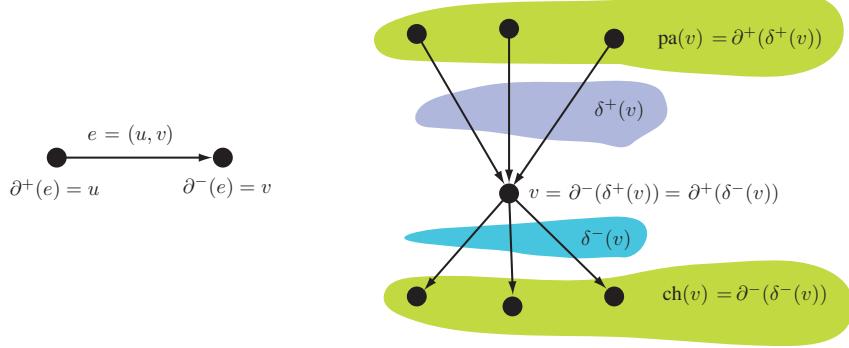


Figure 4.1: Notation in pictures used for directed graphs.

if $F \subseteq E$ is a set of edges, then $\partial^+(F)$ is the set of parents of all edges in F so $\partial^+(F) = \cup_{f \in F} \partial^+(f)$, and $\partial^-(F)$ is the set of children, $\partial^-(F) = \cup_{f \in F} \partial^-(f)$. For $v \in V$, we also use $\delta^-(v)$ to refer to all the outgoing edges starting at v and $\delta^+(v)$ to refer to all the incoming edges into v . We generalize this to sets as well so that we can talk about $\delta^-(U)$ for $U \subseteq V$. Lastly, the parents of a node $v \in V$ can be referred to as $\text{pa}_G(v) = \text{pa}(v) = \partial^+(\delta^+(v))$. The children of a node v is given by $\text{ch}(v)$. Of course, all this is all relative to a given graph G so the middle form is used only when the graph is implicitly but unambiguously given. See Figure 4.1 for examples of this.

Like before, all probability distributions p that correspond to a given DAG $G = (V, E)$ will have as many random variables as there are nodes $|V|$ in G , where each random variable corresponds to a node.

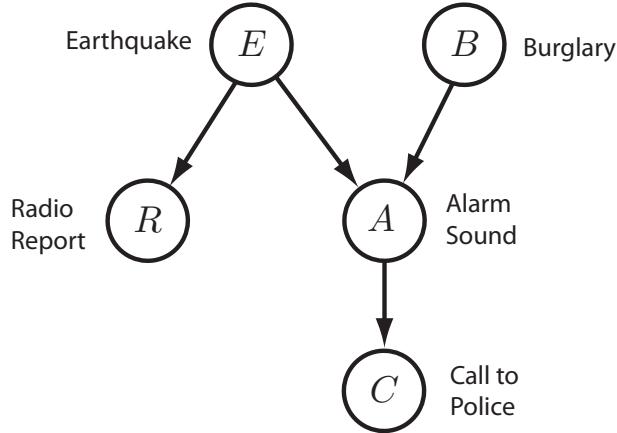


Figure 4.2: The classic example of a Bayesian network describing a situation common in Los Angeles: earthquakes, burglaries, alarms, police calls, and radio reports.

The classic example of a Bayesian network is given in Figure 4.2. There are five binary events that can occur in this domain: an earthquake can occur (E), a burglary (B), a house-alarm can sound (A), a radio report stating that an earthquake has occurred (R), and a call to police as a result of the house alarm (C). This BN describes the following joint distribution:

$$p(B, E, A, C, R) = p(B)p(E)p(A|B, E)p(R|E)p(C|A) \quad (4.1)$$

The situation is as follows. Without knowing the value of any of the variables, we have some nominal “belief” that there could be an earthquake occurring at any given moment $p(E = 1)$ and a belief that there

could be a burglary occurring $p(B = 1)$. If we learn one of the events, it should not have any effect on our belief in the other events. I.e., if we learn that a burglary has occurred (we learn that the event $\{B = 1\}$ has occurred), then this does not affect our belief that an earthquake has occurred, and vice versa. In other words, it is reasonable to assume that these events E and B are independent. Once we learn that a call to the police has occurred stating that the house alarm has went off (the event $\{C = 1\}$ has occurred), our belief in both $\{E = 1\}$ and $B = 1$ increases. That is, we should see that $p(E = 1|C = 1) > p(E = 1)$ and $p(B = 1|C = 1) > p(B = 1)$. More interestingly, however, once $\{C = 1\}$ has occurred, E and B are no longer independent. I.e., if we have learned that the alarm has called the police $\{C = 1\}$ and we subsequently hear a radio report about an earthquake $\{R = 1\}$, then this increases our belief in $\{E = 1\}$ and decreases our belief that a burglary has occurred. It is said that learning about the radio report which increases our belief in $E = 1$ “explains away” the belief that a burglary occurred. I.e., E and B are no longer independent given knowledge of C . A Bayesian network was developed to be able to model such belief augmentations, where “belief” in this case is synonymous with probability.

The above illustrative example shows why we might want models with more causal semantics, but we are still interested in how the Bayesian network defines a family of probability distributions as we next pursue.

Consider the following family defined relative to a DAG G :

$$\mathcal{F}(G, \mathcal{M}^{(\text{df})}) = \left\{ p : p(x) = \prod_{v \in V} p(x_v | x_{\text{pa}(v)}) \right\} \quad (4.2)$$

This is the *directed factorization* (df) property of Bayesian networks, and it states that a Bayesian network corresponds to all distributions that factorize as a product of conditional distributions, one for each child variable given all of its parents.

Before we move on to the semantics of a Bayesian network, we first note that given any ordering of the variables, any probability distribution can be written as a product of distributions conditioned on the previous variables in the order. That is, let $\sigma = (\sigma_1, \sigma_2, \dots, \sigma_N)$ be an ordering of the variables. Then $p(x_1, x_2, \dots, x_N) = p(x_{\sigma_1}, x_{\sigma_2}, \dots, x_{\sigma_N})$ for any permutation order σ and moreover, we have by the definition of conditional probability that:

$$p(x) = p(x_1, x_2, \dots, x_N) = \prod_{i=1}^N p(x_i | x_1, \dots, x_{i-1}) \quad (4.3)$$

$$= \prod_{i=1}^N p(x_{\sigma_i} | x_{\sigma_1}, \dots, x_{\sigma_{i-1}}) \quad (4.4)$$

Now suppose, moreover, that $p \in \mathcal{F}(G, \mathcal{M}^{(\text{df})})$ so that

$$p(x) = \prod_{i=1}^N p(x_i | \text{pa}(x_i)) \quad (4.5)$$

The first thing to note is that we can find some ordering σ so that $\text{pa}(x_i) \subseteq \{x_{\sigma_1}, \dots, x_{\sigma_{i-1}}\}$. This is true because the directed graph is acyclic. Secondly, for this order, the set that is not required, $\{x_{\sigma_1}, \dots, x_{\sigma_{i-1}}\} \setminus \text{pa}(x_i)$, should intuitively seem to be some form of conditional independence property of the family. In particular, it seems that for such an order, we should find that $X_i \perp\!\!\!\perp \{X_{\sigma_1}, \dots, X_{\sigma_{i-1}}\} \setminus \text{pa}(X_i) | \text{pa}(X_i)$. This indeed will be the case, as we will see when we study Markov rules for Bayesian networks.

Note that like the factorization property of MRFs from before, this property defines a family of distributions by a set of factorization properties that must hold for every family member. In the Bayesian network

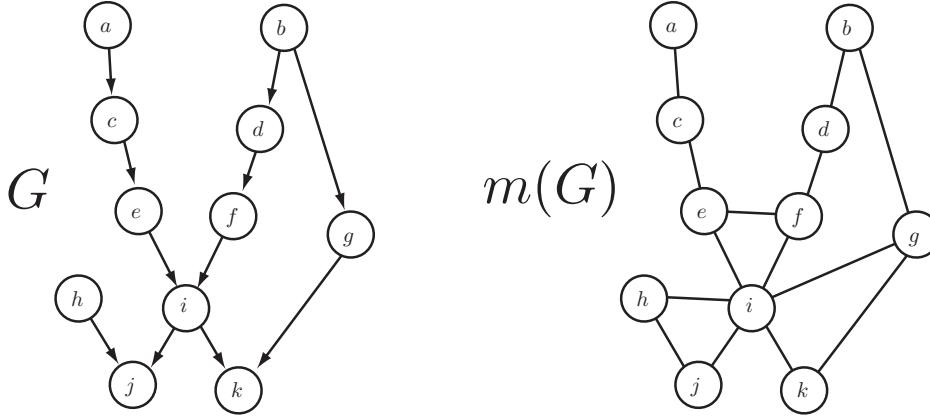


Figure 4.3: Examples of a DAG \$G\$ and its moralized version \$m(G)\$. Note that any factor \$p(v|\text{pa}(v))\$ becomes a clique in the moralized graph so that if \$p\$ is such that directed factorization property holds in the Bayesian network, then \$p\$ is also such that the global Markov property holds in \$m(G)\$.

case, there are \$|V|\$ factors, each one corresponds to a conditional distribution of the child variable given its parents and every variable takes its turn being the child.

Consider now a transformed graph where we form a clique for each child variable and its parents. That is form clique \$C_v\$ for child variable \$v\$ as \$C_v = \{v\} \cup \text{pa}(v)\$ and then form an undirected graph with set of cliques \$\mathcal{C} = \cup_{v \in V} C_v\$. Refer to this undirected graph as \$G_m = (V, E_m)\$ where \$e \in E_m\$ if there is a \$v\$ such that \$e \in C_v\$. Since any conditional distribution of a child \$v\$ given its parents is a valid factor over \$X_{C_v}\$, then for any \$p \in \mathcal{F}(G, R^{(\text{df})})\$ we have that \$p \in \mathcal{F}(G_m, R^{(\text{f})})\$.

Moreover, we see from the alphabetical theorem in the previous chapter that if \$p \in \mathcal{F}(G, R^{(\text{df})})\$ we have that \$p \in \mathcal{F}(G_u, R^{(\text{g})})\$. This means that such a \$p\$ is globally Markov with respect to an undirected graph \$G_m\$ obtained by forming cliques for each child and parent set in the original DAG.

This process, of converting from a DAG to an undirected graphical model, by forming a clique from each directed factor, is called *moralization*. Given a DAG \$G\$, let \$m(G)\$ be its moralization, then the above is a simple proof that:

Theorem 23.

$$\mathcal{F}(G, R^{(\text{df})}) \subseteq \mathcal{F}(m(G), R^{(\text{f})}) \subseteq \mathcal{F}(m(G), R^{(\text{g})}).$$

The reason it is called moralization is that any unconnected (unmarried) parents of a child node are connected with undirected edges (i.e., married, or made moral), and then the directions of any remaining edges are dropped. The example in Figure 4.3 shows a DAG and its moralized version.

Is moralization the smallest elevation from BN to MRF in the sense that there is no smaller MRF family that will completely cover the BN? In other words, we have that \$\mathcal{F}(G, R^{\text{bn}}) \subseteq \mathcal{F}(m(G), R^{\text{mrf}})\$ but is there any other operation, say called \$m'\$ such that \$\mathcal{F}(G, R^{\text{bn}}) \subseteq \mathcal{F}(m'(G), R^{\text{mrf}}) \subset \mathcal{F}(m(G), R^{\text{mrf}})\$? Prove or disprove this statement.

Theorem 23 states an important property. If we start with a DAG \$G\$ and then form \$m(G)\$, then any standard undirected-graph separation property that holds in \$m(G)\$ will require a conditional independence property in any abiding distribution by the global Markov rule. Due to the theorem, this independence property must also hold in the original Bayesian network. Stated in another way, an easy way to read *some* (but not necessarily all) of the independence properties of any \$p \in \mathcal{F}(G, \mathcal{M}^{(\text{df})})\$ is to moralize the graph, and then read off separation properties in \$m(G)\$.

Unlike the global Markov property, however, there are additional constraints placed on the factors in a Bayesian network. Each factor is required to be normalized \$\sum_{x_v} p(x_v | x_{\text{pa}(v)}) = 1\$. This local normalization

property means that there is never any global normalization constant that would need to be computed for the distribution. The local normalization of each factor, moreover, is also one of the things that makes this family of graphical models distinct and gives it some interesting properties.

For example, at least one reason why directed cycles are not allowed under the semantics of Bayesian networks is now clear. Disallowing directed cycles ensures that the above factorization always defines a valid globally normalized probability distribution without the need to compute and/or produce any further normalization. Here, globally normalized means that $\sum_{x_V} p(x_V) = 1$.

Exercise 24 (directed cycle). *Consider the directed 4-cycle graph $A \rightarrow B \rightarrow C \rightarrow D \rightarrow A$. Show that we can obtain a non-normalized global distribution from validly normalized local factors.*

Theorem 25. *Any $p \in \mathcal{F}(G, R^{(df)})$ is globally normalized whenever the directed factors are locally normalized and there are no directed cycles in G .*

Proof. In a DAG, there always exists a node that is not a parent of any other node (if not, we can create a directed chain of nodes, that eventually must lead back to somewhere earlier in the chain thus creating a directed cycle). Choose such a childless node and integrate it out (which we can do since it exists in only one factor as a child and the resulting integration is a factor of unity). Apply this inductively and we get the result. \square

Since all global normalization requires is that each factor is locally normalized with respect to the child variable, each factor may be produced separately of all the others. This has important practical real-world advantages — for example it might be that certain sets of variables and their parents have significantly more training data than others, so each factor can be trained optimally (e.g., each factor can use however much training data is available for the child and its parents, and each factor is free to use whatever implementation it wants, e.g., one factor might choose a probabilistic SVM, another might choose a sparse table, a third might choose a neural network).

In a MRF, training is typically global and it is not as straightforward to train individual clique functions which participate in the overall probability distribution in a non-normalized fashion (note, the EM algorithm could be used for this). The training of each factor in a Bayesian network, moreover, can occur separately (in isolation and in parallel). We will discuss this further in later chapters.

Note that the process of moralizing a graph and moving to the MRF case means that the global Markov property does not utilize the fact that factors are locally normalized. We might therefore expect that some aspect of what a Bayesian network is stating about its distributions is being lost in the conversion to the moralized graph, and this is indeed the case. The aspect that is being lost might be computational or it might just be semantics. Perhaps fortunately, we will see when we discuss decomposable models that no computational aspect is being lost at least when doing exact probabilistic computations, but certain semantic properties are being lost, and this might be important from the perspective of an application domain. Moreover, it has not yet been shown **<< check this >>** that converting a BN to a UGM via moralization will not lose information that could potentially be used for approximate inference.

There are of course other properties that equivalently define the family of distributions that are represented by a Bayesian network. Unsurprisingly, some of these properties define when a conditional independence statement exists based on a form of graph separation. In the Bayesian network case, graph separation is a bit more complicated so we define it carefully here.

In a MRF, we had a very simple notion of graph separation. We can state that notion via what we call blocked or active paths. That is, we say that a set of nodes A is separated in a MRF from another set of nodes B by a third set C in the graph if *all paths* from any node in A to any node in B must go through (or is *blocked* by) some node in C . We stress again that for separation to exist, all paths must be blocked. Any path that is not blocked is said to be *active*. In the MRF case, the notion of blocking is very simple — a path

from a node in A to a node in B is blocked by C if the path contains a node in C . Separation means that *all* paths are blocked.

In a Bayesian network, like in any graphical model, we can also talk about separation in terms of paths and blockage, but we need a new definition of blocked. We define the notion of *directional separation*, or *d-separation*. Since the only graphical difference between a MRF and a BN is that the edges in a BN have arrows, this means that the notion of “blocked” must pay attention to the arrows to determine what paths are blocked and what paths are not. In this case, blocked is a bit more complicated. To distinguish between an MRF and a BN, a blocked path in a Bayesian network will be called *d-blocked*.

Definition 26 (d-blocked). *A path is d-blocked by $C \subset V$ in a Bayesian network if \exists a node v on the path such that either of the following two cases hold:*

Case 1: $v \in C$, and along the path we have at v either serial arrows $\rightarrow v \rightarrow$, or $\leftarrow v \leftarrow$, or divergent arrows $\leftarrow v \rightarrow$

Case 2: We have at v convergent arrows $\rightarrow v \leftarrow$ along the path, and neither v nor any of v 's descendants are in C .

If a path is not d-blocked by $C \subset V$, it is called *d-active*.

Definition 27 (d-Separation). *A set of nodes $A \subset V$ is d-separated from $B \subseteq V$ by $C \subset V$ if all paths from any node in A to any node in B are d-blocked.*

This definition means that if there exists even one path between a node in A and a node in B that is d-active by nodes in C , then d-separation does not hold — *all* paths must be d-blocked for d-separation to hold.

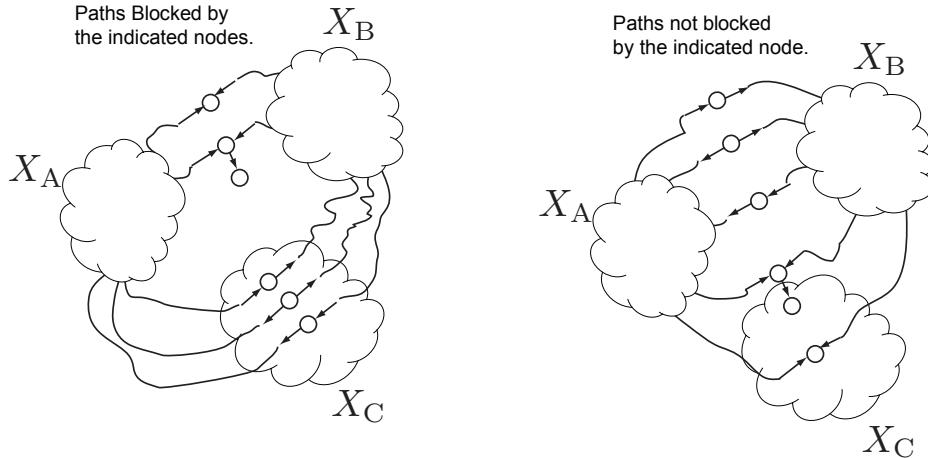


Figure 4.4: Examples of blocked and unblocked paths in d-separation of BNs.

Consider the following family of models:

$$\mathcal{F}(G, R^{(ds)}) = \{p : X_A \perp\!\!\!\perp X_B | X_C \text{ in } p \text{ whenever } C \text{ d-separates } A \text{ and } B \text{ in } G\} \quad (4.6)$$

This family is defined in an analogous way to the global Markov property of MRFs except here d-separation is used rather than the usual (undirected) graph separation.

We next define a few additional families.

Definition 28 (Ancestral set). A set of nodes $A \subseteq V$ in a DAG G is an ancestral set if $\text{pa}(a) \in A$ whenever $a \in A$. If A is an ancestral set, the predicate $\text{Anp}(A)$ is true.

Note that the intersection of ancestral sets is ancestral — if A, B are both ancestral then so is $A \cap B$ since if $v \in (A \cap B)$ then $\text{pa}(v) \subseteq A$ and $\text{pa}(v) \subseteq B$, so $\text{pa}(v) \subseteq (A \cap B)$.

The first family uses the notion of an ancestral set in a Bayesian network. Given a set of nodes $A \subseteq V$, we will want to extend this set by adding to A all parents of nodes in A , and then repeating this process. I.e., consider the following algorithm for constructing $\text{An}(A)$ that repeatedly adds parents to a growing set until there is no change.

Algorithm 1: AncestralHull

Input: A DAG $G = (V, E)$ and set of nodes $A \subseteq V$

Result: $\text{An}(A)$, the ancestral hull of A

A' $\leftarrow A$;

repeat

$A'' \leftarrow A'$;

$A' \leftarrow A'' \cup (\cup_{a \in A'} \text{pa}(a))$;

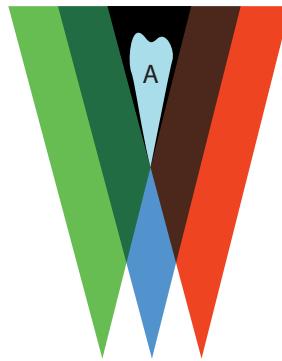
until $A'' = A'$;

return A' ;

The algorithm is clearly going to terminate since it never subtracts nodes at each step and there are only a finite number of nodes. The result of the algorithm $\text{An}(A)$ is called the *ancestral hull* of A , and this is because $\text{An}(A)$ is the smallest ancestral set that contains A , i.e.,

$$\text{An}(A) = \bigcap_{B: A \subseteq B, \text{Anp}(B)} B$$

This yields the smallest ancestral set since we intersect all of them as is shown in the following figure:



In the figure, we have a set A that is not ancestral, and we intersect all containing ancestral sets (which includes the green, blue, red, and all intersections of green, blue, and red) and we are left with the smallest one (black) that contains A .

Exercise 29. Show that the result of the above algorithm and mathematical expression for $\text{An}(A)$ yield identical results.

Notation: recall the notation, that if $G = (V, E)$ is a graph, and $A \subseteq V$ is a subset of nodes, then $G[A] = (A, E')$ is a vertex induced sub-graph, where $E' = E \cap (A \times A)$ is the edges restricted to be adjacent only to nodes A .

It is possible to make a very strong, entirely graphical, statement about the relationship between d-separation and undirected graphs formed by finding ancestral hulls and moralizing the result. We stress that the statement is *graphical*, in that it is a property only of the graph itself, and has no necessary or intrinsic relationship to probability distributions. We will of course use this property so that families of distributions defined in terms of statements made by d-separation and by ancestral hulls will be identical.

Theorem 30. Suppose A, B, C are disjoint subsets of V in a DAG $G = (V, E)$. Then A and B are d-separated by C in G iff C separates A and B in $m(G[\text{An}(A \cup B \cup C)])$.

Proof. **If:** Suppose that C does not d-separate A from B , i.e., there is a d-active path from some node in A to some node in B , meaning all nodes on this path do not cause the path to be d-blocked. We must show that this active path causes there to be an undirected path (u-path) entirely in $m(G[\text{An}(A \cup B \cup C)])$ between A and B but not involving C .

First, we show that all nodes in the (DAG) path must be in $\text{An}(A \cup B \cup C)$ — given a v along this path, we must show that either $v \in A \cup B \cup C$ or that there is a path of directed descendants from v to some node in either A , B , or C . We also use the fact that v does not cause the path to be blocked. If the edges at v are converging, then to not be blocked at v , either $v \in C$ or v has some descendant in C , so v is in the ancestral set. If on the other hand the edges at v are either serial or divergent, then to not be blocked at v , we have that $v \notin C$. However, we may follow a string of diverging edges starting at v along the path until either we arrive at A or B (so that v is in the ancestral set), or we reach a new converging edge node (say at u) in which case, like above, either $u \in C$ or u has descendants in C , thus again making v in the ancestral set. Thus the path is in $\text{An}(A \cup B \cup C)$.

Next, any serial or diverging edge node v along the path was not in C . And any node with converging edges along the path has two parents that are not in C (since each of the parents must be serial or diverging along the path), and due to moralization the two parents are directly connected, thus creating an edge circumventing C . Thus, after moralization, we have a u-path entirely in $m(G[\text{An}(A \cup B \cup C)])$ between A and B but not involving C .

Only-If: Suppose that C does not separate A and B in $m(G[\text{An}(A \cup B \cup C)])$. This means there is an undirected path (u-path) in $m(G[\text{An}(A \cup B \cup C)])$ between some node starting in A , ending at some node in B , and not involving any node in C . We must show that it is possible to find such a u-path that implies the existence of an original d-active path in the DAG.

We first note, by assumption, that if v is in the u-path, then $v \notin C$. Also, all v is in the smallest ancestral set containing A , B , and C , so any node must be either in A , B , C , or be an ancestor of at least one of the three sets. Consider each node v . First, assume adjacent edges are original graph edges, not arising due to moralization. If the edges are if serial or diverging at v , then the corresponding DAG path is not blocked at this point. If the edges are converging at v , then v 's parents are connected (due to moralization) and we may consider a modified u-path that skips v and instead uses the edge directly connecting v 's parents.

Next, consider a node v that is adjacent to some edge that is not part of the original graph, which must be a moralization edge. Let u be the child of the two parents connected by this moralization edge. Note that u is not on the u-path. If $u \in C$ or u has a descendant in C , then a directed path that involves u is not blocked in the DAG at that point. If u and none of u 's descendants are in C , then u must be an ancestor of a node in either A or B or both (since $\text{An}(\cdot)$ is the smallest ancestral set), w.l.o.g. lets say at least A . One of u 's parents steps towards A along the u-path and the other parent steps towards B . We can thus create a new path to u 's descendant in A and connect it to B via the appropriate parent of u to construct a new path, that by-passes this moralization edge, and that is not blocked at this point.

Therefore, a DAG path exists that is not blocked. □

Consider the following family of models:

$$\mathcal{F}(G, R^{(ah)}) = \{p : X_A \perp\!\!\!\perp X_B | X_C \text{ in } p \text{ whenever } C \text{ separates } A \text{ and } B \text{ in } m(G[\text{An}(A \cup B \cup C)])\} \quad (4.7)$$

This family says that if we form the smallest ancestral set containing A, B, C and then moralize that graph, then the global Markov property in the result determines conditional independence for all members of the family.

Corollary 31. *We have that d-separation family and the ancestral hull family are identical. I.e.,*

$$\mathcal{F}(G, R^{(ds)}) = \mathcal{F}(G, R^{(ah)}) \quad (4.8)$$

Figure 4.5 illustrates how we can answer independence questions by looking at the moralized ancestral set.

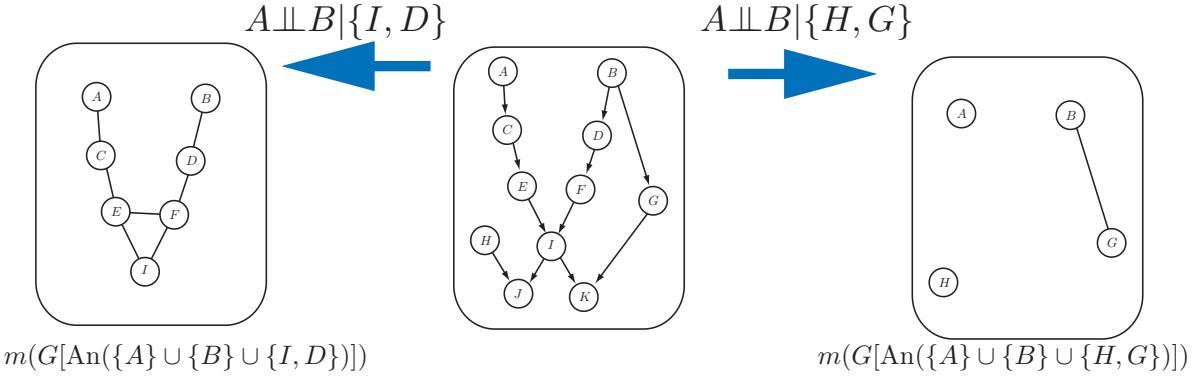


Figure 4.5: Example of a BN (center) and two independence properties required by that BN which are obtained by finding the ancestral set containing the variables in the query, moralizing, and then using the global Markov rule.

Exercise 32. Consider Theorem 23 and also consider Corollary 31. The first case states that we can obtain independence properties required by the BN by moralizing the graph and then using the global Markov rule on the moralized undirected graph. The second case states that, for a given query, we can find the ancestral set and then moralize, and obtain independence properties identical to what is required by the BN.

Come up with an example where an independence property required by the BN is lost when using Theorem 23, but that is not lost when using Theorem 31.

Theorem 33.

$$\mathcal{F}(G, M^{(df)}) \subseteq \mathcal{F}(G, M^{(ah)}) \quad (4.9)$$

Proof. Consider a $p \in \mathcal{F}(G, R^{(df)})$, choose any disjoint $A, B, C \subseteq V$ and let $D = \text{An}(A \cup B \cup C)$. Then we can write

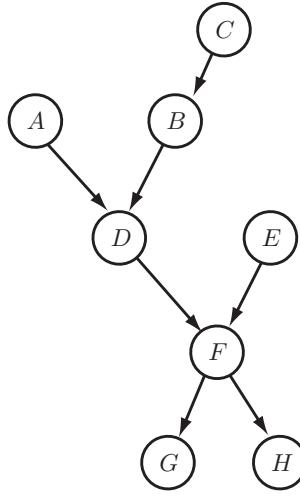
$$p(x) = \prod_{v \in V} p(x_v | x_{\text{pa}(v)}) = \left(\prod_{v \notin D} p(x_v | x_{\text{pa}(v)}) \right) \left(\prod_{v \in D} p(x_v | x_{\text{pa}(v)}) \right)$$

Note that if $v \in D$, then $\text{pa}(v) \in D$, so the second product involves only variables in D , while the first product has no child variables in D . Therefore, any conditional independence property over A, B, C would be determined entirely by the second product in the marginal:

$$p(x_D) = \sum_{x_{V \setminus D}} \prod_{v \in V} p(x_v | x_{\text{pa}(v)}) = \prod_{v \in D} p(x_v | x_{\text{pa}(v)})$$

The right hand side clearly obeys the directed factorization rule on the induced sub-graph $G[D]$ — meaning that $p(x_D) \in \mathcal{F}(G[D], \mathcal{M}^{(df)})$. But then apply Theorem 23 to the right hand side, and we get that $p(x_D) \in \mathcal{F}(m(G[D]), \mathcal{M}^{(g)})$. \square

The next family of models is often used as the definition of a Bayesian network. It states that a variable is independent of its non-descendants given its parents. The non-descendants of a node v , written as $nd(v)$, consists of all nodes that are not-descendants, so this includes both the parents and any other indirect ancestors of v and also any ancestors of any descendants of v that are themselves not descendants of v . In other words, for $v \in V$, $nd(v) = V \setminus (v \cup de(v))$ where $de(v)$ are the descendants of v (see the following figure where $nd(D) = \{A, B, C, E\}$, $nd(E) = \{A, B, C, D\}$, and $nd(F) = \{A, B, C, D, E\}$)



Consider the following family of models:

$$\mathcal{F}(G, R^{(dl)}) = \{p : \forall v \in V(G), X_v \perp\!\!\!\perp X_{nd(v)} | X_{pa(v)} \text{ in } p \text{ where } nd(\cdot) \text{ and } pa(\cdot) \text{ is w.r.t. graph } G\} \quad (4.10)$$

This corresponds to the directed local Markov property (as we will soon see) Bayesian networks.

The following theorem is perhaps the most important defining characteristic of Bayesian networks. Note that this theorem does not require that the corresponding densities are positive.

Theorem 34.

$$\mathcal{F}(G, R^{(df)}) = \mathcal{F}(G, R^{(ah)}) = \mathcal{F}(G, R^{(ds)}) = \mathcal{F}(G, R^{(dl)})$$

Proof. We already know that $\mathcal{F}(G, R^{(df)}) \subseteq \mathcal{F}(G, R^{(ah)})$ (by Theorem 33) and that $\mathcal{F}(G, R^{(ah)}) = \mathcal{F}(G, R^{(ds)})$ (by Theorem 31).

To show that $\mathcal{F}(G, R^{(ah)}) \subseteq \mathcal{F}(G, R^{(dl)})$, consider a $p \in \mathcal{F}(G, R^{(ah)})$. In G , note that for any v , $\{v\} \cup nd(v)$ is an ancestral set since any ancestor of v is in $\{v\} \cup nd(v)$ and any ancestor of any node in $nd(v)$ is not a descendant of v so is also in $v \cup nd(v)$. Also note that $pa(v)$ separate v from $nd(v) \setminus pa(v)$ in $m(G[\{v\} \cup nd(v)])$ yielding

$$v \perp\!\!\!\perp nd(v) \setminus pa(v) | pa(v)$$

in p which, as discussed in Chapter INTRO (properties of independence), is identical to

$$v \perp\!\!\!\perp nd(v) | pa(v)$$

since $pa(v) \subseteq nd(v)$. Therefore, $p \in \mathcal{F}(G, R^{(dl)})$.

To show that $\mathcal{F}(G, R^{(dl)}) \subseteq \mathcal{F}(G, R^{(df)})$, we use induction on the number of nodes. It obviously holds with one node. Consider $p \in \mathcal{F}(G, R^{(dl)})$ over $n + 1$ nodes. Since G is a DAG, we can find a childless node v , so the only non-descendants of v are v 's ancestors. We therefore have that:

$$p(x) = p(x_v | x_{V \setminus v})p(x_{V \setminus v}) = p(x_v | x_{pa(v)})p(x_{V \setminus v})$$

Since $p(x_{V \setminus v})$ is over n nodes, we have that $p(x_{V \setminus v}) \in \mathcal{F}(G[V \setminus v], R^{(df)})$ and with the construction above, we have produced a directed factorization over all $v \in V$, so that $p \in \mathcal{F}(G[V], R^{(df)})$. \square

Therefore, due to the equivalence of the above families, we are free to switch between the corresponding rules when working with Bayesian networks and figuring out what properties are required by, and what assumptions are made by, a given BN. Like in the MRF case, this is useful since it is important from a scientific modeling perspective to ensure that there are not any unintended consequences of a given modeling assumption.

4.1.1 Examples

In this section, we look at a few examples and show how given a graph, the set of independence properties required to be a member of each of the above families are identical.



Figure 4.6: An example of a BN representing a three-variable Markov chain.

In Figure 4.6, we have a Markov chain $X_1 \rightarrow X_2 \rightarrow X_3$. On the left, when $C = \emptyset$, and so nothing d-separates anything else, or no marginal independence properties exist for any $p \in \mathcal{F}(G, \mathcal{M}^{(ds)})$. On the right, X_2 d-separates X_1 and X_3 implying that any $p \in \mathcal{F}(G, \mathcal{M}^{(ds)})$ must have $X_1 \perp\!\!\!\perp X_3 | X_2$. On the other hand, if we take $p \in \mathcal{F}(G, \mathcal{M}^{(df)})$, then $p(x_1, x_2, x_3) = p(x_1)p(x_2|x_1)p(x_3|x_2) = p(x_1, x_2)p(x_3|x_2)$, and

$$p(x_3|x_1, x_2) = \frac{p(x_1, x_2, x_3)}{p(x_1, x_2)} = p(x_3|x_2) \quad (4.11)$$

also requiring $X_1 \perp\!\!\!\perp X_3 | X_2$. The above theorems imply that this is not a coincidence since the families are identical.



Figure 4.7: A BN consisting of arrows diverging from a center node.

In Figure 4.7, we have arrows diverging from X_2 in $X_1 \leftarrow X_2 \rightarrow X_3$. On the left, we have $C = \emptyset$, so nothing d-separates anything else, or there are no marginal independence properties for any $p \in \mathcal{F}(G, \mathcal{M}^{(ds)})$. On the right, X_2 d-separates X_1 and X_3 implying that any $p \in \mathcal{F}(G, \mathcal{M}^{(ds)})$ we must have $X_1 \perp\!\!\!\perp X_3 | X_2$. On the other hand, if we take $p \in \mathcal{F}(G, \mathcal{M}^{(df)})$, then $p(x_1, x_2, x_3) = p(x_2)p(x_1|x_2)p(x_3|x_2) = p(x_1, x_2)p(x_3|x_2)$, and

$$p(x_3|x_1, x_2) = \frac{p(x_1, x_2, x_3)}{p(x_1, x_2)} = p(x_3|x_2) \quad (4.12)$$

requiring $X_1 \perp\!\!\!\perp X_3 | X_2$.

Our third example is in Figure 4.8, where we have arrows converging on X_2 in $X_1 \rightarrow X_2 \leftarrow X_3$. On the left, when $C = \emptyset$, X_1 and X_3 are d-separated, so for any $p \in \mathcal{F}(G, \mathcal{M}^{(ds)})$, $X_1 \perp\!\!\!\perp X_3$. On the right, when



Figure 4.8: A BN consisting of arrows converging onto a center node. This is an example of what is known as a “V-structure.”

$C = \{X_2\}$, nothing d-separates anything else. So it is **not** the case that $X_1 \perp\!\!\!\perp X_3 | X_2$. On the other hand, if we take $p \in \mathcal{F}(G, \mathcal{M}^{(\text{df})})$, then $p(x_1, x_2, x_3) = p(x_1)p(x_3)p(x_2|x_3, x_2)$ and

$$p(x_1, x_3) = \sum_{x_2} p(x_1, x_2, x_3) = p(x_1)p(x_3) \quad (4.13)$$

requiring $X_1 \perp\!\!\!\perp X_3$. The existence of a V-structure is what gave us the unconditional independent but conditionally dependent property of Figure 4.2.

4.1.2 Specificity

In the previous section, we saw that there were examples of factor graphs that excluded certain probability distributions that could not be excluded using MRFs without also excluding many other models that were not excluded from the factor graph.

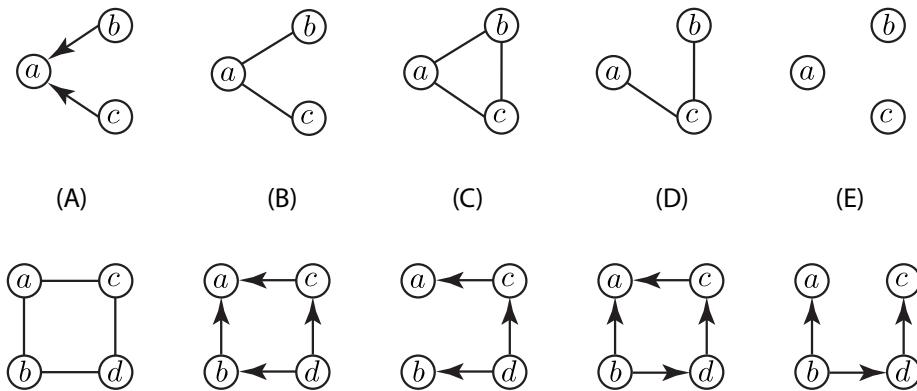


Figure 4.9: Examples of a BN DAG (top-A) and four attempts to find a MRF over three variables that define the same family, and a MRF undirected graph (bottom-A) and four attempts to find a Bayesian network over four variables that define the same family. The example shows that MRFs and Bayesian networks are distinct.

The same kinds of issues come up with Bayesian networks. Consider the BN in Figure 4.9-top-A, which is a simple BN over three variables a, b, c . This is called a *v-structure* and implies only the conditional independence statement

$$a \perp\!\!\!\perp c \quad (4.14)$$

but no other independence statements hold in general. To the right of this BN are four attempts to produce a MRF that can produce exactly the same family. The first case (top-B) fails since while $a \perp\!\!\!\perp c | b$ it is not the case that $a \perp\!\!\!\perp c$. The second case (top-C) expresses no conditional independence relations at all. The third case says only that $a \perp\!\!\!\perp b | c$ but nothing more, and the last case says that all pairs are mutually independent. As can be seen, none of these MRFs therefore define the same family of models as is defined at top-A and therefore, Bayesian networks have a different specificity than do MRFs.

Of course, the tables are turned when trying to find a BN that can match the properties of the MRF 4-cycle. The four-cycle is shown in Figure 4.9 (bottom A). This graph requires that all distributions are such that both

$$a \perp\!\!\!\perp d | \{b, c\} \text{ and } c \perp\!\!\!\perp b | \{a, d\}.$$

An attempt to find a BN that can produce both these independence statements, and nothing more, is shown on the right bottom figures. In the first case (bottom B), we have that $a \perp\!\!\!\perp d | \{b, c\}$ but not that $c \perp\!\!\!\perp b | \{a, d\}$ due to the v-structure $b \rightarrow a \leftarrow c$. An attempt to rectify this problem might be the second case (bottom C), where we now have $c \perp\!\!\!\perp b | \{a, d\}$ but at the cost of adding other properties that were not part of the original 4-cycle, such as $a \perp\!\!\!\perp \{b, d\} | c$ and $\{a, c\} \perp\!\!\!\perp b | d$. We might try switching the direction of the arrow between b and d (an operation that, at this point we do not yet know, will never change the family, see Markov equivalence graphs), as shown in bottom D, but this has the same problems as bottom B. Lastly, we might try bottom E, but that has similar problems to bottom C. Simple enumeration will show that there is no BN over four variables that can produce the same

Exercise 35. Come up with a BN whose properties are not precisely represented by a MRF.

Exercise 36. Come up with a MRF whose properties are not precisely represented by a BN.

Exercise 37. In the following, $G = (V, E)$ is a DAG over $N = |V|$ nodes, and $\sigma = (\sigma_1, \sigma_2, \dots, \sigma_N)$ is some permutation of the indices $(1, 2, \dots, N)$, Consider the following family of models:

$$\begin{aligned} \mathcal{F}(G, \mathcal{M}^{(pm)}) = \{p : X_i \perp\!\!\!\perp \{X_{\sigma_1}, \dots, X_{\sigma_{i-1}}\} \setminus pa(X_i) | pa(X_i) \text{ in } p \\ \text{for any permutation } \sigma \text{ where } \forall i \ pa(x_i) \subseteq \{x_{\sigma_1}, \dots, x_{\sigma_{i-1}}\}\} \end{aligned} \quad (4.15)$$

Show that

$$\mathcal{F}(G, \mathcal{M}^{(pm)}) = \mathcal{F}(G, \mathcal{M}^{(df)}) \quad (4.16)$$

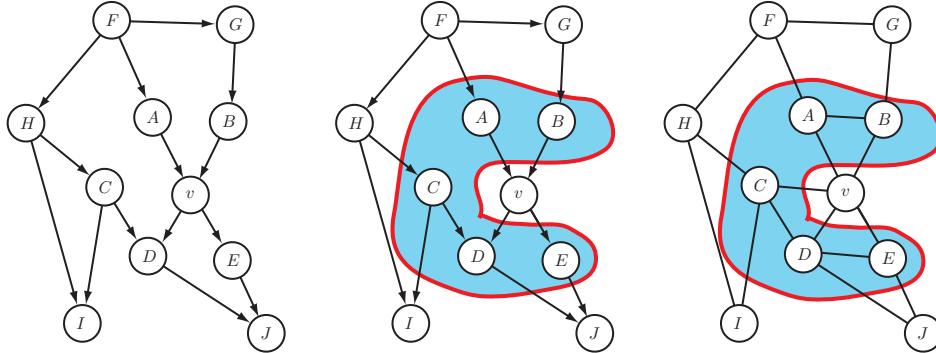


Figure 4.10: Left: An example of a DAG. Middle: the Markov blanket in terms of the BN, where for node $v \in V$, the blanket is the parents, children, and parents of common children of v . Right: The Markov blanket w.r.t. $m(G)$.

Recall the definition of a Markov blanket in the MRF chapter. The essential idea is that the Markov blanket of variable X_v are those other variables that render X_v independent of all remaining variables. In the MRF, the Markov blanket of a node was all of its immediate neighbors. In the BN case, we find the Markov blanket either directly or via the moralized graph. In the latter case, the Markov blanket of a node X_v where $v \in V(G)$ for some DAG G is just the neighbors of X_v in $m(G)$. Speaking directly in terms of the DAG, the blanket of X_v is the parents, children, and parents of common children of X_v . For DAG G , and variable $v \in V$, we'll refer to the Markov blanket as $bl(v)$. So we have

$$bl(v) = pa(v) \cup ch(v) \cup \{w : ch(w) \cap ch(v) \neq \emptyset\} \quad (4.17)$$

Then by looking at $m(G)$ we have the required independence property of any family member:

$$X_v \perp\!\!\!\perp X_{V \setminus \{bl(v) \cup \{v\}\}} | X_{bl(v)} \quad (4.18)$$

Exercise 38. Consider the following family of models:

$$\begin{aligned} \mathcal{F}(G, \mathcal{M}^{(mb)}) = \{p : X_v \perp\!\!\!\perp X_{V \setminus \{bl(v) \cup \{v\}\}} | X_{bl(v)} \text{ where} \\ \forall v \in V, bl(v) \text{ is defined relative to } G.\} \end{aligned} \quad (4.19)$$

What is the relationship between $\mathcal{F}(G, \mathcal{M}^{(mb)})$ and $\mathcal{F}(G, \mathcal{M}^{(df)})$? (i.e., is one family contained in the other, and is that containment proper, or is there another relationship?). Justify your answer.

4.1.3 Continuous Examples

Multivariate Gaussian distribution. Compare to MRF representation of Gaussian distributions. What happens when moving from BN Gaussian to UGM Gaussian and compare with the moralization process. Compare this with Discrete BNs.

4.2 Dependency networks

Some researchers have found that the semantics of Bayesian networks, and d-separation in particular, can be complicated especially when wanting to use Bayesian networks as a graphical representation for visually displaying the variables involved in some applications domain.

Since a BN is directed, it can in some cases be useful to think of the direction of the edges as being related to causality. In some cases, however, one may wish to use a directed model not to express or imply causality, but rather that there is only some predictive dependence between a parents values and its children. In other words, given a variable X_v with $v \in V$, it is such that some other set of variables X_U with $U \subset V$ can predict X_v well, but the semantics should not be one that suggests that X_U necessarily causes X_v .

The goal of dependency networks is to represent predictability (or dependence) without necessarily implying causality. There are two types of dependency networks, those that are necessarily consistent with some underlying joint distribution $p(x)$ and those that are not consistent. We describe each in turn. In both cases, and like in a BN, each node in the graph corresponds to one random variable.

Consistent dependency networks we are given a directed necessarily cyclic graph (so not a DAG) $G = (V, E)$. Like in a Bayesian network, each node X_v is involved in a conditional independence relationship related to its parents, where parents are defined in the same way as for Bayesian networks (i.e., the nodes adjacent to v that point towards but not away from v , again denoted $pa(v)$). The graph prescribes that the following statement must be true for each $v \in V$:

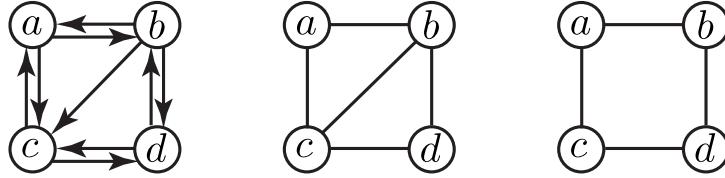
$$p(x_v | x_{V \setminus \{v\}}) = p(x_v | x_{pa(v)}) \quad (4.20)$$

The dependency networks are consistent in that they must be consistent with respect to some underlying joint distribution. I.e., consider the following family:

$$\mathcal{F}(G, R^{(cdn)}) = \left\{ p : \forall v \in V(G), p(x_v | x_{V \setminus \{v\}}) = p(x_v | x_{pa(v)}) \text{ whenever } p(x_v | x_{V \setminus \{v\}}) = \frac{p(x)}{p(x_{V \setminus \{v\}})} \right\} \quad (4.21)$$

Here, we are saying that p is a member of this family if, when we compute the conditional distribution $p(x_v | x_{V \setminus \{v\}})$ from the joint distribution, then that conditional distribution satisfies the conditional independence property given by the graph for each node.

What if some $u \in \text{pa}(v)$ but $v \notin \text{pa}(u)$ in the dependency network? Here is an example. Consider the following dependency network.



From the rules above, we must have that

$$a \perp\!\!\!\perp d | \{b, c\} \text{ and } d \perp\!\!\!\perp a | \{b, c\} \text{ and } b \perp\!\!\!\perp c | \{a, d\}$$

but nothing more. In particular, it does not say that c is independent of anything since $\text{pa}(c) = \{a, b, d\}$. We know, however, from property C1 of conditional independence that the only consistent distribution would also require $c \perp\!\!\!\perp b | \{a, d\}$ this implying that the dependency network is redundant in some way, it does not express its family in the most efficient way, where efficient means fewest numbers of edges.

We define the notion of a *minimal consistent dependency network* graph if for all $v \in V$, we have that there exists no $U \subseteq \text{pa}(v)$ such that

$$p(x_v | x_{\text{pa}(v)}) = p(x_v | x_{\text{pa}(v) \setminus U}) \quad (4.22)$$

In the above example, we see that $p(c|a, b, d) = p(c|a, d)$ so the edge from b to c is redundant thereby making the graph non-minimal.

In fact, it can be shown that for positive distributions, all minimal dependency networks are bi-directional (i.e., for $G = (V, E)$, if $(u, v) \in E$ then $(v, u) \in E$).

Moreover, it is not too difficult to see that the set of distributions in $\mathcal{F}^+(G, R^{(\text{cdn})})$ when G is minimal is exactly the same as those for any of the families of a Markov random field, where the resulting undirected graph is obtained by dropping all edge directions in the dependency network. This follows since we can construct a minimal dependency network from a MRF by creating two opposing edges in the dependency network for all undirected edges. Conversely, we can give an undirected edge to all pairs of nodes that have a bi-directional edge in the dependency network, and the result follows from the local Markov property and the Hammersley-Clifford theorem.

We will leave to the reader to decide if the semantics of a dependency network are simpler and more intuitive than those of a Bayesian network.

4.3 Bayesian Networks and Causality

«« TODO: »» Where does it work? Where does it fail? Briefly describe “Causal BNs” and intervention. This is a huge topic, and there is no way that this section will do it justice. See Pearl’s Causality book (2009 version) for details.

4.4 Bayesian Networks with Constraints

Define BNs but where additional undirected edges may be added between nodes corresponding to factor constraints. What normalization issues come up. How this can be done by a non-pure BN using virtual evidence.

4.5 Ancestral Graphs

« TODO: » Fill in this section.

4.6 Chain Graphs

A type of directed graphical model that combines many of the features of Markov random fields and Bayesian networks is called “Chain graphs”. These are graphs that have both a directed acyclic element and also an undirected element to them. What is interesting is that by combining these two aspects of two different models, one is able to define families that have more specificity than either MRFs or BNs alone.

We start the definition of a Chain graph with a simple modification of a BN. Let us start with a Bayesian network $G = (V, E)$ which is necessarily a DAG. Normally, each node $v \in V$ and each of v 's parents is a single (presumably scalar) random variable. Lets now suppose, however, that associated with each $v \in V$ in a BN is really a collection (or vector) of one or more nodes, lets call it $U(v)$. We also define the union of the collections of nodes associated with v 's parents as $U(\text{pa}(v)) = \cup_{w \in \text{pa}(v)} U(w)$. For each $u \in U(v)$, u may have as parents (denoted as $\text{pa}(u)$) only nodes in $U(\text{pa}(v))$. That is, we have that for each $u \in U(v)$, $\text{pa}(u) \subseteq U(\text{pa}(v))$ and that $\cup_{u \in U(v)} \text{pa}(u) = U(\text{pa}(v))$.

If we define $U = \cup_{v \in V} U(v)$, then this may determine probability distributions over all of the nodes X_U based on factorization properties such as:

$$p(x_U) = \prod_{v \in V} p(x_{U(v)} | U(\text{pa}(v))) \quad (4.23)$$

which will lead to a valid probability distribution since the underling graph is a DAG.

A chain graph doesn't stop there, however, as it includes further graphically-specified structural properties of each of the individual factors. Let us consider a given $v \in V$ and consider a set of nodes $W(v) \triangleq U(v) \cup U(\text{pa}(v))$. For each of these factors, there is a Markov random field $G_v = (W(v), F(v))$ that defines the factorization properties of this factor, where $F(v) \subseteq W(v) \times W(v)$. First, there may be undirected edges between each $u \in U(v) \subseteq W(v)$ and other nodes in $U(v)$, in addition to have the directed edges that define u 's parents — that is, there is a potential sparsity associated with the nodes in $U(v)$ specified in terms of a Markov random field. Other edges in $F(v)$ are defined as follows: any parent of any node $u \in U(v)$ specified by a directed edge becomes an undirected edge in $F(v)$, and any two nodes in $F(v) \setminus U(v)$ are connected. This means that the flexibility of the Markov random field components of the chain graph comes from the undirected edges over the nodes in $U(v)$ and the backbone directed structure of the containing BN.

From the above, we might surmise that there are in fact two nested sets of factorization properties associated with chain graphs, the first associated with the enclosing Bayesian network, and the second associated with the undirected graphs corresponding to each sub-collection. This is indeed the case, as a chain graph family may be defined as follows:

$$\mathcal{F}(G, R^{(\text{cg})}) = \left\{ p : p(x_U) = \prod_{v \in V} p(x_{U(v)} | x_{U(\text{pa}(v))}) \right\} \quad (4.24a)$$

where for each $v \in V$, we may write

$$p(x_{U(v)} | x_{U(\text{pa}(v))}) = \prod_{c \in C(v)} \phi_c(x_c) \quad (4.24b)$$

and where $C(v)$ are the cliques associated with the sub-collection graph $(W(v), F(v))$ with $c \subseteq W(v)$ for each $c \in C(v)$. Like in any MRF, it is assumed that the set of factors normalize to a valid (conditional in

this case) probability distribution and if not, then an additional normalization constant may be given as in the following:

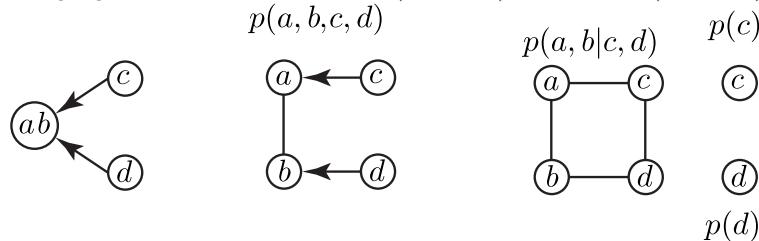
$$p(x_{U(v)} | x_{U(\text{pa}(v))}) = \frac{1}{Z(x_{U(\text{pa}(v))})} \prod_{c \in C(v)} \phi_{C(v)}(x_c) \quad (4.25)$$

so we see that each local factor is a conditional random field (See Section ??). But the key defining characteristic of this family is the set of nested factorization assumptions, the top level based on an underlying DAG, and the next level based on further factorization of each set of nodes and their parents.

We immediately see that any Bayesian network is a chain graph. In such case, $|U(v)| = 1$ for all v (all collections are a single node). Moreover, any Markov random field is also a chain graph — here, $|V| = 1$ and $U(v)$ would comprise all the nodes in the Markov random field. Since there are no parents in this case, there are no restrictions to the set of edges that may be added between nodes in $U(v)$.

The intriguing thing about chain graphs is that they have the potential to provide more specificity than either Bayesian networks and Markov random fields alone. We saw earlier that the family of Bayesian networks could not be represented by Markov random fields or factor graphs, the Markov random field family could not be represented by Bayesian networks, and factor graph family could not be represented by either Bayesian networks or MRFs. Are there chain graphs that represent a family that the other types of graphical models are unable to represent?

Consider the following figure over four variables $p(a, b, c, d)$ with $p \in \mathcal{F}(G, R^{(\text{gc})})$.



On the left is the enclosing DAG, the middle is the actual chain graph G , and on the right are the corresponding MRFs for each of the components. There are four components, one for each node in the DAG. We see that any distribution in this family is such that:

$$p(a, b, c, d) = p(a, b | c, d)p(c)p(d) \quad (4.26)$$

which comes from the underlying DAG. We moreover have a particular factorization property over the conditional distribution, i.e., it must factor according to the cliques in the 4-cycle, so that:

$$p(a, b | c, d) = \phi_{a,b}(a, b)\phi_{b,d}(b, d)\phi_{d,c}(d, c)\phi_{c,a}(c, a). \quad (4.27)$$

Note that we are saying that the conditional distribution $p(a, b | c, d)$ factors in this way, rather than the joint distribution $p(a, b, c, d)$. It might be thought that the factor $\phi_{d,c}(d, c)$ is redundant here but this is not the case since any conditional distribution will need to have some c, d -dependent normalizing constant (often taking the form of $1/Z(c, d)$). Therefore, at the very least we have that

$$\phi_{d,c}(d, c) = \frac{1}{Z(c, d)} \quad (4.28)$$

where $Z(c, d) = \sum_{a,b} \phi_{a,b}(a, b)\phi_{b,d}(b, d)\phi_{d,c}(d, c)\phi_{c,a}(c, a)$.

From this distribution we see that $c \perp\!\!\!\perp d$ since $p(c, d) = \sum_{a,b} p(a, b | c, d)p(c)p(d) = p(c)p(d)$. Moreover, we see that the distribution may be represented as:

$$p(a, b, c, d) = \phi_{a,b}(a, b)\phi_{b,d}(b, d)\phi'_{c,d}(c, d)\phi_{c,a}(c, a) \quad (4.29)$$

where $\phi'_{c,d}(c, d) = \phi_{c,d}(c, d)p(c)p(d)$ which means that the joint distribution factors according to the cliques on the right of the figure. Therefore, we know that

$$a \perp\!\!\!\perp d | \{b, c\} \text{ and } c \perp\!\!\!\perp b | \{a, d\}$$

The question with regards to specificity is, can we find either a Bayesian network, factor graph, or MRF that expresses exactly this set of independence statements (no more and no less)?

We know that the generic 4-cycle over a MRF does not require $c \perp\!\!\!\perp d$. Dropping the edge between c and d in the 4-cycle will require $c \perp\!\!\!\perp d$ but will then require that $a \perp\!\!\!\perp d | \{b\}$ which may not be true in the 4-cycle family. Simple enumeration shows that there is no MRF over 4 variables that express precisely the conditional independence statements required by the above chain graph.

A similar problem arises for BNs and factor graphs. Neither of these types of graphical model may express precisely the requirements to be a member of the chain graph family, meaning that there are instances of distributions contained in the other families that are not contained in the chain graph family. Again, the BN, MRF, or factor graph can of course cover these models, but in doing so loose the specificity that the chain graph offers thereby making chain graphs a unique graphical model semantics.

Exercise 39. Come up with a different chain graph whose properties are not entirely represented by either a BN or a MRF.

What are the uses of chain graphs? As can be seen above, they have more “resolution” than do either MRFs or BNs and therefore can graphically capture the idiosyncrasies of a larger collection of distribution. From the scientific modeling perspective, therefore, when one wishes to express model as a chain of conditional random fields (CRFs), they can be useful. On the other hand, as we will see in future chapters, there is not any inherent computational benefit from representing families of distributions with chain graphs, at least for performing mathematically exact probabilistic quantities (i.e., quantities that do not deal with any potential numerical issues due to the use of finite-precision floating point representations of numbers, which is a distinct from controlled mathematical approximation). In fact, MRFs are sufficient to represent all that is necessary for exact probabilistic quantities, as we will soon see.

4.7 Summary

« « TODO: » » Include summary table of families, where in the text they are defined, and their containment properties. Mention generality/ specificity again, and in general that we use the graph and its graph theoretical properties to proscribe laws that tell us about properties of families of law-abiding probability distributions.

Chapter 5

Evidence in Graphical Models, and Soft and Virtual Evidence

5.1 Chapter Overview

The use of probability to describe a physical process means that one acknowledges, perhaps reluctantly, that complete information about that process is not available and one must resort to some representation of uncertainty. That uncertainty may be inherent to the process itself (i.e., the process is truly random) or it may instead stem from the lack of complete knowledge of that process. Whatever the reason, probability is only one of the possible representations of uncertainty that is available, but it is one that is backed up by a mathematical theory, something that may partially explain probability's appeal.

Probability distributions convey uncertainty about a set of events relevant to the process. For example, for random vector X , rather than expressing a deterministic relationship between X_A and $X_{V \setminus A}$, we consider all possible pairs of values x_A and $x_{V \setminus A}$ and score a given pair's likelihood as $p(x_A, x_{V \setminus A})$.

Any given distribution p expresses uncertainty about a process *a priori*, i.e., given no additional information. It is typically the case, however, that for a specific case new information about that process comes in, and as that new information, certain modifications to p (regarding the remaining events that are not specified) need to be made. In other words, in one way or another, we need to move from an *a priori* expression of uncertainty to an *a posterior* expression of uncertainty.

As new knowledge comes in regarding that process, how is one to integrate that information so as to form a new probabilistic model? New information is often termed “evidence”, and when we have a probability distribution we often discover facts such as $X = 3$ for random variable X . Given such a probability distribution over a set of random variables, how should that probability distribution be updated when information arrives in this form? This begets further questions such as, is this the only form in which information arrives and if not what other useful forms are there? Might the information be a statement of certitude about certain random variables ($X = 3$), and if so what should that certitude imply about the probabilities of that event and the rest of the random variables in the distribution.

More interestingly, this external information that comes in about a process might itself possess uncertainty, and only express preferences amongst certain values of some of the random variables. For example, we may discover that either $X = 2$ or $X = 3$ but $X \neq 1$ and $X \neq 0$. We may furthermore discover that there are certain numeric preferences over the two possibilities $X = 2$ and $X = 3$ and that the numeric preferences might only arrive in the form of non-negative numbers.

The questions raised above has been the topic of “belief” updating, where there is some initial or prior belief (represented by a probability distribution) inherent in a probability model, and when evidence comes in, that belief is updated in some way [336]. We do not wish herein to assume or require any cultural or cognitive notion of what “belief” might mean, and therefore we avoid using this term when possible, using

instead only the more generic terms “probability” or “score”, and we analyze rules for its updating when presented with different forms of evidence. When we do use the term “belief” and its like, we mean it only as a convenient surrogate for these more generic and mathematically agnostic terms.

The predominant form of probability updating that has been in use today has been Bayes’ rule. I.e., we have a distribution $p(x, y)$, then this imparts a marginal probability over y as $p(y) = \sum_x p(x, y)$. If we discover that $X = 3$, then this changes our belief in y from $p(y)$ to

$$p(y|X = 3) = p(y) \frac{p(X = 3|y)}{p(X = 3)}. \quad (5.1)$$

In other words, we update the prior belief $p(y)$ to obtain the posterior belief by multiplying by a factor consisting of the odds ratio $p(X = 3|y)/p(X = 3)$.

On the other hand, what if there is only uncertain information about the event imparted to the variable X ? That is, for each value of X , we might only have some numeric expression regarding how certain we are for each. How should this uncertainty be represented? If the event gives us preferences about different values of x , should this uncertainty be represented as a true distribution over X as in $P(X = x)$? In such case, shouldn’t any resulting joint distribution over $p(x, y)$ thereafter say that the resulting marginals should match this new uncertainty? Are there other notions of uncertainty about an event, one that doesn’t require so rigid an evaluation as this? These are some of the questions we attempt to address.

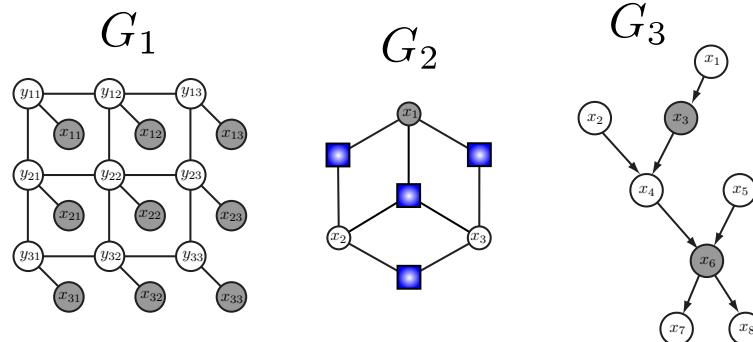
In Pearl’s classic text [336], he defines the notion of virtual evidence and virtual children in a Bayesian network. Virtual evidence is defined as a generalization of standard evidence in Bayesian networks. This simple construct provides a significant increase in the power and representational capabilities of Bayesian networks, and solves some of the questions above quite naturally. Some of these capabilities, however, are already quite naturally encoded in MRF. We define and give an interpretation to such evidence generalizations and uncertainty over evidence, and see how this has been applied. We will see that many statistical models can be described in the Bayesian network framework only with the use of such evidence generalization.

5.2 Traditional Evidence and Zero Probability Events

It is often the case that some subset of the variables with index set $E \subseteq 1 : N$ are “evidence nodes” [336, 207] or equivalently are said to be “observed” (or be a *finding*), meaning that we actually know the values of those random variables. This means that by some process, evidence has arrived, and as part of that evidence we are certain about the values of those random variables. The rest of the variables, those for which evidence is not available, are known as either “hidden”, “unknown”, “unobserved”, or “latent” variables.

The evidence set can be denoted as $X_E = \bar{x}_E$ or simply just \bar{x}_E . All other variables in the network we presumably do not know, and are referred to as *hidden* or *unobserved* variables.

We often display evidence in a graphical model as a shaded variables, as in the following figure.



On the left, we see a MRF with 18 variables, 9 each the forms Y_{ij} and X_{ij} for $1 \leq i, j \leq 3$. The observed variables are $X_{ij} = \hat{x}_{ij}$. In general, we will use lowercase hatted variables (e.g., \hat{x} , \hat{y}) to denote that they are fixed values. The middle case is a factor graph with one variable $X_1 = \hat{x}_1$ observed. The right case is a Bayesian network with $(X_3, X_6) = (\hat{x}_3, \hat{x}_6)$ observed.

Once we have received evidence, the question becomes what does it mean? I.e., how should we interpret the fact that we have discovered that $X_E = \bar{x}_E$ and how should that influence the probabilities of the remaining variables. More importantly, in previous chapters, we defined a graphical model as a family of distributions that obey obligatory rules of membership. One of the questions we wish to address is how does the introduction of evidence change the nature of this family? We do this below in the context of the various forms of potential evidence that might occur.

5.2.1 Evidence as a sample from a statistical process

A typical way to interpret evidence is that it is a partial sample of a draw from the underlying probability distribution. I.e., a complete sample has occurred $\hat{x} \sim p(x)$, but it is not fully specified as $\hat{x}_{V \setminus E}$ is unavailable.

Suppose we are given a data set $\mathbf{D} = \{\bar{x}_E^{(i)}\}_{i=1}^M$ of size M , of samples of (or draws from) some distribution p , so that for each i , $x^{(i)} \sim p(X)$. In each sample, $x_E^{(i)}$ consists of values only of the variables $E \subseteq V$ and some of the values $X_{V \setminus E}$ are missing and unavailable. The subset of values that we are given for each sample is the evidence under that sample. The other random variables, the ones which are not revealed or $X_{V \setminus E}$, are hidden variables, relative to the current sample.

In general, a different sample might reveal a different subset, so the evidence and hidden random variables could be different from sample to sample and E becomes E_i . In practice, however, it is common in such a data set for one fixed set of random variables to consistently be revealed for all samples.

Under the above incomplete-sample interpretation of evidence, it is important to realize that it does not mean that the event has probability one. I.e., if we happen to receive evidence that $X_E = \bar{x}_E$, that does **not** mean that $P(X_E = \bar{x}_E) = 1$. Rather, in this interpretation we need to marginalize away all other non-evidence random variables to discover the probability of the observed event as in

$$p(X_E = \bar{x}_E) = \sum_{x_{V \setminus E}} p(x_{V \setminus E}, \bar{x}_E). \quad (5.2)$$

It might even be the case that we find out that $X_E = \bar{x}_E$, but also that $P(X_E = \bar{x}_E) = 0$, meaning that the event $X_E = \bar{x}_E$ is the impossible event (i.e., an event that receives zero probability). Of course, this can only happen if $p(x)$ is not a strictly positive distribution for all x . While a zero probability event might seem like something that would never occur (and therefore that you would never encounter or need to worry about), in practice, zero probability events might occur in a number of real-world circumstances. The key reason is that the evidence might not come from a sample of the Bayesian network's distribution itself, rather it comes from some other external information source relative to the BN distribution. This might happen for a number of reasons.

First, we must realize that a BN might only be an approximate representation of some underlying true process. By modeling that process stochastically, we are essentially saying that either there is some inherent uncertainty about or within the underlying physical phenomena, or alternatively that there is some peculiarity within the process (e.g., noise) that we wish not, need not, or can not (for computational reasons) model in a detailed way. We resort to treating that peculiarity as randomness. If there was a true random process in the physical world that we wished to model, such a BN representation might or might not correspond to that truth. A zero probability event, therefore, might occur if there is an inaccuracy in the model specification — i.e., a given BN model might lead to the case where we happen to observe something that the model, as it is specified, says will not happen with non-zero probability.

There are a number of reasons why such an inaccuracy might arise:

1. There might be insufficient information about the true model in training data, or the training set size might be small, and regularization alone might not be enough to sufficiently smooth the model so as to remove zero probability events.
2. Even when there is plenty of training data, the training and test data distributions might be different.
3. Many parameter learning procedures are iterative, and during the process of learning we will not have yet produced the ultimate most-accurate model – a zero probability event would mean that the parameters have not yet been properly learned. Moreover, the parameters may never be “properly” learned unless 1) there is sufficient training data and 2) within the model family spanned by the parameter space lies the true model (e.g., the conditions under which maximum likelihood estimation can yield truth).
4. There might be a bug in the specification of the model — a user, when using a software toolkit, might have introduced the bug, or there might be a bug in the toolkit itself.
5. The BN might utilize concise sparse representations of CPTs where low probability events are truncated to zero probability — this can reduce the memory needed to store a model, but might lead to zero probability events. In any case, when this happens the model is said to “score” the evidence with zero probability.

A second reason a set of random variables with zero-probability might occur is as follows: during the computation of probabilistic quantities (discussed in more detail below), we might temporarily encounter assignments to random variables (that are set, say, during a summation or search operation) which end up having zero probability (see Section 5.3.1). These have been referred to “no-goods” in the constraint-satisfaction and SAT literature [108]. For example, given a probability distribution $p(X_1, X_2)$ over two variables X_1 and X_2 , we might wish to perform the sum:

$$p(x_1) = \sum_{x_2} p(x_1, x_2) \quad (5.3)$$

It might be the case that for some particular value pair $\bar{x}_1 \in \mathcal{D}_{X_1}, \bar{x}_2 \in \mathcal{D}_{X_2}$, we have that $p(\bar{x}_1, \bar{x}_2) = 0$. Mathematically, these zero probability events are summed together and, of course, have no effect on the result. But we have encountered zero probability events along the way. If there are many of these zero probability events, summing them naively as implied above is quite wasteful. Of course, in this case the zero probability arises only because of the way a computational strategy is being performed, it says nothing about the way nature provides us with evidence.

In any case, we must be prepared to encounter random variable values that are impossible from the perspective of a current distribution p . Therefore, it might be said that the evidence comes from an sovereign entity, unobligated to the current p .

5.2.2 Evidence as probability revision

A second form of evidence incorporation says that the probability of the event that occurred itself needs to be revised. In this form, we might find out $X_E = \bar{x}_E$ but here we must update the probability model so that $p(X_E = \bar{x}_E) = 1$. This means that the distribution $p(X_V)$ needs to be revised to $p'(X_V)$ so that under p' it agrees with the new event probability. One way to do this is as follows:

$$p'(x_V) = p(x_{V \setminus E} | x_E) \delta(x_E, \bar{x}_E) = p(x_V) \frac{\delta(x_E, \bar{x}_E)}{p(x_E)} \quad (5.4)$$

where $\delta(x_E, \bar{x}_E)$ is one only if $x_E = \bar{x}_E$ and is otherwise zero (see Section 5.3), and where we assume $p(x_E) > 0$.

Unlike evidence as an incomplete sample described in Section 5.2.1, evidence that requires probability revision is best thought as arising from somewhere outside of the current representation.

That is, the finding $X_E = \bar{x}_E$ does not necessarily have anything to do with the current distribution (or its underlying measure). Like in the previous section, p is only an approximate representation of a real physical process, and it assigns probabilities to all possible (zero or non-zero probability) events that can occur x_V . Unlike the previous section, here the evidence can't really be thought of a draw from any probability distribution. Rather, evidence is a specific statement about how the probabilities of a particular event needs to be revised, i.e., so that X_E becomes a constant random variable.

The difference between this form of evidence, and that described in Section 5.2.1 is that here, evidence directly specifies information about the probabilities of some subset of random variables. I.e., the evidence is a direct statement made about the model itself (that we should revise our model so that $p(\bar{x}_E) = 1$ and $p(X_E \neq \bar{x}_E) = 0$). Once we discover this piece of evidence, our probability model should be so adjusted so as to state that \bar{x}_E occurs with certainty.

In Section 5.2.1, however, the evidence has made a statement not about the probabilities but only about the outcomes of a set of random variables. We discover that $X_E = \bar{x}_E$. If it is the case that we receive a data set in which for each sample we have such evidence, it is possible to combine this together to adjust the model (as in any parameter adjustment method such as maximum-likelihood, or some other optimization procedure used as a form of machine learning). But the evidence itself does not dictate how the model probabilities should be updated. With probability revision, utilizing a training data set where every sample has different evidence would be difficult if not impossible. With two samples $x^{(1)}$ and $x^{(2)}$ with $x_E^{(1)} \neq x_E^{(2)}$ then the above equation implies

$$p'(x_V) = p(x_V) \frac{\delta(x_E, \bar{x}_E^{(1)})}{p(x_E)} \frac{\delta(x_E, \bar{x}_E^{(2)})}{p(x_E)} \quad (5.5)$$

yielding $p'(x_V) = 0$ for all x_V in the resulting “trained” model.

We generalize this form of evidence update in Section 5.5.4.1.

5.2.3 Probability of evidence

Next, we introduce three frequently needed calculations including: 1) computing the probability of the evidence (this section); 2) computing (posterior) probabilities of the hidden variables given any evidence; and 3) finding the most likely assignment of (all or a subset of) the hidden variables in a conditional distribution (conditioning on the evidence). In all three cases, we assume evidence of the form described in Section 5.2.1, returning to Section 5.2.2 a bit later.

To compute the probability of the evidence, we simply sum out over all hidden variables and compute:

$$p(\bar{x}_E) = \sum_{x_{V \setminus E}} p(x_{V \setminus E}, \bar{x}_E) \quad (5.6)$$

If we set $H = U \setminus E$, then this notation really means:

$$p(\bar{x}_E) = \sum_{x_{H_1} \in \mathcal{D}_{X_{H_1}}} \sum_{x_{H_2} \in \mathcal{D}_{X_{H_2}}} \dots \sum_{x_{H_{|H|}} \in \mathcal{D}_{X_{H_{|H|}}}} p(x_{V \setminus E}, \bar{x}_E) \quad (5.7)$$

Clearly, this computation if done naively would require a cost of $O(|\mathcal{D}_{X_{V \setminus E}}|)$ operations (exponential in the size of $U \setminus E$). In this discussion, however, we are not concerned with how to perform a given computation, but instead what the computation is in order to understand evidence.

5.2.4 Posterior Probabilities

Second, we might want to calculate the posterior probability of some subset of variables given some other (evidence) subset. Suppose $E \subseteq U$ is a set of variables that we have evidence for. We might very well be interested in the conditional probability $p(x_S | \bar{x}_E)$ where $S \cap E = \emptyset$. As mentioned above one way of viewing this problem is that of probability (or belief) revision. I.e., the BN has an initial set of beliefs over the set of possible values \mathcal{D}_{X_S} of the set random variables X_S , and the belief in $x_S \in \mathcal{D}_{X_S}$ is calculated as $p(x_S) = \sum_{x_{V \setminus S}} p(x_S, x_{V \setminus S})$. Once evidence is introduced, we wish to revise the beliefs in each $x_S \in \mathcal{D}_{X_S}$. A standard way of doing this would be to use Bayes rule, given the posterior belief $p(x_S | \bar{x}_E)$ as above.

If it is the case that $S \cup E = U$, then this means that we must compute the $|\mathcal{D}_{X_S}|$ sized conditional probability table (CPT)

$$p(x_S | \bar{x}_E) = \frac{p(x_S, \bar{x}_E)}{p(\bar{x}_E)} = \frac{p(x_S, \bar{x}_E)}{\sum_{x_S} p(x_S, \bar{x}_E)} \quad (5.8)$$

If on the other hand it is the case that $S \cup E \subset U$, then we have a partition of U into S, E , and H , where $H = U \setminus \{S \cup E\}$, and we must calculate

$$p(x_S | \bar{x}_E) = \frac{\sum_{x_H} p(x_S, x_H, \bar{x}_E)}{p(\bar{x}_E)} = \frac{\sum_{x_H} p(x_S, x_H, \bar{x}_E)}{\sum_{x_S} \sum_{x_H} p(x_S, x_H, \bar{x}_E)}. \quad (5.9)$$

Again, note that $x_V \equiv \{x_S, x_H, x_E\}$. We are often interested in quantities such as

$$p(x_i | \bar{x}_E) = \frac{\sum_{x_{V \setminus \{i\} \cup E}} p(x_{V \setminus E}, \bar{x}_E)}{p(\bar{x}_E)} \quad (5.10)$$

for all $i \in \{U \setminus E\}$. This is the same as above where $S = \{i\}$ for all $i \in U \setminus E$ (so S contains one element and takes turn being all variables in the graph). More generally still, we may partition S into $S = (S_1, S_2, \dots, S_k)$, and then compute $p(x_{S_i} | \bar{x}_E)$ for $i \in \{1, \dots, k\}$. This would be needed, for example, in order to perform EM, gradient-, or max-margin based training of the parameters in the network.

5.2.5 Most likely (Viterbi) values

Third, it is often necessary to find:

$$x_S^* = \operatorname{argmax} x_S p(x_S | \bar{x}_E) = \operatorname{argmax} x_S p(x_S, \bar{x}_E) \quad (5.11)$$

where it is the case that $S \cup E = U$. The algorithm is sometimes called the most-probable explanation (MPE) and can be solved via the *max-product* algorithm (and corresponds to the Viterbi algorithm in HMMs). Again, how to perform this computation is a subject for a different chapter. It should be clear what the desired computation is.

5.3 Evidence as Smart Sums

In the preceding section, we gave the evidence a different name. Specifically, evidence was set as $X_E = \bar{x}_E$ for some subset $E \subseteq U$.

When we consider the three operations in the preceding section, (where we are using sums and or max operations), we can treat the evidence simply as the application of a delta function within the full summation. For example, we have that:

$$p(\bar{x}_i) = \sum_{x_i} p(x_i) \delta(x_i, \bar{x}_i) \quad (5.12)$$

where

$$\delta(x_i, \bar{x}_i) = \begin{cases} 1 & x_i = \bar{x}_i \\ 0 & \text{else} \end{cases} \quad (5.13)$$

is the Kronecker delta function. Similarly

$$p(\bar{x}_i, \bar{x}_j) = \sum_{x_i, x_j} p(x_i, x_j) \delta(x_i, \bar{x}_i) \delta(x_j, \bar{x}_j) \quad (5.14)$$

or generally:

$$p(\bar{x}_S) = \sum_{x_S} p(x_S) \delta(x_S, \bar{x}_S) \quad (5.15)$$

where

$$\delta(x_S, \bar{x}_S) \triangleq \prod_{k \in S} \delta(x_k, \bar{x}_k) \quad (5.16)$$

In some sense, we can bring the delta functions into the sums, so that the sums themselves know about the evidence. If we denote such a “smart sum” for variable x_i with evidence \bar{x}_i as

$$\bar{x}_i \sum_{x_i} f(x_i) \triangleq \sum_{x_i} f(x_i) \delta(x_i, \bar{x}_i) \quad (5.17)$$

or more generally, for evidence \bar{x}_E , we have

$$\bar{x}_E \sum_{x_V} f(x_V) \triangleq \sum_{x_V \in \mathcal{D}_{X_V}} f(x_V) \delta(x_E, \bar{x}_E) = \sum_{x_V \in \mathcal{D}_{X_V}} f(x_{V \setminus E}, x_E) \delta(x_E, \bar{x}_E) = \sum_{x_{V \setminus E} \in \mathcal{D}_{X_{V \setminus E}}} f(x_{V \setminus E}, \bar{x}_E) \quad (5.18)$$

In the left sum, we are essentially summing over all possible values of all variables \mathcal{D}_{X_V} and the ones that do not meet the condition that the subset $x_E = \bar{x}_E$ are annihilated by the delta. On the right most sum, we are only summing over the residual, i.e., the variables that remain free to vary and are not guaranteed to produce a zero value if they don't agree with the evidence.

Of course, it would computationally be better to not sum together many zero values, but again we do not discuss that is beyond the scope of this discussion. Notationally it is very convenient as all evidence variables can be treated like hidden nodes under the use of smart sums or delta functions.

When considering the three problems we mentioned above, it turns out that receiving evidence for a set of random variables is equivalent to using a delta function (or smart summation) with respect to the evidence values. In other words, evidence, in the context of the three problems, can be seen as in some sense an external application of a delta function that annihilates the probability score of all random variable values that do not correspond to the received evidence. Let's now look at each of the three problems in this light.

5.3.1 Computing the probability of evidence

The probability of evidence can now be written by summing over *all* variable assignments in the distribution:

$$p(\bar{x}_E) = \sum_{x_V} p(x_V) \delta(x_E, \bar{x}_E) = \sum_{x_V} p(x_V) \prod_{i \in E} \delta(x_i, \bar{x}_i) = \bar{x}_E \sum_{x_V} p(x_V). \quad (5.19)$$

Any values of variables that do not agree with the evidence are annihilated by the delta function. But for the purposes of computing $p(\bar{x}_E)$ it doesn't matter mathematically if we either: 1) sum over all variable values \mathcal{D}_{X_V} annihilating the ones that do not match the evidence, or 2) sum over only the non-evidence variables keeping the evidence variables fixed in the formula.



Figure 5.1: Simple Virtual Evidence. On the left, vertex X_i is connected to the rest of the BN and is in this case hidden. On the right, vertex X_i has one additional single child V_i , where it is the case that $V_i = 1$ is always true. The child is not connected to any other vertex in the network. The CPT $p(V_i = 1|X_i = x) = \delta(x, \bar{x})$, where \bar{x} is the desired observed value. This construct is therefore a Bayesian network way of expressing evidence in exactly the same way as was done using delta functions in Section 5.3.

5.3.2 Computing the posterior probability of a set of vars

Computing the posterior $p(x_S|\bar{x}_E)$ where $S \subseteq (U \setminus E)$ and $H = U \setminus (E \cup S)$ can also be easily written in this form:

$$p(x_S|\bar{x}_E) = \frac{p(x_S, \bar{x}_E)}{p(\bar{x}_E)} = \frac{\sum_{x_H} p(x_S, x_H, \bar{x}_E)}{\sum_{x_H, x_S} p(x_H, x_S, \bar{x}_E)} = \frac{\sum_{x_H, x_E} p(x_S, x_H, x_E) \delta(x_E, \bar{x}_E)}{\sum_{x_H, x_E, x_S} p(x_H, x_S, x_E) \delta(x_E, \bar{x}_E)} \quad (5.20)$$

$$= \frac{\bar{x}_E \sum_{x_H} p(x_S, x_H, x_E)}{\bar{x}_E \sum_{x_H, x_S} p(x_H, x_S, x_E)} = \frac{\bar{x}_E \sum_{x_H} p(x_V)}{\bar{x}_E \sum_{x_H, x_S} p(x_V)} \quad (5.21)$$

5.3.3 Computing the maximum

Again, the same thing occurs, but we essentially substitute a max for a sum[1] (and we could define a smart max if we wished)

$$x_S^* = \operatorname{argmax}_{x_S} x_S p(x_S, \bar{x}_E) = \operatorname{argmax}_{x_S} x_S \max_{x_E} p(x_S, x_E) \delta(x_E, \bar{x}_E) \quad (5.22)$$

5.4 Scaling Deltas, and Virtual children in a Bayesian network

The above discussion introduces evidence into a model by adding an additional factor, the delta function, that ensures that no value other than the evidence value can occur with no-zero score. This suggests that we can augment a graphical model with additional structure to reflect this additional factor. The way that this is done, of course, depends on the type of graphical model. We start the discussion with Bayesian networks.

Consider computing $p(\bar{x}_i)$. In the last section, we expressed this using delta functions as:

$$p(\bar{x}_i) = \sum_{x_i} p(x_i) \delta(x_i, \bar{x}_i) \quad (5.23)$$

Now suppose that the random variable X_i had an extra single child $V_i \notin X_V$, and that V_i is connected to no other node in the BN other than its parent X_i . Also suppose that V_i is always unity. This does not mean that V_i is a constant random variable as in $p(V_i = 1) = 1$. More precisely, it does not mean that the distribution designates the event $\{V_i = 1\}$ as certain, where the event probability is

$$p(V_i = 1) = \sum_{x_V} p(x_V, V_i = 1) \quad (5.24)$$

which in general could be any value between zero and one.

Note that $V_i \notin \{X_u : u \in U\}$ which means that the variable V_i is not contained in the set of variables comprising our original BN, and thus does not represent a variable that we assume exists within the physical process that we are using the BN to represent. Rather, V_i is a “virtual child”, and it is in the BN only to the extent that it allows us to mathematically model the notion of evidence. Specifically, a virtual child exists for the purposes of constraining its parent to be, with non-zero probability, the given evidence value only (more discussion on this is given in Section 5.5.2).

When each variable $X_i : i \in E$ has its own virtual child V_i , we force V_i 's parent X_i to be a particular value again using a delta function:

$$p(V_i = 1 | X_i = x_i) = \delta(x_i, \bar{x}_i). \quad (5.25)$$

Once $V_i = 1$ is observed, the only event that explains or allows this is the event $X_i = \bar{x}_i$. The CPT needs only a partial specification, however, because $p(v_i | x_i)$ is never fully used — values in the CPT corresponding only to the case $V_i = 1$ are needed (since we always observe $V_i = 1$). In other words, we need $p(v_i | x_i)$ to be deterministic when $v_i = 1$, but it can be anything for $v_i \neq 1$. For example, we have $p(V_i = 1 | X_i = x_i) = \delta(x_i, \bar{x}_i)$ but $p(V_i = j | X_i = x_i)$ for $j \neq 1$ can be arbitrary other than the simplex (i.e., sum to unity) constraint (but see below when we discuss the fact that it is only the relative ratios that matter).

With the above CPTs, the computation of $p(\bar{x}_E)$ is performed as:

$$p(\bar{x}_E) = \sum_x p(x) \prod_{i \in E} p(V_i = 1 | x_i) \quad (5.26)$$

We see that the full distribution $p(V_i = v_i, X_i = x_i)$ is not needed since many of the values are never used. In fact, as we will see below, all that is really needed are the ratios of likelihoods.

5.4.1 Evidence scores other than unity

There is nothing special about the value of unity in the previous section. In fact, we could just as easily have defined the CPT for V_i to be the following:

$$p(V_i = 1 | X_i = x_i) = \alpha \delta(x_i, \bar{x}_i). \quad (5.27)$$

where $\alpha > 0$. The case of $\alpha \neq 1$ has only a very trivial (or even no effect) on the three quantities we are interested in. That is, α does *not* indicate strength of evidence. Note that this is true even if $\alpha > 1$ in which case it no longer can be interpreted as a probability (see Section 5.5).

In the following, we will assume that each evidence node $X_i, i \in E$ is hidden but has a corresponding virtual child $V_i, i \in E$ where each virtual child is observed to always have value $V_i = 1$, and where each V_i CPT implementation has partial specification:

$$p(V_i = 1 | X_i = x_i) = \alpha_i \delta(x_i, \bar{x}_i) \text{ for each } i \in E \quad (5.28)$$

for some fixed set of values \bar{x}_i . Also, define:

$$\alpha = \prod_{i \in E} \alpha_i \quad (5.29)$$

We show that we can substitute α_i with $\alpha'_i = \beta \alpha_i$ for any $\beta > 0$ with impunity.

5.4.1.1 Probability of Evidence

The probability of evidence becomes:

$$p(\bar{x}_E) = \sum_{x_V} p(x_V) \prod_{i \in E} \alpha_i \delta(x_i, \bar{x}_i) = \alpha \sum_{x_V} p(x_V) \delta(x_E, \bar{x}_E) \quad (5.30)$$

Thus, the probability of evidence is just α times the evidence probability computed in Section 5.3.1 and Section 5.2.3 for all possible values of $\bar{x}_E \in \mathcal{D}_{X_E}$.

5.4.1.2 Posterior Probability

The posterior probabilities are identical as long as $\alpha \neq 0$. Using the notation from Section 5.3.2, we have

$$p(x_S | \bar{x}_E) = \frac{p(x_S, \bar{x}_E)}{p(\bar{x}_E)} = \frac{\sum_{x_H, x_E} p(x_S, x_H, x_E) \alpha \delta(x_E, \bar{x}_E)}{\sum_{x_H, x_E, x_S} p(x_H, x_S, x_E) \alpha \delta(x_E, \bar{x}_E)} = \frac{\sum_{x_H, x_E} p(x_S, x_H, x_E) \delta(x_E, \bar{x}_E)}{\sum_{x_H, x_E, x_S} p(x_H, x_S, x_E) \delta(x_E, \bar{x}_E)} \quad (5.31)$$

which is identical to Equation 5.20 in Section 5.3.2. Thus, α has no effect on the posterior probability since it cancels out. Therefore, α (or its constituent α_i values) can not be interpreted as a form of strength of evidence at all. Since there is no gain of generality allowing $\alpha_i < 0$, we assume $\alpha_i > 0$ for each i .

5.4.1.3 Most likely assignments

The set of most likely assignments are also not affected by α_i , as long as $\alpha_i > 0$ for all $i \in E$ (so that $\alpha > 0$). We have:

$$x_S^* = \operatorname{argmax}_{x_S} x_S p(x_S, \bar{x}_E) = \operatorname{argmax}_{x_S} x_S \max_{x_E} p(x_S, x_E) \alpha \delta(x_E, \bar{x}_E) = \operatorname{argmax}_{x_S} x_S \max_{x_E} p(x_S, x_E) \delta(x_E, \bar{x}_E) \quad (5.32)$$

5.5 Generalization of Evidence, Uncertain Evidence, and Virtual Evidence

We now address uncertain evidence which generalizes evidence from the above. There are several forms of uncertain evidence which relate mostly to how numeric scores are associated with different alternate evidence hypotheses [337]. In this section, we describe what has been called *virtual* or *intangible* evidence [336, 337].

The evidence in Section 5.4.1 will henceforth be referred to as *hard evidence*. Recall that the evidence may come from some process external to the current distribution represented by the Bayesian network, and the evidence values in some sense might not at all be associated with the particular current parameter values of the CPTs in the original Bayesian network (i.e., the one without the virtual children defined in Section 5.4).

The effect is that the probability of every variable assignment $X_V = x_V$ is multiplied by one of two possible values, either $\alpha > 0$ or zero depending on if x_V agrees with \bar{x}_E . Suppose that x_E can take on M possible values, so $|\mathcal{D}_{X_E}| = M$. We take values $x_E \in \mathcal{D}_{X_E}$ in lexicographic order such that x_E^j is in the j^{th} position in this order, for $j = 1 \dots M$. Consider a length- M weight vector where the position corresponding to \bar{x}_E has value α and the remaining positions have value 0 such as $(0, 0, \dots, 0, \alpha, 0, \dots, 0)$. With this set of weights, the weight corresponding to any assignment $X_E \neq \bar{x}_E$ is infinitely smaller than the weight corresponding to the assignment $X_E = \bar{x}_E$, because (loosely) $\alpha/\infty = 0$, and this is true for all $\alpha > 0$. This is similar to what we saw earlier where the value of $\alpha > 0$ is irrelevant.

Virtual evidence is when we relax the restriction that the weight values are either finite ($\alpha > 0$ for one assignment \bar{x}_E) or zero (for the remaining assignments). Instead, there is a function $\alpha : \mathcal{D}_{X_E} \rightarrow \mathbb{R}^+$, that provides a non-negative real for every assignment x_E . We can also think of this as a vector of pairs. That is

$$((\bar{x}_E^1, \alpha^1), (\bar{x}_E^2, \alpha^2), \dots, (\bar{x}_E^M, \alpha^M)) \quad (5.33)$$

where \bar{x}_E^i is the i 'th possible assignment to \bar{x}_E and α^i is the corresponding weight.

Virtual evidence may thus encode a form of uncertainty about the event or events that occurred encoded by the α vector. Unlike hard evidence, with virtual evidence more than one event can occur and the weight encodes a preference for these various events to have occurred. It might be said that these weights express relative “degrees of belief” or “degrees of confidence” in those particular assignments of variables but again we do not wish to ascribe either the notions of “belief” or “confidence” to virtual evidence. Just as with hard evidence, the virtual evidence quantities are obtained external to the Bayesian network which might be seen as an incomplete sample from the distribution. We discuss a probabilistic interpretation of this in Section 5.5.2.

As we saw in Section 5.4.1, the function $\alpha(\cdot)$ is relevant only up to a constant of proportionality. The same thing applies in the more general notion of evidence, as we now show.

5.5.1 Why only the ratios matter?

We show here that only the ratios α^i/α^j matter.

First, let us suppose that $|\mathcal{D}_{X_E}| = M > 0$. We take values $x_E \in \mathcal{D}_{X_E}$ in lexicographic order such that $\alpha^j \triangleq \alpha(x_E)$ if x_E is the j^{th} in this order, for $j = 1 \dots M$, where $\alpha^j \geq 0$. We generalize the delta functions above to take on more than 0/1 values, specifically:

$$\delta(x_E; \{(\bar{x}_E^j, \alpha^j)\}_{j=1}^M) = \delta(x_E; (\bar{x}_E^1, \alpha^1), (\bar{x}_E^2, \alpha^2), \dots, (\bar{x}_E^M, \alpha^M)) \triangleq \begin{cases} \alpha^1 & \text{if } x_E = \bar{x}_E^1 \\ \alpha^2 & \text{if } x_E = \bar{x}_E^2 \\ \vdots & \\ \alpha^M & \text{if } x_E = \bar{x}_E^M \end{cases} \quad (5.34)$$

This new delta function (generalizing the delta), applies the value α^j when our evidence variables X_E take on particular value \bar{x}_E^j . We see that $\delta(x_E; \{(\bar{x}_E^j, \alpha^j)\}_{j=1}^M) = \beta \delta(x_E; \{(\bar{x}_E^j, \alpha^j)\}_{j=1}^M)$ when $\alpha'^j = \beta \alpha^j \ \forall j$.

We define a new quantity based on considering

$$\left\{ (\bar{x}_E^j, \alpha^j) \right\} \quad (5.35)$$

as a valid probabilistic event:

$$p(\{(\bar{x}_E^j, \alpha^j)\}_{j=1}^M) \triangleq \sum_{x_V} p(x_V) \delta(x_E; \{(\bar{x}_E^j, \alpha^j)\}_{j=1}^M) \quad (5.36)$$

This quantity generalizes the probability of evidence defined in Section 5.2.3, but here it may be seen as the score of the (uncertain) evidence. It is not entirely correct to call this a probability unless certain constraints are made on the $\alpha(\cdot)$ weights (described in Section 5.5.2 below). We may consider it to be the expected value of the uncertain evidence, however, as we have:

$$p(\{(\bar{x}_E^j, \alpha^j)\}_{j=1}^M) = \sum_{x_V} p(x_V) \delta(x_E; \{(\bar{x}_E^j, \alpha^j)\}_{j=1}^M) \quad (5.37a)$$

$$= \sum_{x_E} p(x_E) \delta(x_E; \{(\bar{x}_E^j, \alpha^j)\}_{j=1}^M) \quad (5.37b)$$

$$= E_p[\delta(x_E; \{(\bar{x}_E^j, \alpha^j)\}_{j=1}^M)] \quad (5.37c)$$

In this view, standard evidence may be seen as the expected value of a delta function since $p(\bar{x}_E) = E_p[\delta(x_E, \bar{x}_E)]$, so that the above generalization to uncertain virtual evidence is quite natural.

Relating to the three procedures given earlier, the values of the vector $(\alpha^1, \alpha^2, \dots, \alpha^M)$ do not really matter up to a non-negative constant. We can just as easily use vector $(\alpha'^1, \alpha'^2, \dots, \alpha'^M)$ where $\alpha'^i = \beta\alpha^i$ for any $\beta > 0$. All that matters for the three procedures is the relative ratios of the various alphas, or α^i/α^j , $i, j \in 1 : M$. We define what we mean by “matters” in the next few sections below. In order to be able to generalize standard evidence, we also allow some of the coefficients to be zero (e.g., $\alpha^j = 0$) in which case the ratios might be zero or infinite, as described earlier. We next examine our three procedures:

5.5.1.1 Score and Probability of virtual evidence

To compute the score of the virtual evidence, we perform the following:

$$p(\{(\bar{x}_E^j, \alpha^j)\}_{j=1}^M) = \sum_{x_V} p(x_V) \delta(x_E; \{(\bar{x}_E^j, \alpha^j)\}_{j=1}^M) = \sum_{j=1}^M \alpha^j \left(\sum_{x_V} p(x_V) \delta(x_E, \bar{x}_E^j) \right) \quad (5.38)$$

$$= \sum_{j=1}^M \alpha^j p(\bar{x}_E^j) \quad (5.39)$$

so the probability of the virtual evidence is simply a weighted sum of the probabilities of each of the individual hard-evidence evidence probabilities, where the weights are just the α_i values. In this form, it is easy to see how if $\alpha^j = \delta(j, \bar{j})$ for a particular value \bar{j} , then we have standard evidence.

There are additional points worth making here as well, namely 1) the α^j values might themselves factorize, 2) constraints may be placed on the α^j values to make the virtual evidence score a probability, and 3) if no constraints are given it is only the ratios that matter.

First, the α^j values might themselves factorize. In other words, we can think of

$$\alpha^j = \prod_{i \in E} \alpha_i^j \quad (5.40)$$

where α_i^j is the weight value applied to the probability of any assignment to X_V whenever $X_i = \bar{x}_i^j$ for $i \in E$. In such case, we can write the virtual evidence score as:

$$p(\{(\bar{x}_E^j, \alpha^j)\}_{j=1}^M) = \sum_{j=1}^M \left(\sum_{x_V} p(x_V) \prod_{i \in E} \alpha_i^j \delta(x_i, \bar{x}_i^j) \right) \quad (5.41)$$

In this case, the evidence variables are free to vary separately with respect to each other in such a way that as long as one of the variables, say $X_i, i \in E$ takes on a particular value $X_i = \bar{x}_i^j$, then the probability of the assignment any X_V with $X_i = \bar{x}_i^j$ will be affected by the factor α_i^j , and this is irrespective of the assignments to any other variables $X_{i'}$ where $i' \in E, i' \neq i$.

On the other hand, it might be desirable to have a weight value jointly for each a pair or a group of evidence variable values. We might, for example, partition the evidence set E into disjoint subsets $E = \{E_1, E_2, \dots, E_L\}$. In this case, we get

$$p(\{(\bar{x}_E^j, \alpha^j)\}_{j=1}^M) = \sum_{j=1}^M \left(\sum_{x_V} p(x_V) \prod_{l=1}^L \alpha_l^j \delta(x_{E_l}, \bar{x}_{E_l}^j) \right) \quad (5.42)$$

In this case, we have a separate value α_l^j for each set of assignments to the subgroup X_{E_l} . Clearly, this approach generalizes and subsumes both Equation 5.38 (when $L = 1$) and Equation 5.41 (when $L = |E|$).

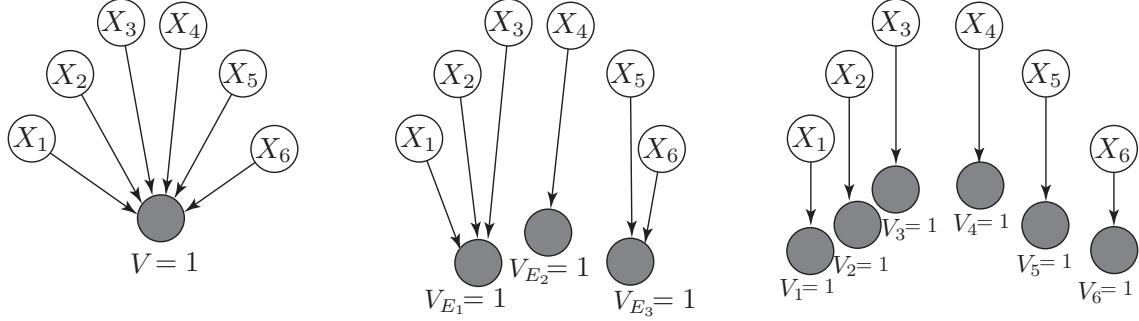


Figure 5.2: Virtual Evidence variants depending on the factorization of the evidence.

In fact, uncertain evidence in this way (and depending on the factorization of α^j) can easily be encoded in a BN, the same way as hard evidence can. Here, any set of evidence nodes X_E that has received virtual evidence possesses a new virtual child node V , outside of the universe, that is always observed to have value 1 (unity), and where we set $p(V = 1|X_E = x_E) = \delta(x_E; \{(\bar{x}_E^j, \alpha^j)\}_{j=1}^M)$. Again, this is only a partial specification of the CPT $p(V|X_E)$ as the values for $p(V \neq 1|X_E = x_E)$ need not be specified since they are never used. Of course, the value $V = 1$ is arbitrary, and it could be any observed value for V as long as the corresponding CPT agrees with that value in how it applies the uncertain evidence. Also, the factorization of α^j is expressed depending on the number of virtual children that exist. If there is one global virtual child $V = 1$, then $\alpha^j = \beta p(V = 1|X_E = \bar{x}_E^j)$ does not (necessarily) factorize, as shown on the left in Figure 5.2, where $\beta > 0$ is a scalar constant of proportionality (which as we will see does not matter). Equation 5.39 becomes

$$p(\{(\bar{x}_E^j, \alpha^j)\}_{j=1}^M) = \beta \sum_{j=1}^M p(V = 1|X_E = \bar{x}_E^j) p(\bar{x}_E^j) \quad (5.43)$$

If we wish to have partial factorization, so that say $\alpha^j = \alpha_{E_1}^j \alpha_{E_2}^j \alpha_{E_3}^j$, we express this in a BN as shown in the center of Figure 5.2. In this case,

$$\alpha^j = \beta_{E_1} p(V_{E_1} = 1|X_{E_1} = \bar{x}_{E_1}^j) \beta_{E_2} p(V_{E_2} = 1|X_{E_2} = \bar{x}_{E_2}^j) \beta_{E_3} p(V_{E_3} = 1|X_{E_3} = \bar{x}_{E_3}^j), \quad (5.44)$$

where again $\beta_i > 0$ are constants of proportionality. Because of the moralization property of Bayesian networks (namely that once moralized, all parents of an observed child have to appear in at least one clique in the junction tree), such factorization of the virtual evidence can have significant computational complexity reductions. If we wish to have full factorization, then we get the right of Figure 5.2. Here, $\alpha^j = \prod_i \beta_i p(V_i = 1|X_i = \bar{x}_i^j)$.

Second, what are the constraints on α^j to make the score a true probability? First, we must note that the model in Equation 5.39 is not a mixture model — i.e., we are not saying that the constraints are such that $\sum_j \alpha^j = 1$ and where $0 \leq \alpha^j \leq 1$. Rather, we require only that $\alpha^j \geq 0$ (or in the factorized case, that $\alpha_i^j \geq 0$ for all i, j). When we view virtual evidence as virtual pendant children, they are connected to nothing except for their parents, we see that the scores are viewed as likelihoods $p(V = 1|x_E = \bar{x}_E)$.

Third, we can see again that assuming $\alpha^j > 0 \forall j$, given fixed ratios of the α^j values, the virtual evidence score is the same up to a constant factor. In other words, if we were to compute the virtual evidence score (Equation 5.39) with two sets of virtual evidence values, $\{\alpha^j\}_j$ and $\{\alpha'^j\}_j$ with $\alpha'^j = \beta \alpha^j$, then the final virtual evidence score would also have the same resulting ratio, i.e.,

$$p(\{(\bar{x}_E^j, \alpha^j)\}_{j=1}^M) = \frac{p(\{(\bar{x}_E^j, \alpha'^j)\}_{j=1}^M)}{\beta}. \quad (5.45)$$

Therefore, we can normalize our virtual evidence score to obtain any desired set of α^j values and only effect the result with a multiplicative constant.

5.5.1.2 Posterior probability of hidden variables

The computation of posterior probabilities (or the “revised belief”) shows more precisely how β does not matter since the posterior probabilities are identical for all $\beta > 0$. Therefore, the effect of the vector is identical up to a constant of proportionality. We have

$$p(x_S | \{(\bar{x}_E^j, \alpha^j)\}_{j=1}^M) = \frac{p(x_S, \{(\bar{x}_E^j, \alpha^j)\}_{j=1}^M)}{p(\{(\bar{x}_E^j, \alpha^j)\}_{j=1}^M)} = \frac{\sum_{x_H, x_E} p(x_S, x_H, x_E) \delta(x_E; \{(\bar{x}_E^j, \alpha^j)\}_{j=1}^M)}{\sum_{x_H, x_E, x_S} p(x_H, x_S, x_E) \delta(x_E; \{(\bar{x}_E^j, \alpha^j)\}_{j=1}^M)} \quad (5.46)$$

$$= \frac{\sum_{j=1}^M \alpha^j p(x_S, \bar{x}_E^j)}{\sum_{j=1}^M \alpha^j p(\bar{x}_E^j)} \quad (5.47)$$

The numerator and denominator in the 2nd equality are both themselves scores. The ratio of the scores is a proper probability since any common constant within the α^j values will cancel out. Therefore, it is clear that the posterior probabilities do not at all depend on the α^j values other than their relative relationships $\alpha^i / \sum_j \alpha^j$, any common factor cancels out. This moreover only depends on the pair-wise ratios, since given α^i / α^j for all i, j we can form

$$\alpha'^j = \frac{1}{\sum_i \alpha^i / \alpha^j} \quad (5.48)$$

which recovers α^j up to a constant of proportionality. This holds also of course for the factorized versions described in the previous section.

5.5.1.3 The Variable Assignment with the Maximum Score: Viterbi

We can compute the Virtual Evidence Viterbi (Viterbi) assignment of the variables as well.

The set of most likely assignments are also only affected by the relative values of $\{\alpha^j\}_j$, as long as $\alpha^j \geq 0$ for all $i \in E$. We have for all $\beta \geq 0$:

$$\begin{aligned} x_S^* &= \operatorname{argmax} x_S p(x_S, \{(\bar{x}_E^j, \alpha^j)\}_{j=1}^M) \\ &= \operatorname{argmax} x_S \max_{x_E} p(x_S, x_E) \delta(x_E; \{(\bar{x}_E^j, \alpha^j)\}_{j=1}^M) \\ &= \operatorname{argmax} x_S \max_{j \in 1:M} p(x_S, \bar{x}_E^j) \alpha^j \\ &= \operatorname{argmax} x_S \max_{j \in 1:M} p(x_S, \bar{x}_E^j) \beta \alpha^j \\ &= \operatorname{argmax} x_S \max_{x_E} p(x_S, x_E) \beta \delta(x_E; \{(\bar{x}_E^j, \alpha^j)\}_{j=1}^M) \end{aligned} \quad (5.49)$$

So the maximum assignment is the same regardless of the actual values of α^j as long as the ratios are identical.

In the above, the result x_S^* is the maximum assignment to X_S with respect to the maximum assignment to the virtual evidence variables X_E . This is the generalization of the Viterbi assignment, since a Viterbi assignment computes the maximum over all hidden variables jointly. We may instead wish to define the best

maximum assignment as:

$$x_S^* = \operatorname{argmax}_{x_E} x_S \sum_{x_E} p(x_S, x_E) \delta(x_E; \{\bar{x}_E^j, \alpha^j\}_{j=1}^M) \quad (5.50)$$

$$= \operatorname{argmax}_{x_S} x_S \sum_{j=1}^M p(x_S, \bar{x}_E^j) \alpha^j \quad (5.51)$$

Equation 5.49 corresponds to the Viterbi approximation to the maximum given by Equation 5.50, which arguably incorporates more of the virtual evidence information. Note that in both cases, again it is only relative values that matter. The second case, however, can in general be much more computationally difficult.

5.5.2 Sample space interpretation of virtual evidence

A graphical model consists of a collection of variables X_V and a family of probability distributions $p(x_V)$ over those variables. How, then, is it possible to allow for the extra virtual children V_i to influence the set of existing random variable via generalized evidence since the random variables and their probabilities are, in some sense, all that there is to be known about the distribution. In this section we answer this question by showing how evidence and virtual versions thereof can be seen as a sample from the underlying probability space.

We assume there exists an underlying probability space (Ω, \mathcal{F}, P) , where Ω is the event space, \mathcal{F} is a σ -field [27] of subsets of Ω , and P is a probability measure defined on \mathcal{F} .

The original set of random variables extracts from Ω set of measurable subsets $\mathcal{F}_X \subseteq \mathcal{F}$, with

$$\mathcal{F}_X \stackrel{\Delta}{=} \{\Omega_{x_A} : x_A \in \mathcal{D}_{X_A}, A \in 2^U \setminus \{\emptyset\}\} \quad (5.52a)$$

where

$$\Omega_{x_A} = \{\omega : X_A(\omega) = x_A\}. \quad (5.52b)$$

That is, \mathcal{F}_X are the set of events that correspond to some specific assignment to some set of random variables in U . If $\mathcal{F}_X \subset \mathcal{F}$, then there are more events in \mathcal{F} than what is carved out by the random variable partitionings (which is typical). Regarding evidence events x_E , we thus have that

$$p(x_E) = P(\Omega_{x_E}). \quad (5.53)$$

Suppose next that there is an event $\eta \in \mathcal{F}$ that has a conditional relationship with (i.e., it is not independent of) the partitioning $\{\Omega_{x_E}\}$ of Ω corresponding to values of X_E ,

$$\bigcup_{x_E \in \mathcal{D}_{X_E}} \Omega_{x_E} = \Omega \quad (5.54)$$

The event η is not independent of the events $\{\Omega_{x_E}\}$ and we can specify the conditioning relationship as:

$$p(\eta | \Omega_{x_E}) \quad (5.55)$$

More precisely, there is a conditional independence assumption (more on this below).

We next identify this event with an extra virtual variable $V = 1$, so that

$$\{V = 1\} \equiv \{\eta\} \quad (5.56)$$

thereby yielding the conditional probability

$$p(\eta | \Omega_{x_E}) = p(V = 1 | X_E = x_E). \quad (5.57)$$

Therefore, these extra virtual children V that we use in the BN merely correspond to probabilistic events in the underlying event space that may or may not have been designated or expressed by \mathcal{F}_X .

Hard (normal) evidence can be easily interpreted in this frame work. Here, we have that

$$\{V = 1\} \equiv \{\eta\} \equiv \{X_E = \bar{x}_E\} \quad (5.58)$$

so that

$$p(\eta|\Omega_{x_E}) = p(V = 1|X_E = x_E) = \delta(x_E, \bar{x}_E). \quad (5.59)$$

In this case, the event η corresponds precisely to $\{V = 1\}$, an event that is already in \mathcal{F}_X . Specifically, we have that $\eta \equiv \{V = 1\} \equiv \{\omega \in \Omega : X_E(w) = \bar{x}_E\} \equiv \{X_E = \bar{x}_E\}$ so that $\eta \in \mathcal{F}_X$. Thus, hard evidence does not require that $\mathcal{F}_X \subset \mathcal{F}$. It perhaps may also be said that this construct is using influence in the conditional probability in the reverse direction. Normally, we assume $p(a|b)$ puts constraints on a given b , but here we are saying that event a has occurred, and what possible way is there to explain that a occurred via b ? This is a feature of all evidence.

On the other hand, it would normally not be the case that the event η would preclude (with zero probability) many other events from occurring and that $p(\eta|x_E)$ would be non-zero for a variety of values of x_E . Suppose that η is such an event with $\eta \in \mathcal{F} \setminus \mathcal{F}_X$, so η does not correspond with any one assignment of any set of random variables $X_A = x_A$, for $A \subseteq U$. The event η (like in the above, equivalently $V = 1$) is conditionally related to a partition of the event space corresponding to \mathcal{D}_{X_E} , yielding

$$p(\eta|\Omega_{x_E}) = p(V = 1|X_E = x_E) \propto \alpha(x_E) \quad (5.60)$$

We already saw in Section 5.5.1 that $\alpha(x_E)$ is required only up to a constant of proportionality, so if we were to start with any vector $(\alpha^1, \alpha^2, \dots, \alpha^M)$, with $\alpha^i > 0, \forall i$, we can normalize using $\max_i \alpha^i$ to produce

$$p(\eta|\Omega_{x_E}) = p(V = 1|X_E = x_E) = \frac{\alpha(x_E)}{\max_{x_E} \alpha(x_E)} \quad (5.61)$$

It is important to realize that in making the above statement, we have made a conditional independence statement about the event η and the rest of the events in the sample space. That is, we assume that

$$\eta \perp\!\!\!\perp \Omega \setminus \Omega_{x_E} | \Omega_{x_E} \quad (5.62)$$

for all values x_E . From the perspective of the corresponding random variables, the assumption is that $V \perp\!\!\!\perp X_{V \setminus E} | X_E$. This is perhaps not surprising, however, as this is precisely the assumption made by the right of Figure 5.1 where $V_i \perp\!\!\!\perp X_{V \setminus \{i\}} | X_i$. Also see Figure 5.2 for an examples of this. The validity of this conditional independence assumption, as does the validity of any conditional independence assumption, depends on the specific application in which evidence is utilized (Section ??).

Thus, we see that evidence and virtual evidence is simply an expanded view into the underlying probability space governing the model.

5.5.3 How can we use $P(V = 1|X_E = \bar{x}_E)$ without having $P(V = 1, X_E = \bar{x}_E)$? A philosophy.

As mentioned in earlier sections, we can think of hard evidence as obtaining partial information about some otherwise unknown sample of the set of random variables X_V described by a BN. In one way or another, this partial information comes from a mechanism external to the BN itself. If the BN reflects truth (meaning the physical object we are representing can perfectly accurately be represented by a BN under a certain parameterization), then evidence can be seen as a sample from the BN where only a subset $X_E \subseteq U$ of the random variables are revealed. If the BN is only an approximate model of truth, then our assumption is

merely that the hidden variables are our best guess as to the process that lead to the observed variables being their current values. In either case, the external evidence says that we should “believe” a sample $X_V = x_V$ infinitely more when the portion $X_E = \bar{x}_E$ than when $X_E \neq \bar{x}_E$.

Virtual evidence is no different, other than the fact that we might obtain a more general form of external information than just hard knowledge of the outcome of some subset of the random variables. Given what we saw about ratios of values above, the safest way to interpret virtual evidence then is to think of it in the following way based on the relative values of the α^j values. That is, the external information we obtain $\{(\bar{x}_E^j, \alpha^j)\}_{j=1}^M$ says only the following:

- There was a sample in the underlying sample space (e.g., η or $V = 1$). We have learned from this source something about a subset X_E of the random variables, where $E \subseteq U$. Other than the effect on X_E , the source has no *direct* influence on the remaining random variables $X_{V \setminus E}$. All other things being equal (i.e., for all possible values of $X_{V \setminus E}$), we have learned that an assignment to X_V with $X_E = \bar{x}_E^j$ should be believed a factor α^j/α^i more an assignment to X_V with $X_E = \bar{x}_E^i$, for all $1 \leq i, j < M$. More specifically, the probability of any assignment to X_V , i.e., $p(X_V = x_V)$ should be multiplied by α^j when $X_E = \bar{x}_E^j$, for $j = 1 \dots M$.

One can give the interpretation of information coming from source external to the BN that provides a form of relative “belief” of different random variable values. The information, however, is only relative, in that it says that one event (i.e., a sub-variable assignment) should be preferred by some constant factor more than another event. This is easily encoded using likelihoods $p(V = 1|X_E = \bar{x}_E^j)$.

Typically, the use of the conditional term $p(V = 1|X_E = \bar{x}_E^j)$ means that we have available the joint distribution $p(X_E = \bar{x}_E^j, V = 1)$ I.e., that the event $\{V = 1\}$ is one that has the same status as any of the other events being modeled \mathcal{F}_X (see Section 5.5.2). The event $V = 1$, however, is special in that we only have available information about it in terms of how it relates conditionally to other variables in our system, namely X_E . That is, the following assumptions are being made:

- As mentioned earlier, $V \perp\!\!\!\perp X_{U \setminus E}|X_E$, meaning that V only indirectly effects variables other than X_E . X_E renders the event V independent of the rest of the model $X_{U \setminus E}$.
- Information in how V effects the model only is available in terms of $P(V = 1|X_E)$, and this is an atomic object (i.e., the virtual evidence is not amenable to deconstruction in terms of Bayes rule to obtain $p(X_E|V = 1)p(V = 1)/p(X_E)$). This means that the joint distribution over the entire model $P(X_V, V)$ is not available.

There are several ways this might arise in practice. First, many statistical models inherently represent only the conditional rather than the joint distribution. For example, discriminative models such as multi-layered perceptrons [57], support-vector machines [428] (when endowed with a distribution), and various conditional maximum entropy models [342] represent only the construct $p(A|B)$ without ever needing any representation of the joint distribution $p(A, B)$. Given observations consisting of both events of the form $V = 1$ and $V = 0$ (for a binary V variable), it would be possible to learn the distribution $p(V = 1|X_E)$ and apply it to the model as above.

Second, there are scenarios where we can reason only about ratios of likelihoods $p(V = 1|\bar{x}_E^j)/p(V = 1|\bar{x}_E^i)$, or more generally pairwise preferences, rather than the absolute values of the likelihoods themselves. Since only ratios determine the effect, we are comfortable when this occurs.

We take another example from Pearl’s text ([336]), where we have BN with four variables, B for burglary, S for alarm sound, W for Watson’s testimony, and G for Gibbon’s testimony. The network so given is such that the following factorization holds (Figure 2.1 on page 43 in [336], reproduced in Figure 5.3 here.).

$$p(H, S, W, G) = p(H)p(S|H)p(G|S)p(W|S) \quad (5.63)$$

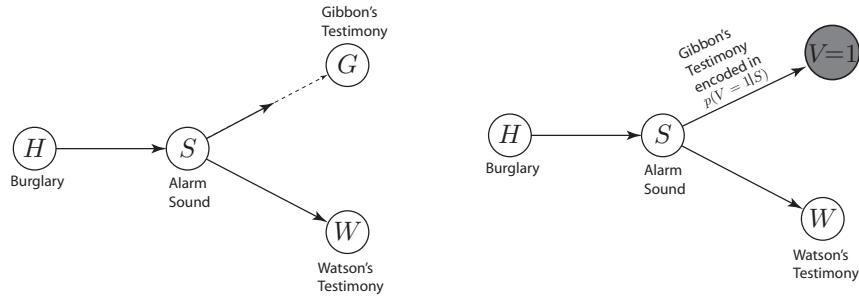


Figure 5.3: Left: Figure 2.1 from page 43 of Pearl [336]. Right: An interpretation in terms of virtual evidence via the likelihood $p(V = 1|S)$

The ultimate goal is to compute the probability of H (a burglary occurred) given knowledge about any of the following: the alarm went off $S = 1$, Watson's testimony W , and/or Gibbon's testimony G . In the example (section 2.2.2 of [336]), however, it is not known if the alarm went off, so that the variable S is hidden. The only thing that is known is perhaps some information about G and or W .

Virtual evidence arises in [336] when it is not known how precisely the value of G effects the remainder of the network, but rather where only some aspect of G provides us with ratios of preferences about H . For example, from G we are able only to ascertain that it is four times more likely that the alarm went off ($S = 1$) than otherwise ($S = 0$). In other words, any probability of the joint set of random variables $p(H, S, W)$ where $S = 1$ (alarm) should be multiplied by a number that is four times as large as when $S = 0$ (no alarm). We are neither able to obtain (nor are we interested in obtaining) the probability of G or the direct relationship between G and the rest of the network. Therefore, rather than encoding this information source as a variable G which is a child of S (as done in Equation 5.63), we can instead think of the application of a generalized delta function the joint probability of the three variables remaining after G is removed from the network, i.e., $p(h, s, w)\delta(s; \{(1, 8), (0, 2)\})$. This means that when $s = 1$ the probability is multiplied by 8 and when $s = 0$ it is multiplied by 2, a ratio of 4 to 1. In other words, the information from G imbues the variable S with virtual evidence favoring $S = 1$ with 4 to 1 odds. The universe only consists of the variables H, S, W , but there is information from outside the universe gives preferential treatment to certain values of S .

While Pearl [336] treats this as a Bayesian network in his Figure 2.1, he uses a different notation for his variable G which is not mentioned in the text explicitly. Namely, he states that G is a child of S only via a multi-patterned edge, where the first part is solid and the second part is dashed. What is meant by this diagram is the notion of virtual evidence, as given on the right in Figure 5.3. As mentioned above, this could be encoded by a Bayesian network with a variable V , always observed so that $V = 1$, with CPT $p(V = 1|S = s) = 8\delta(s, 1) + 2\delta(s, 0)$.

As is mentioned in Pearl's text, this external information source does not really impart a probability. In other words, it is correct neither to say that $p(S = 1|G = g) = 0.8$, $p(S = 0|G = g) = 0.2$ nor to say $p(G = g|S = 1) = 0.8$, $p(G = g|S = 0) = 0.2$ for any value of g since this would imply that this evidence is obtained by a finding of a variable within X_V . We next quote Pearl[336] directly¹

These difficulties arise whenever the task of gathering evidence is delegated to autonomous interpreters who, for various reasons, cannot explicate their interpretive process in full detail but nevertheless often produce informative conclusions that summarize the evidence observed. In our case, Mr. Holmes [an external observer] provides us with a direct mental judgment, based

¹In [336], a Mr. Holmes is the person who discusses with Mrs. Gibbon's her view of the alarm going off, and gleans from this discussion information external to the universe leading to the 4 to 1 preference in favor of alarm.

on Mrs. Gibbon's testimony, that the hypothesis *Alarm sound* should be accorded a confidence measure of 80%. The interpretation process remains hidden, however, and we cannot tell how much of the previously obtained evidence was considered in the process. Thus, it is impossible to integrate this probabilistic judgment with previously established beliefs unless we make additional assumptions. [i.e., we cannot compute $p(G, S)$.]

The prevailing convention in the Bayesian formalism is to assume that probabilistic summaries of virtual evidence are produced independently of previous information; they are interpreted as local binary relations between the evidence and the hypothesis upon which it bears, independent of other information in the system. For this reason, we cannot interpret Mr. Holmes's summary as literally stating $P(S|G) = 0.8$. $P(S|G)$ should be sensitive to variations in crime rate information — $P(H)$ — or equipment characteristics — $P(S|H)$. The impact of Gibbon's testimony should be impervious to such variations. Therefore, the measure $P(S|G)$ cannot represent the impact the phone conversation has on the truth of *Alarm sound*.

The likelihood ratio, on the other hand, meets this locality criterion, and for that reason probabilistic summaries of virtual evidence are interpreted as conveying likelihood information. For example, Mr. Holmes's summary of attributing 80% credibility of the *Alarm sound* event can be interpreted as:

$$P(G|\text{Alarm sound}) : P(G|\text{No alarm sound}) = 4 : 1 \quad (5.64)$$

In Pearl's description, he states that the crux lies in the *local binary relations between the evidence and the hypothesis upon which it bears*. By this, it is meant the evidence V should only influence S and should not directly influence anything else in the network, and that it should influence S only via ratios. These ratios are exactly what may be encoded by the virtual evidence construct given on the right of Figure 5.3. This information is obtained from a source external to the universe modeled by the BN's probability distribution. In the example, the information was obtained by an interview of Mrs. Gibbon and imparted into the network by the application of weights to the probabilities of the variables within the universe. Another example of this sort is given in [337] and in the references contained therein.

A third reason for the validity of the utilization of $p(V = 1|X_E)$ is that we may wish merely to impose constraints on certain variable configurations or states in our model. Perhaps there are only a subset of values $\mathcal{D}'_{X_E} \subset \mathcal{D}_{X_E}$ that are to be allowed. We may set $p(V = 1|X_E = x_E) = \delta\{x_E \in \mathcal{D}'_{X_E}\}$. Parameter learning can easily proceed, for example, in a model that is subject to these constraints. The semantics of the constraints, moreover, are similar to those used in constraint-networks [108]. In fact, in this way, virtual evidence can be seen to provide BNs the ability to represent hybrid constructs (see Section 5.7.1).

5.5.4 Virtual Evidence, Bayesian Inference, and Bayesian Reasoning

Bayesian reasoning has a long history in statistics, and it would lead us far too astray from the current topic to even touch the surface of its complexities. We refer the reader to [365, 336] and the multitudinous references in Bayesian statistics community that are available (and that we do not cite).

The essential tenet of Bayesian reasoning is that given prior uncertainty $p(A)$ over an event $\{A\}$, and given some evidence E that is related to $\{A\}$, that prior can be updated based on the likelihood to give posterior uncertainty, i.e., $P(A|E) = \left(\frac{p(E|A)}{p(E)}\right)p(A)$. This principle can be applied in a number of cases, including:

- Bayesian statistical inference, where $A = \theta$ is a set of parameters of a parametric [365] or non-parametric statistical model, and where $E = x_{1:N}$ is data that is drawn from an unknown distribution. The goal is to produce a posterior over parameters $p(\theta|x_{1:n})$ which can be used for future predictions, mixtures, maximum a-posterior estimation, confidence estimation, and so on (see [365]).

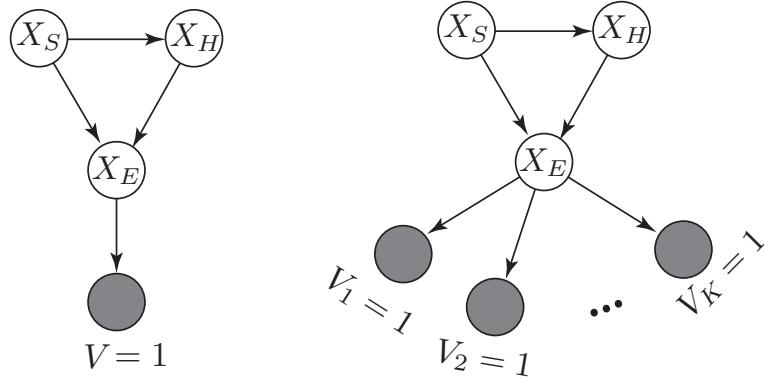


Figure 5.4: Left: The conditional independence assumption necessary for virtual evidence, namely that $V \perp\!\!\!\perp \{X_S, X_H\} | X_E$. Right, multiple sets of virtual evidence combine together in a natural way by having multiple virtual children.

- Belief revision, where A represents some event in the world (e.g., “it is raining outside”), and E represents some potentially related event (e.g., “the ground is wet”). In this case, the Bayesian approach simply uses Bayes rule to update prior beliefs about the state of the world to posterior beliefs.

Virtual evidence corresponds to Bayesianism in that it is possible to use Bayes rule to derive the posterior that is given in Equation 5.47. Specifically, we have:

$$p(x_S | V = 1) = \sum_{x_E} \sum_{x_H} p(x_S, x_E, x_H | V = 1) \quad (5.65)$$

$$= \sum_{x_E} \sum_{x_H} \frac{p(V = 1 | x_S, x_E, x_H) p(x_S, x_E, x_H)}{p(V = 1)} \quad (5.66)$$

$$= \frac{\sum_{x_E} \sum_{x_H} p(V = 1 | x_S, x_E, x_H) p(x_S, x_E, x_H)}{\sum_{x'_S} \sum_{x'_E} \sum_{x'_H} p(V = 1 | x'_S, x'_E, x'_H) p(x'_S, x'_E, x'_H)} \quad (5.67)$$

$$= \frac{\sum_{x_E} \sum_{x_H} p(V = 1 | x_E) p(x_S, x_E, x_H)}{\sum_{x'_S} \sum_{x'_E} \sum_{x'_H} p(V = 1 | x'_E) p(x'_S, x'_E, x'_H)} \quad (5.68)$$

which is identical to Equation 5.47. Bayes rule was used to get Equation 5.66, but it was necessary to utilize the conditional independence assumption that $V \perp\!\!\!\perp \{X_S, X_H\} | X_E$ to obtain Equation 5.68 (see left of Figure 5.4).

Therefore, while Bayesian reasoning lies at the heart of both Bayesian inference and Bayesian belief updating, they generally are applied to different random objects.

5.5.4.1 Soft Evidence vs. Virtual Evidence

Another form of belief updating exists as well, including Jeffrey’s evidence [337], or what is sometimes called *soft evidence*. This is a direct generalization of the probability revision approach discussed in Section 5.2.2.

Unlike virtual evidence, where one has the likelihoods $p(V = 1 | x_E)$, one instead has a probability distribution $p'(x_E)$ that is distinct from $p(x_E) = \sum_{x_{U \setminus E}} p(x_V)$. The goal in this approach is to update $p(x_V)$ to $p'(x_V)$ so that $p'(x_{U \setminus E} | x_E) = p(x_{U \setminus E} | x_E)$. This is done using Jeffrey’s update rule

$$p'(x_V) = \sum_{x'_E} p(x_V | x'_E) p'(x'_E) \quad (5.69)$$

In this notation, we note that x_V includes x_E (since $E \subset U$), and have that $p(x_V|x'_E) = 0$ if $x_E \neq x'_E$.

Soft evidence is fundamentally a different form of evidence than is virtual evidence. Soft evidence is a direct statement about the underlying statements made by a model (e.g., it directly states something about the parameters of a parametric model). In the case above, soft evidence states that the marginal of the distribution over X_E must equal $p'(x_E)$. Virtual evidence, on the other hand, does not prescribe properties to the distribution, but rather is a generalization of evidence regarding some of the random variables (and not their parameters) in the model. Of course, virtual evidence can be used to train the parameters of the model (say using maximum likelihood) but the way in which the evidence influences the parameters is only indirect with virtual evidence.

One way to see the difference between the two approaches is to see how the different forms of evidence combine. Suppose that we have two unequal sets of soft-evidence, i.e., $p'(x_E)$ and $p''(x_E)$. These two pieces of evidence do not combine, since if we apply Jeffrey's rule twice, only the latter application will survive since each application wipes out whatever is the current marginal over X_E . If the two sets of soft-evidence do not agree, they are fundamentally incompatible with each other.

Multiple sets of non-agreeing virtual evidence, on the other hand, can be sensibly unified, i.e., the different evidence will combine together to produce a logical combined result. For example, given the two virtual evidence constructs $p(V = 1|x_E)$ and $p(V' = 1|x_E)$ where V and V' are different random variables, the natural generalization of the above is to combine the two by multiplication, leading to the updated posterior:

$$p(x_S|V = 1, V' = 1) = \frac{\sum_{x_E} \sum_{x_H} p(V = 1|x_E)p(V' = 1|x_E)p(x_S, x_E, x_H)}{\sum_{x'_S} \sum_{x'_E} \sum_{x'_H} p(V = 1|x_E)p(V' = 1|x_E)p(x'_S, x'_E, x'_H)} \quad (5.70)$$

under the appropriate set of conditional independence statements. Generalizing this further to the vector observation $V_{1:K} = (1, 1, \dots, 1)$, we get:

$$p(x_S|V_{1:K} = (1, 1, \dots, 1)) = \frac{\sum_{x_E} \sum_{x_H} p(x_S, x_E, x_H) \prod_{k=1}^K p(V_k = 1|x_E)}{\sum_{x'_S} \sum_{x'_E} \sum_{x'_H} p(x'_S, x'_E, x'_H) \prod_{k=1}^K p(V_k = 1|x_E)} \quad (5.71)$$

as shown on the right in Figure 5.4.

5.6 Virtual Evidence, Undirected Graphical Models, and Factor Graphs

An undirected graphical model is a representation of any probability distribution that factors with respect to an undirected graph. In particular, if $p(x)$ factors with respect to undirected graph $G = (V, E)$, then we can write:

$$p(x) = \frac{1}{Z} \prod_{c \in \mathcal{C}} \phi_c(X_c) \quad (5.72)$$

where \mathcal{C} are the set of cliques in the graph. This means that any distribution $p(x)$ that factors with respect to graph G must be such that it can be validly written in this way.

Any Bayesian network can be written to factor as an undirected graph by defining cliques corresponding to each child and its parents (although some of the BN's factorizations, in particular V-structures, might be lost in this process when going to an undirected model). I.e., for each variable $i \in U$ and its parents π_i , we define a clique $c_i = \{i\} \cup \pi_i$, and form the set of cliques as $\mathcal{C} = \cup_i c_i$. Of course in this case, $Z = 1$ since the model already normalizes.

Virtual evidence can thus be seen as applying another clique function on a set of variables corresponding to X_E . Starting start with Equation 5.72, we add a virtual child corresponding to the factor $p(V_i = 1|X_i =$

$x_i) = f(x_i)$. The new distribution becomes

$$p'(x) = \frac{1}{Z} f(x_i) \prod_{c \in \mathcal{C}} \phi_c(X_c) = f(x_i)p(x) \quad (5.73)$$

Since X_i is only one variable, we can easily absorb the new factor $f()$ into any clique that contains X_i without changing any of the factorization properties of the graph. In this case, the graphical structure does not change and the same graphical model and all of its corresponding factorization properties still applies. On the other hand, the distribution does not any longer properly normalize unless we were to specify the above with a different constant Z' . Supposing further that there does not exist a $c \in \mathcal{C}$ such that $E \subseteq c$, then we form a new clique $c' = E$ and define the corresponding clique function $\phi_{c'}(x'_c) = p(V = 1|X_{c'} = x_{c'})$. Here the graph structure would change (i.e., X_E would turn into a sub-clique). Also, Z once again would need to be recomputed to obtain a normalized distribution. However, as shown in the sections above, Z is often not necessary.

A factor graph, on the other hand, is a bipartite graph representation of a probability distribution where factors in the distribution are made explicit. We are given a bipartite graph $G = (V, F, E)$ where V are the left-hand-side nodes, F are the right-hand-side nodes, and $E \subseteq V \times F$ are the edges that exist only between left and right-hand sides. Each $v \in V$ is associated with a random variable and each $f \in F$ is associated with a factor over some subset of the random variables. For every node $f \in F$, the subset of nodes in V connected to f correspond to the arguments of the factor f , and we call this V_f . I.e., $f \in F$ corresponds to the factor $f(X_{V_f})$. A factor graph represents any distribution $p(x)$ that can be validly written in the following way:

$$p(x) = \frac{1}{Z} \prod_{f \in F} \phi_f(X_{V_f}) \quad (5.74)$$

where $\phi_f()$ are arbitrary non-negative functions.

A factor graph can be used to represent the factorization of a BN precisely, since all factors in the BN can be represented by some factor $f \in F$. Adding a virtual child V_i to X_i correspond to adding another factor to the model, one that places a constraint on the possible value of X_i . Lets do this for each of an undirected model and a directed model.

A factor graph representation of virtual evidence would add one more node into the right hand side of the graph F , the node would only be connected to X_i , node's factor function would be the factor $f(x_i)$. Therefore, the factor graph does change in response to a virtual child of a single variable. If $|E| > 1$, we would just add a factor f' such that $X_{f'} = X_E$. The same normalization adjustment would occur here as what occurs in the undirected model.

When might one use virtual evidence vs. an undirected or factor graph? In some sense, there is no real difference, since any factor in a graph can be represented by an appropriate virtual evidence function $p(V = 1|x_E)$, up to a normalization constant. Sometimes, it is useful to parameterize factors in an undirected model using a log-linear form, so that $\phi_c(x_c) = \exp(\lambda^T g(x_c))$ where λ is a vector of coefficients and $g(\cdot)$ is a vector of functions on the variables x_c . It would be possible to define a virtual evidence function in the same way.

It can be useful at times to specify everything in terms of a log-linear model. On the other hand, it is sometimes quite useful to consider locally normalized factors as in a BN, where $p(x_i|x_{\pi_i})$ can be specified as x_i as a noisy function of its parents. In such a model, having the ability to add virtual evidence factors can be quite useful. The method of training also can have an influence on what to try. Recent work on discriminative training [445, 421, 256] are such that the exponential form is mathematically quite tractable, but such discriminative optimization criterion could just as easily be applied to a BN that utilizes virtual evidence. Therefore, the choice of what to use is really up to the user.

5.7 Applications

We present several applications where the use of virtual evidence has been usefully applied, and these include hybrid neural-network/hidden-Markov model-based speech recognition [60, 310] which has recently been generalized to use deep neural networks and has had lead to important improvements in speech recognition performance, backoff-based language models, and in dynamic Bayesian networks.

5.7.1 Expansion of Bayesian Network Models

Many statistical models can be described in the Bayesian network framework only with the use of such evidence generalization.

Example of the 4-cycle with a Bayesian network and VE.

Example of the factor graph 2-way vs. 3-way interaction with a Bayesian network and VE.

There is a lack of purity of such models, but then it is recovered by the sample space view. Why still then use a Bayesian network to represent such a factorization when one can instead use an MRF?

5.7.2 Virtual Evidence, Hybrid ANN/HMM systems, and hybrid deep neural networks/DGMs

Hybrid artificial neural network (ANN) - hidden Markov model HMM systems [60, 310] (or Hybrid ANN/HMM systems) were widely used in the 1990s for neural network based speech recognition systems. More recently, starting in around 2011, deep, or many-layered, neural network variants were utilized and shown to significantly improve performance [389]. Using neural networks with HMMs in this way is essentially a strategy where the neural network is employed as the unary potential of the HMM with the Markov chain acting as the smoothing function (see §8.4 and in particular §8.4.8 for a view of HMMs as Markov chain smoothing functions on top of a set of unary potentials). In this section, we will see that hybrid ANN/HMMs, and their deep variants, can be seen as the application of virtual evidence on an underlying Markov chain backbone.

Just a bit on notation and background on HMMs (full details and a complete description of HMMs are given in §8.4).

An HMM can be seen as a probability distribution over $2T$ random variables that factorizes as follows:

$$p(x_1, x_2, \dots, x_T, y_1, y_2, \dots, y_T) = p(x_{1:T}, y_{1:T}) = \prod_{t=1}^T p(x_t|y_t)p(y_t|y_{t-1}) \quad (5.75)$$

In a typical HMM, the T discrete random variables $Y_{1:T}$ form a Markov chain, so that

$$p(y_{1:T}) = \prod_{t=1}^T p(y_t|y_{t-1}), \quad (5.76)$$

and it is this Markov chain that is unobserved, or hidden, hence the name “hidden” Markov model.

The value T is not typically prescribed beforehand and might itself be of varied length or, in online streaming inference, might be either undefined or only implicitly defined only once the end of the stream is encountered.

The set of continuous or discrete random variables $X_{1:T}$ are typically observed and, in the case of speech recognition (and many other applications), represent a deterministic transformation of the acoustic speech waveform so it can be embedded in a vector stochastic process (i.e., each time step produces corresponds to an observed vector). We say that x_t is a *feature vector* at time t and y_t is the state at time t . Typically, x_t

is a vector in a continuous space but it could be a vector of integers, or even a vector of mixed real/integer values. The treatment of real vs. integer values is different.

When the vector $x_t \in \mathbb{R}^n$ is real, the observation distribution $p(x_t|y_t)$ is often modeled using a Gaussian mixture, i.e.,

$$p(x_t|y_t) = \sum_{m=1}^M p(x_t|m, y_t)p(m|y_t) \quad (5.77)$$

where $p(m|y_t)$ is a state-dependent multinomial and $p(x_t|m, y_t)$ is a Gaussian whose mean and variance are indexed by the pair (m, y_t) . That is,

$$p(x_t|m, y_t) = \mathcal{N}(x_t; \mu_{m,y_t}, \Sigma_{m,y_t}) \quad (5.78)$$

where μ_{m,y_t} is an n -dimensional mean vector and Σ_{m,y_t} is an $n \times n$ positive-definite matrix.

When x_t takes only integer values, then one often uses a multinomial distribution, but we do not address this case at this point.

In a Hybrid HMM/MLP systems, rather than using a Gaussian mixture for $p(x_t|y_t)$ we instead use a neural network, more specifically a multi-layered perceptron (i.e., a multi-layered perceptron which can be shown to approximate posterior probabilities when appropriately trained [57]). That is, let $p_{\text{NN}}(y_t|x_{t-M:t+M})$ be the output distribution of an MLP or a deep neural network at time t , using as input a window of width $2M + 1$ centered at time t , to produce local estimates of the temporally local posterior probabilities. Typically, anything more than two weight layers is considered deep, although it is often the case that ten or more layers are used in some deep systems.

We define a window of features of length $2M + 1$ as follows:

$$x_{t-M:t+M} \triangleq \{x_{t-M}, x_{t-M+1}, \dots, x_t, x_{t+1}, \dots, x_{t+M}\} \quad (5.79)$$

Hence, $x_{t-M:t+M}$ can be seen as a $n \times 2M + 1$ matrix that is indexed by the center point t .

The question is, how to merge the information that the ANN $p_{\text{NN}}(y_t|x_{t-M:t+M})$ locally garners from the acoustic time window with the stochastic process of the HMM (or really the Markov chain since once we strip away the observed Gaussian distribution of the HMM, all that is left is a Markov chain)?

The ANN is used in place of $p(x_t|y_t)$ in two general ways.

5.7.2.1 Prior Normalization of ANN Outputs

In the first way, as is argued in [60], we normalize the posterior by the prior, yielding scaled likelihoods, namely:

$$\text{scaled likelihood of } x_{t-M:t+M} = \frac{p_{\text{NN}}(y_t|x_{t-M:t+M})}{p(y_t)} = \frac{p_{\text{NN}}(x_{t-M:t+M}|y_t)}{p_{\text{NN}}(x_{t-M:t+M})} \quad (5.80)$$

where $p(y_t)$ is a prior probability over states. The right hand side of Equation (5.80) defines a ratio of two quantities, $p_{\text{NN}}(x_{t-M:t+M}|y_t)$ and $p_{\text{NN}}(x_{t-M:t+M})$ neither of which are ever explicitly computed — rather, we only compute them indirectly via the first ratio of the ANN output and the prior $p(y_t)$.

When this is used in an HMM, we replace the (Gaussian mixture) factors $p(x_t|y_t)$ in Equation (5.75) with a factor corresponding to this scaled likelihood. What we get instead of Equation (5.75) is a model that factors as follows:

$$\prod_{t=1}^T \frac{p_{\text{NN}}(y_t|x_{t-M:t+M})}{p(y_t)} p(y_t|y_{t-1}) = \prod_{t=1}^T \frac{p_{\text{NN}}(x_{t-M:t+M}|y_t)}{p_{\text{NN}}(x_{t-M:t+M})} p(y_t|y_{t-1}) \quad (5.81)$$

$$= \left(\prod_t \frac{1}{p_{\text{NN}}(x_{t-M:t+M})} \right) \prod_{t=1}^T p_{\text{NN}}(x_{t-M:t+M}|y_t) p(y_t|y_{t-1}) \quad (5.82)$$

Note that there is just a constant w.r.t. $y_{1:T}$ at the beginning, namely, $\prod_t \frac{1}{p_{\text{NN}}(x_{t-M:t+M})}$ which does not affect the Viterbi path or any posteriors over states. In other words, this constant does not matter in the cases where the model is used since

$$\underset{y_{1:T}}{\operatorname{argmax}} \prod_{t=1}^T \frac{p_{\text{NN}}(y_t|x_{t-M:t+M})}{p(y_t)} p(y_t|y_{t-1}) = \underset{y_{1:T}}{\operatorname{argmax}} \prod_{t=1}^T p(x_{t-M:t+M}|y_t) p(y_t|y_{t-1}) \quad (5.83)$$

Hence, the Viterbi paths are the same with or without the constant. Computing marginal posterior distributions over singletons y_t or pairs y_{t-1}, y_t also are unaffected by this constant.

It is worth considering the case, at this point, when $M = 0$, meaning a window of size one. In this case, once we divide by the priors as above, we get a model of the form:

$$\left(\prod_t \frac{1}{p_{\text{NN}}(x_t)} \right) \prod_{t=1}^T p_{\text{NN}}(x_t|y_t) p(y_t|y_{t-1}) \propto \prod_{t=1}^T p_{\text{NN}}(x_t|y_t) p(y_t|y_{t-1}) \quad (5.84)$$

which then has exactly the same form as the HMM in Equation (5.75) except that the observation distribution $p(x_t|y_t)$ comes rather than from a Gaussian and instead comes implicitly from a (potentially deep) neural network. Therefore, a discriminatively trained neural network can be used to produce a proper (scaled) likelihood value for the HMM, when the goal is either to compute a Viterbi path (or N-best list), or marginal posterior probabilities.

More generally, however, rather than taking only the features x_t at the current time with $M = 0$, we take a window of features $x_{t-M:t+M}$ with $M > 0$. That is, we have an MLP that produces a distribution over states $p_{\text{NN}}(y_t|x_{t-M:t+M})$ for each time frame. For example, in speech recognition, the ANN estimates take the form $p_{\text{ANN}}(y_t|x_{t-M:t+M})$, where $M = 4$, so that the posteriors may utilize information not just from the current but also from surrounding time window of nine frames.

For the general $M > 0$ case, the factorization resulting from applying the posterior-normalized ANN output is as follows:

$$p(x_1, x_2, \dots, x_T, y_1, y_2, \dots, y_T) = p(x_{1:T}, y_{1:T}) \propto \prod_{t=1}^T p(x_{t-M:t+M}|y_t) p(y_t|y_{t-1}) \quad (5.85)$$

If one attempted to define an HMM in the above form, one would immediately note that the r.h.s. is not a distribution since we are repeatedly using overlapping parts of the window in different factors. It can be easily seen, moreover, that the joint distribution $p(q_{1:T}, x_{1:T})$ does not factorize even proportionally with respect to any Bayesian network into a product of factors consisting of the likes of $p(x_{t-M:t+M}|y_t)/p(x_{t-M:t+M})$.

To turn it into a proper distribution one would need to normalize this somehow, perhaps as follows:

$$p(x_1, x_2, \dots, x_T, y_1, y_2, \dots, y_T) = p(x_{1:T}, y_{1:T}) = \frac{1}{Z} \prod_{t=1}^T p(x_{t-M:t+M}|y_t) p(y_t|y_{t-1}) \quad (5.86)$$

where Z ensures that the left-hand-side is a proper distribution. Hence, such a factorization can be defined via an undirected graphical model using (perhaps strange) factors of the form either $p(x_{t-M:t+M}|y_t)/p(x_{t-M:t+M})$ or just $p(x_{t-M:t+M}|y_t)$. This factorization would be unusual since normally when one performs a global normalization, there is no need nor desire to also have locally normalized factors that, like in a Bayesian network, do not also automatically ensure global normalization. This global normalization with such factors, however, is not typically done and so we are left with an unnormalized distribution with factors having overlapping features.

Even so, in fact, in speech recognition something like this is done even more commonly via the typical use of delta and double-delta features (see §8.8.9). This results in a model more like:

$$p(x_1, x_2, \dots, x_T, y_1, y_2, \dots, y_T) = p(x_{1:T}, y_{1:T}) = \prod_{t=1}^T p(f(x_{t-M:t+M})|y_t) p(y_t|y_{t-1}) \quad (5.87)$$

where $f(x_{t-M:t+M})$ is deterministic processing that extracts from $x_{t-M:t+M}$ the feature vector to be used at the current time t which might include the overlapping window of features, and also delta and delta-delta features. Hence, $f(x_{t-M:t+M})$ might indeed use more than the features at the current time frame leading to an overlapping of information in the HMM-like model above. This is almost always done in hybrid systems (and in the more recent deep models for speech recognition).

Again, since normalization is not done, this is not a proper probability distribution. In a little while, however, we will explain such a construct using the σ -algebras and event spaces that were offered as an explanation of virtual evidence in §5.5.2.

5.7.2.2 No Normalization of ANN Outputs

A second option for using the ANN outputs is to ignore the prior normalization. That is, we replace the (Gaussian mixture) factor $p(x_t|y_t)$ with the ANN output $p_{\text{NN}}(y_t|x_{t-M:t+M})$ directly, and produce a very “un-HMM-like” model that produces scores, for a given path $y_{1:T}$, of the form:

$$\prod_{t=1}^T p_{\text{NN}}(y_t|x_{t-M:t+M}) p(y_t|y_{t-1}) \quad (5.88)$$

Like in the above, there is no valid factorization of the joint distribution $p(q_{1:T}, x_{1:T})$ into a form like Equation (5.88), and a Bayesian network (again) does not factorize proportionally. Once again, one could normalize this model to produce a joint distribution $p(x_{1:T}, y_{1:T})$ as in

$$p(x_{1:T}, y_{1:T}) = \frac{1}{Z} \prod_{t=1}^T p_{\text{NN}}(y_t|x_{t-M:t+M}) p(y_t|y_{t-1}). \quad (5.89)$$

Again, however, this kind of global normalization is typically not done in practice.

In any event, we see that the observation score in this section, like in the previous, is obtained from an MLP rather than a Gaussian mixture.

5.7.2.3 Hybrid ANN/HMM systems as Pearl’s virtual evidence

The hybrid ANN/HMM systems, in both of cases of ANN score application in the two previous sections, be seen in terms Pearl’s virtual evidence applied to the variable y_t at each time. In this particular, we have time-dependent (or time-inhomogenous) virtual evidence being applied.

In particular, the underlying “universe” of random variables that partition the event space (as described in §5.5.2) consists only of a Markov chain $Y_{1:T}$. The ANNs are providing external virtual evidence to each variable Y_t in the form of

$$p(y_{1:T}) = \prod_{t=1}^T p(y_t|y_{t-1}) \delta(y_t, \{(j, \alpha_t^j)\}_{j=1}^{|Y|}) \quad (5.90)$$

and where the weight are such that

$$\alpha_t^j = p_{\text{NN}}(y_t|x_{t-\tau:t+\tau}) / p(y_t) = p_{\text{NN}}(x_{t-\tau:t+\tau}|y_t) / p_{\text{NN}}(x_{t-\tau:t+\tau}) \quad (5.91)$$

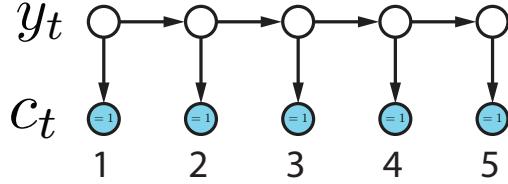


Figure 5.5: Hybrid ANN/HMM seen as virtual evidence. On the top we see a Markov chain for $p(y_{1:T}) = \prod_t p(y_t|y_{t-1})$. On the bottom, we see a collection of time-inhomogeneous factors for $p(c_t|y_t)$ which, depending on the form, imparts time-dependent virtual evidence to the Markov chain.

in the prior normalized case, or otherwise

$$\alpha_t^j = p_{\text{NN}}(y_t|x_{t-\tau:t+\tau}) \quad (5.92)$$

when there is no normalization.

Another way of seeing this is to note that Virtual evidence assumes that there is a variable c_t at time t always observed to be 1, and that it imparts a score on the state variables as in

$$\prod_{t=1}^T p(c_t = 1|y_t)p(y_t|y_{t-1}) \quad (5.93)$$

If we allow the virtual evidence scores to be time-inhomogeneous, we express this with a t -dependent factor, of the form:

$$\prod_{t=1}^T p_t(c_t = 1|y_t)p(y_t|y_{t-1}) \quad (5.94)$$

Note that now we have a separate virtual evidence factor $p_t(c_t = 1|y_t)$ for each time t .

It is also notable that things like the Viterbi paths and posteriors do not change up to a proportionality constant, and so the above is equivalent to:

$$\prod_{t=1}^T \beta_t p_t(c_t = 1|y_t)p(y_t|y_{t-1}) \quad (5.95)$$

where β_t is a time-dependent positive constant (but note that it is not a function of y_t , but it may be a function of $x_{t-M:t+M}$).

We can thus implement a Hybrid HMM/MLP by making the following identification: In the scaled likelihood approach:

$$p_t(c_t = 1|y_t) \leftarrow p_{\text{NN}}(y_t|x_{t-M:t+M})/p(y_t) \quad (5.96)$$

and in the second approach, just make the following identification:

$$p_t(c_t = 1|y_t) \leftarrow p_{\text{NN}}(y_t|x_{t-M:t+M}) \quad (5.97)$$

Figure 5.5 shows an example of a graphical model for a hybrid ANN/HMM using virtual evidence.

Since we now know that the external information provides unique information only up ratios, we see that the normalization constant $p(x_{t-\tau:t+\tau})$ is irrelevant to our three main inference problems. Moreover, we may assume that each variable Q_t has a virtual child $C_t = 1$ that is always observed to have value 1.

5.7.2.4 Generalized Hybrid Deep NNs/Bayesian Networks

Let us now generalize this notion to arbitrary Bayesian networks. A (collection of) discrete node(s) $X_E \subseteq U$ is given virtual evidence via some process entirely separate from the universe of variables X_V . This external process is specified via a joint distribution over two sets of variables, X_E and Z , and is given by $p(X_E, Z)$. Note that the variables within the distribution $p(X_E, Z)$ have some overlap with X_V (namely X_E) but $p(X_E, Z)$ also contains innovation Z . The set (X_E, Z) might be called a second *partially overlapping universe* relative to (X_E, Z) . Regardless of the name, it should be clear that the distribution $p(X_E, Z)$ is entirely separate from the distribution represented by the Bayesian network $p(X_V) = p(X_E, X_{U \setminus E})$. The distribution $p(X_E, Z)$ might itself be modeled by a Bayesian network, or might otherwise be modeled by a factors of neural networks, generalized linear models, support vector machines, or any other parametric or non-parametric and linear and/or non-linear form [187].

The question becomes, how do we utilize $p(X_E, Z)$ within $p(X_V)$? More specifically, suppose that Z becomes known, so that in the partially overlapping universe, we find that $Z = z$. The resulting distribution becomes:

$$p(X_E, Z = z) = p(X_E|Z = z)p(Z = z) = p(Z = z|X_E)p(X_E) \quad (5.98)$$

The portion that overlaps X_E is still unknown in both universes, but given $Z = z$ we have a refined notion of what X_E should be from the 2nd universe. There are several possible ways that we might use the information obtained in the 2nd universe to the 1st universe's benefit.

First, we might apply to $p(X_V)$ a delta function of the following form:

$$p(x_V)\delta(x_E, \{(\bar{x}_E^j, p(\bar{x}_E^j, Z = z))\}_{j=1}^M) \equiv p(x_V)\delta(x_E, \{(\bar{x}_E^j, p(Z = z|\bar{x}_E^j)p(\bar{x}_E^j))\}_{j=1}^M) \quad (5.99)$$

$$\equiv p(x_V)\delta(x_E, \{(\bar{x}_E^j, p(\bar{x}_E^j|Z = z))\}_{j=1}^M) \quad (5.100)$$

In other words, we can apply either use full joint distribution $p(\bar{x}_E^j, Z = z)$ or equivalently the likelihood $p(\bar{x}_E^j|Z = z)$ from the 2nd universe as a virtual evidence weight for the set of random variables X_E in the first universe. This is because $p(Z = z)$ is a constant for all values $x_E \in A_E$. Without loss of generality, let us call this the *posterior* case, since we are directly applying the posterior of x_E given z .

Alternatively, we might apply the probability of $Z = z$ in the 2nd universe as the first universe's virtual evidence weights as follows:

$$p(x_V)\delta(x_E, \{(\bar{x}_E^j, p(Z = z|\bar{x}_E^j))\}_{j=1}^M) \quad (5.101)$$

Note that this is not equivalent to applying say $p(\bar{x}_E^j|Z = z)$ as weights since $p(x_E)$ is not necessarily a constant in the second universe. We call this case the *likelihood* case, since we apply as virtual evidence weights to x_E in universe 1 the likelihood of the external data $Z = z$ given \bar{x}_E^j . This case corresponds to the hybrid ANN/HMM system mentioned above, since the value $p(\bar{x}_E^j|Z = z)/p(\bar{x}_E^j) = p(Z = z|\bar{x}_E^j)/p(Z = z)$ is proportional to the likelihood $p(Z = z|\bar{x}_E^j)$.

Which form of virtual evidence should we use, the *posterior* form of Equation 5.99 or the *likelihood* form of Equation 5.101? The one to use, depends on the application at hand. Examining the right side of Equation 5.99 and Equation 5.101, we see that the only difference is in the application of the prior probabilities $p(\bar{x}_E^j)$. These priors reflect some belief regarding the values of x_E in universe two irrespective of the variable Z . If it is desirable to encode weights on x_E in universe one only based on the local relationship between Z and X_E in universe two, then the likelihood approach Equation 5.101 should be used. If on the other hand we have obtained some external prior knowledge about X_E that we wish to apply *in addition* to the local relation between Z and X_E in universe two, then the *posterior* form Equation 5.99 should be used. We might even decide to encode and utilize weights on x_E based only on the prior information,

irrespective of any local external process, as in:

$$p(x_V) \delta(x_E, \{(\bar{x}_E^j, p(\bar{x}_E^j))\}_{j=1}^M) \quad (5.102)$$

This, however, is simply a restatement of the form given in Equation 5.38.

We see, however, that all three of the above forms are correct, since again all we are expressing in the universe are relative weights regarding different external beliefs about the variables X_E . And again it is only the ratios of these values that count in universe one.

5.7.3 Virtual Evidence and Backoff-based Language Models

Need to write.

5.7.4 Virtual Evidence and the IBM Machine Translation Models

Need to write.

5.8 Evidence in Markov random fields

Evidence can arrive for any probability distribution and this of course includes those described by MRFs. Like in a Bayesian network, a MRFs is such that some of its values are set, and those variables $E \subset V$ are designated as evidence nodes with \bar{x}_E being the value of the evidence.

Let \mathcal{C} be the maxcliques in undirected graph G , then for a MRF, any $p \in \mathcal{F}(G, \mathcal{M}^{(f)})$ we have

$$p(x_V) = \prod_{C \in \mathcal{C}^{(\text{mc})}} \psi_C(x_C) \quad (5.103)$$

With the evidence \bar{x}_E available we have

$$p(x_{V \setminus E}, \bar{x}_E) = \prod_{C \in \mathcal{C}^{(\text{mc})}} \psi_C(x_{C \setminus E}, \bar{x}_{E \cap C}) \quad (5.104)$$

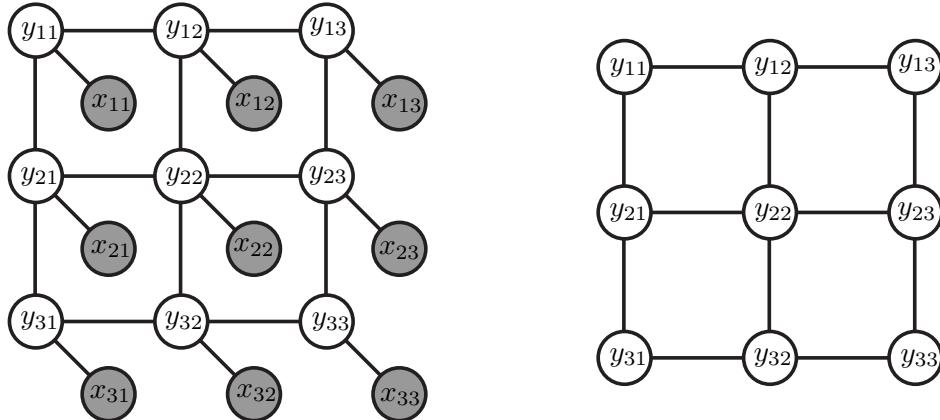


Figure 5.6: Left: an 3×3 MRF where each hidden variable y_{ij} is also attached to an observation variable x_{ij} . If the observation factors are absorbed into the factors over hidden variables, we can see this graph as being identical to the one on the right.

Like in the BN case, we can see evidence as additional Kronecker delta factors that preclude variable values except for those which match the evidence values. Consider, for example, the 3×3 hidden grid model, where x_{ij} are observed variables directly connected to y_{ij} which are arranged in a grid, as shown in Figure 5.6. This is a distribution over $p(\{(x_{i,j}, y_{i,j})\}_{i,j=1}^3)$, where \bar{x}_{ij} is observed in the factors $\psi_{ij,ij}(\bar{x}_{ij}, y_{ij})$ which is a function only of y_{ij} . This factor can be absorbed into any factor that involves y_{ij} . E.g., we can define a new factor as

$$\psi'_{12,13}(y_{12}, y_{13}) = \psi_{12,13}(y_{12}, y_{13})\psi_{12,12}(\bar{x}_{12}, y_{12}) \quad (5.105)$$

Thus, the left model can be viewed as identical to the right model, from the perspective of factorization over hidden variables. For much of our discussion on inference, we will be working with graphs of hidden variables. We will have an implicit understanding that some of the hidden nodes might involve “evidence factors”, and that the evidence factors will change the true nature of the query, but it will not change the computational properties of computing that query. For example, with evidence \bar{x}_E we might wish to compute $p(\bar{x}_E)$ but if the evidence factors are absorbed into the main factors, this computation might seem only to be an elaborate way of computing 1 (unit).

Now virtual evidence in an MRF may be also seen as an additional factor. Any additional factor on a variable provides another “score” on the variable. Like before, $\psi_{12,12}(\bar{x}_{12}, y_{12})$ can be seen as a parameterized soft evidence function, where \bar{x}_{12} and the factor itself is the parameter, and soft evidence is provided for y_{12} . Alternatively, a unitary factor function $\psi_{12}(y_{12})$ can be seen to provide soft evidence for various values of y_{12} . Again, this factor can be absorbed into any other factor that involves variable y_{12} .

There is one difference in how the graph may change between BNs and MRFs when utilizing factors that express soft evidence, or “preference” jointly over multiple random variables. In the BN, let's say that we wish to express such joint preferences over x_E so we would add a factor $p(V = 1|x_E)$. No additional edges would need to be added to the DAG in this case, although by adding this factor all of the elements of x_E would indeed be coupled by active paths (see the rules of d-separation). This means that if there was any independence property amongst the elements of x_E before evidence is applied, then those properties are removed. In the MRF case, we would add an additional factor of the form $\phi(x_E)$ that expresses evidence preferences over x_E . By adding this factor, we are explicitly saying that the graph family needs to be augmented with edges so that E now induces a clique in G ($G[E]$ is a clique).

5.9 Conclusion and Further Reading

Need to write.

Chapter 6

Inference on Trees, Triangulated Graphs, and Junction Trees

6.1 Introduction

We now have a firm grasp of the semantics of graphical models, namely that a graphical model corresponds to a family. We also understand the nature of various forms of evidence that might arrive to the model. We are in a good position to discuss inference.

First, what does inference mean? Inference consists of a number of things, all of which correspond to computing some probabilistic quantity. For example, suppose that we have a distribution $p(x)$ and that some set of nodes are evidence nodes \bar{x}_E . We discussed various quantities we may wish to calculate in previous chapters including the probability of evidence

$$p(\bar{x}_E) \tag{6.1}$$

or the posterior probability of some of the non-evidence variables

$$p(x_S | \bar{x}_E) \tag{6.2}$$

where $S \subseteq V \setminus E$, or the most probable assignment

$$\underset{x_S \in \mathcal{D}_{X_S}}{\operatorname{argmax}} p(x_S, \bar{x}_E). \tag{6.3}$$

In this last case, the most probable assignment, there can be a big difference between the case when $S = V \setminus E$ and when $S \subset V \setminus E$. When $S \subset V \setminus E$ this problem can be made considerably harder, as we see in Section ??.

As was discussed in the chapter on evidence, we are free to think of these quantities either with or without the evidence factors. For example, if we wish to start with a distribution $p(x_1, x_2, x_3)$ where \bar{x}_3 is evidence, and then we wish to get $p(\bar{x}_E)$, this can done either

In any form of graphical model inference, it might not be clear what we mean by performing inference on a graphical model, since inference, as seen as an algorithm, really applies only to a particular distribution. A graphical model, as we have seen, corresponds to a family. Basically, the process we wish to do is the following: We start with a graph G and a set of rules R , which defines a family $\mathcal{F}(G, R)$. We wish to produce an algorithm that can perform inference on any $p \in \mathcal{F}(G, R)$ perfectly accurately. But we wish to do this using only the pair (G, R) , without needing to know anything about the particular $p \in \mathcal{F}(G, R)$ we might encounter. We might think of this as an algorithm that takes as input the pair (G, R) and produces as

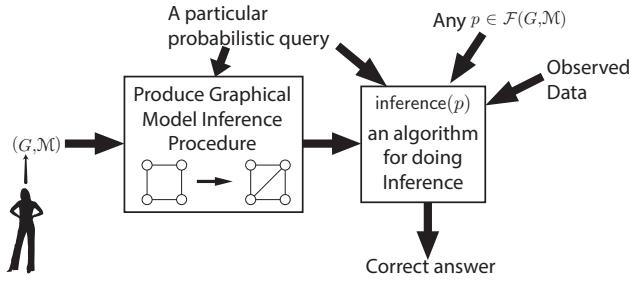


Figure 6.1: Graphical model inference means that we wish to take a graph G and a set of rules defining the graphical model type R , and produce an algorithm inference so that for any $p \in \mathcal{F}(G, R)$, $\text{inference}(p)$ will produce the correct quantity. An essential component for this is that we may wish to run $\text{inference}(p)$ for many different $p \in \mathcal{F}(G, R)$ so that the cost of producing $\text{inference}(p)$ starting from (G, R) is amortized over the many times $\text{inference}(p)$ is run.

output an algorithm inference, and where for any $p \in \mathcal{F}(G, R)$, $\text{inference}(p)$ will produce the correct result. This is shown in Figure 6.1.

The cost of producing algorithm inference is amortized by the members of $\mathcal{F}(G, R)$, since once we have the algorithm it can be used many times for many different distributions. Moreover, there are a number of questions we may wish to ask including:

- Can we map from $(G, R) \mapsto \text{inference}$ automatically, or do we require human intervention?
- Is it possible to find instance inference that is as computationally efficient as possible? Is it even possible to know what the most efficient instance is?
- Is it possible to use only graph-theoretic properties of G ?
- Have we lost anything efficiency-wise in producing a generic algorithm? Might there be cases of $p \in \mathcal{F}(G, R)$ for which a specific algorithm for this particular p might be more efficient? If so, is it more efficient only in the constants (and implementation) or is it inherently more efficient?
- Is there a way to utilize G, R to produce algorithms that produce approximate rather than exact results for inference?

We hope to answer questions of these sorts in this and in the upcoming chapters.

In previous chapters, we discussed the differences between different graphical model families, including families for BNs, MRFs, and factor graphs. We will see that, except for a few exceptions, it will not be detrimental to discuss tree inference only in the context of MRFs. The reason for that is once exact inference is performed, we are implicitly transforming the graph from its original form to one where inference may be performed exactly, and in doing so the model is moved from the family specified by the original graph to the (usually larger and enclosing) family of models for which inference is solved. While this idea and why it is the case might not be clear at this point to the reader, the forthcoming pages will attempt to clarify.

First, suppose that we are starting with a BN. This means that there is some DAG G and we have $p \in \mathcal{F}(G, \mathcal{M}^{(\text{df})})$. We saw that $p \in \mathcal{F}(G, \mathcal{M}^{(\text{df})}) \Rightarrow p \in \mathcal{F}(m(G), \mathcal{M}^{(\text{f})})$, where $m(G)$ is the undirected graph obtained by moralizing the directed graph G . This means that if we have a procedure that solves an inference problem for every member of $\mathcal{F}(m(G), \mathcal{M}^{(\text{f})})$ then we have also solved it for every member of $\mathcal{F}(G, \mathcal{M}^{(\text{df})})$. The question however is have we lost anything in moving to this larger family, and if so, is what we have lost crucial for performing inference efficiently? Clearly any independence properties that arise solely from V-structures are lost since all formally-unconnected parents are now connected. From the

point of view of performing a computation, however, we are concerned about losing these properties only if they make inference more computationally costly. We will examine this issue again at the end of the chapter once inference is clear.

Next, suppose we start with a factor graph $p \in \mathcal{F}(G, \mathcal{M}^{(\text{fg})})$. We can construct a MRF graph G' where for every factor f in the factor graph G there is a clique in the MRF graph G' and therefore, we again have that $\mathcal{F}(G, \mathcal{M}^{(\text{fg})}) \subseteq \mathcal{F}(G', \mathcal{M}^{(\text{f})})$. Again, something is potentially lost in this conversion, in particular the composition of the factors in the cliques of the MRF, but perhaps again we are unconcerned if this does not affect inference complexity. Once again, we revisit this topic at the end of the chapter.

Therefore, we assume that we are given some MRF G that obeys the MRF Markov properties described in Chapter ???. This MRF could have come from a BN, a factor graph, or a could be an original MRF but we will not initially make this distinction or worry about the graphs origin.

The next concern we must have is evidence. We saw in Chapter ?? how standard evidence can be viewed in a model as the multiplication in the model by additional Kronecker delta function. For example, if we find \bar{x}_E then we change the probability model to:

$$p(x) \prod_{e \in E} \delta(x_e, \bar{x}_e) \quad (6.4)$$

In the MRF case, we can consider this to be an additional factor on each node. That is, given $p \in \mathcal{F}(G, \mathcal{M}^{(\text{f})})$ with set of cliques \mathcal{C} ,

$$p(x) = \frac{1}{Z} \prod_{c \in \mathcal{C}} \psi_c(x_c) \quad (6.5)$$

then for each $e \in E$ we find a $c \in \mathcal{C}$ such that $e \in c$ and form new factors

$$\psi'_c(x_c) = \psi_c(x_c) \prod_{e \text{ assigned-to } c} \delta(x_e, \bar{x}_e) \quad (6.6)$$

While the normalization constant $1/Z$ is no longer such that it produces a valid probability model, we saw in Chapter ?? that the normalization does not affect the quantities we are interested in any substantial way. We will address this issue again, however, when we discuss certain conditional models

6.2 Inference on Trees

What does it mean to perform inference on trees? This means that we are discussing all $p \in \mathcal{F}(T, \mathcal{M}^{(\text{f})})$ where $T = (V, E)$ is a graph that is a tree. A tree has the following definition

Definition 40. A graph $G = (V, E)$ is a forest if it is the case that for all $u, v \in V$, there is no more than one path that connects u to v in G . Given a forest G , if for all $u, v \in G$ there is a unique path connecting u and v , then it is called a connected forest or just simply a tree.

We can equivalently define a forest as a graph that does not have any cycles, and a tree is a connected graph without cycles. Figure 6.2 displays several trees, all of which, when considered together as one graph, constitute a forest:

There are other identical characterizations of trees as summarized in the following theorem.

Theorem 41 (Trees, Berge). Let $T = (V, E)$ be an undirected graph with $|V| = n > 2$. Then each of the following properties are equivalent and each can be used to define a tree.

- (a) T is connected and has no cycles

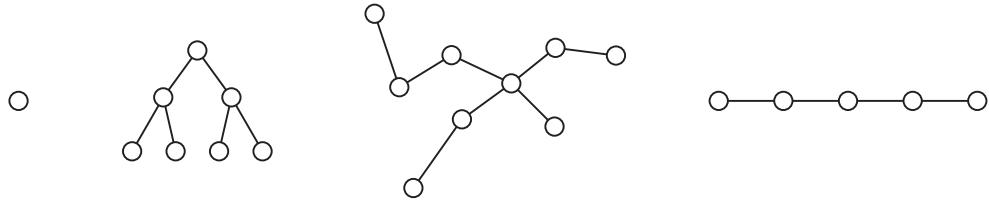


Figure 6.2: Examples of four undirected graphs that are trees. Left: a single node is a tree. Left-center: this shows a tree as it is typically drawn, with an obvious root at the top, and branching as we move downwards. Right-center: A more general tree, any node could be considered the root, but the branching factor at each node would vary from one node to the next. Right: A chain of any length (in this case length 5) is also a tree.

- (b) G has $n - 1$ edges and has no cycles,
- (c) G is connected and contains exactly $n - 1$ edges,
- (d) G has no cycles, and if an edge is added to G , exactly one cycle is created,
- (e) G is connected, and if any edge is removed, the remaining graph is not connected,
- (f) Every pair of vertices of G is connected by one unique path.
- (g) G can be generated via the following process: Start with a node v . Repeatedly choose a next vertex, and connect it with an edge to exactly one previously chosen vertex.

Note that the size of the maximum clique in any tree (or forest) is 2 since there can be no cycles — any set of nodes larger than 2 will not be a clique, but instead will constitute a forest sub-graph of the tree.

From the perspective of a $p \in \mathcal{F}(T, \mathcal{M}^{(f)})$ this means that all factors of p involve at most two variables, namely the variables corresponding to nodes adjacent to some edge in T . Thus, any $p \in \mathcal{F}(T, \mathcal{M}^{(f)})$ can be written as:

$$p(x_1, \dots, x_N) = \prod_{e \in E(T)} \psi_e(x_e) = \prod_{(i,j) \in E(T)} \psi_{i,j}(x_i, x_j) \quad (6.7)$$

Any such distribution is called a *tree distribution*. Note that sometimes tree distributions are written with the associated node factors as well, in the form

$$p(x_1, \dots, x_N) = \prod_{v \in V(T)} \psi_v(x_v) \prod_{(i,j) \in E(T)} \psi_{i,j}(x_i, x_j) \quad (6.8)$$

We see, from the fact that $\mathcal{F}(G, \mathcal{M}^{(cf)}) = \mathcal{F}(G, \mathcal{M}^{(mcf)})$ as mentioned in Chapter XXX that they are the same — node factors can always be held by edge factors, so we will always take the maxclique form of factorization, which in this case means we have factors for each edge.

Note that it is possible to write any distribution $p \in \mathcal{F}(T, \mathcal{M}^{(f)})$ in the following way. For each node $v \in V(T)$, let $p(x_v)$ be the marginal distribution on that node. Also, for every edge $e = (i, j) \in E(T)$, let $p_{i,j}(x_i, x_j)$ be the marginal distribution corresponding to edge e adjacent to nodes i and j . Then we can write p as follows in terms of these *marginals*.

$$p(x_1, \dots, x_N) = \prod_{v \in V(T)} p_v(x_v) \prod_{(i,j) \in E(T)} \frac{p_{i,j}(x_i, x_j)}{p_i(x_i)p_j(x_j)} \quad (6.9)$$

Proof. If $|V(T)| = 1$ or $|V(T)| = 2$, then Equation (6.9) follows immediately. Assume it holds for any tree with $|V(T)| < k$ and assume $|V(T)| = k$. We use the tree generation procedure from Theorem 41-(g) to generate the tree inductively in k , and let T_k be the tree with k nodes. Hence, vertex k is connected to a single unique node in the tree, lets call it $i(k)$. Then we have:

$$p(x_1, x_2, \dots, x_k) = p(x_k | x_1, x_2, \dots, x_{k-1}) p(x_1, x_2, \dots, x_{k-1}) \quad (6.10)$$

$$= p(x_k | x_{i(k)}) p(x_1, x_2, \dots, x_{k-1}) \quad (6.11)$$

$$= \frac{p(x_k, x_{i(k)})}{p(x_{i(k)})} \prod_{v \in V(T_{k-1})} p_v(x_v) \prod_{(i,j) \in E(T_{k-1})} \frac{p_i(x_i, x_j)}{p_i(x_i)p_j(x_j)} \quad (6.12)$$

$$= p(x_k) \frac{p(x_k, x_{i(k)})}{p(x_k)p(x_{i(k)})} \prod_{v \in V(T_{k-1})} p_v(x_v) \prod_{(i,j) \in E(T_{k-1})} \frac{p_i(x_i, x_j)}{p_i(x_i)p_j(x_j)} \quad (6.13)$$

$$= \prod_{v \in V(T_k)} p_v(x_v) \prod_{(i,j) \in E(T_k)} \frac{p_i(x_i, x_j)}{p_i(x_i)p_j(x_j)} \quad (6.14)$$

□

This is an important theorem as it will also have an equivalent form in terms of junction trees, as we will see later in this chapter.

Exercise 42. Is there an alternative way to prove this theorem recursively using the tree separators?

For a given vertex $v \in V(T)$, let $\delta(v)$ be the set of neighbors of v in T . Using this, we can also write the tree distribution in the following way:

$$p(x) = \frac{\prod_{(i,j) \in E(T)} p_i(x_i, x_j)}{\prod_{v \in V(T)} (p_v(x_v))^{| \delta(v) | - 1}} \quad (6.15)$$

Proof. This clearly holds for $|V(T)| = 1$ (where $\delta(v) = \emptyset$) or $|V(T)| = 2$, so assume it holds for $|V(T)| < k$. Choose any vertex in the tree, say v . The removal of node shatters the tree into $|\delta(v)|$ sub-trees each with fewer than k vertices, and each of which can be written as Equation (6.15). Let T_1, T_2, \dots, T_N be these subtrees, with $N = |\delta(v)|$. Within each of these subtrees, moreover, v has only one neighbor, and hence it does not appear in the denominator of Equation (6.15) for the subtrees. Hence, we get:

$$p(x) = p(x_{V(T_1) \setminus v}, x_{V(T_2) \setminus v}, \dots, x_{V(T_N) \setminus v}, x_v) \quad (6.16)$$

$$= p(x_{V(T_1) \setminus v}, x_{V(T_2) \setminus v}, \dots, x_{V(T_N) \setminus v} | x_v) p(x_v) \quad (6.17)$$

$$= p(x_v) \prod_{n=1}^N p(x_{V(T_n) \setminus v} | x_v) \quad (6.18)$$

$$= p(x_v) \prod_{n=1}^N \frac{p(x_{V(T_n) \setminus v}, x_v)}{p(x_v)} \quad (6.19)$$

$$= \frac{1}{(p_v(x_v))^{|\delta(v)|-1}} \prod_{n=1}^N \frac{\prod_{(i,j) \in E(T_n)} p_i(x_i, x_j)}{\prod_{v' \in V(T_n)} (p_{v'}(x_{v'}))^{|\delta(v')|-1}} \quad (6.20)$$

This last factor immediately leads to Equation (6.15). □

Factorizing with respect to a tree, as we will see in later chapters, does not alone necessitate that inference will be easy (as is commonly believed), but with size-2 factors, and the tree property we will see that

in many cases (namely when the random variables do not have too large a domain size) inference can be performed efficiently.

A *chain*, as its name suggests, is a graph consisting of a sequence of nodes that are connected in succession. That is, if there are N nodes v_1, v_2, \dots, v_N , then edges exist only connecting successive pairs – if $e \in E$, then $e = \{v_i, v_{i+1}\}$ for $i \in \{1 \dots N\}$. Figure 6.3 shows several chains.

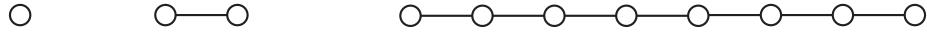


Figure 6.3: Examples of undirected graphs that are chains

A chain is clearly a tree, but a tree is not necessarily a chain. Nevertheless, we see that a chain is in some sense simpler than a tree¹ so we describe inference on chains first. Observe that a chain, moreover, is an example of a dynamic graphical model, and we will discuss aspects of inference that are specific to such graphs and their generalizations in later chapters. Here, we describe inference on chains only to motivate inference on trees, but later we will see that inference on chains lies at the core of inference in dynamic models (which can be seen as fat chains).

Suppose p factors as a chain, which means that

$$p(x) = \prod_{i=1}^{N-1} \psi_{i,i+1}(x_i, x_{i+1}) \quad (6.21)$$

We may be interested in computing the quantity $p(x_3, x_4)$. In order to do this, we would perform the following operations:

$$p(x_3, x_4) = \sum_{x_1} \sum_{x_2} \sum_{x_5} \sum_{x_6} \cdots \sum_{x_N} p(x_1, x_2, \dots, x_N) \quad (6.22)$$

Let us suppose that each random variable has $r = |\mathcal{D}_{X_i}| \forall i$ possible values. If we were to perform this calculation naively as given, then it would require $O(r^N)$ operations as Algorithm 2 suggests.

Algorithm 2: naïve chain computation.

```

1 foreach  $(x_3, x_4) \in \mathcal{D}_{X_3} \times \mathcal{D}_{X_4}$  do
2      Compute  $\sum_{x_1} \sum_{x_2} \sum_{x_5} \sum_{x_6} \cdots \sum_{x_N} p(x_1, x_2, \dots, x_N)$ 

```

Line one of Algorithm 2 is performed r^2 times, and line two requires $O(r^{N-2})$ operations, thus leading to an $O(r^N)$ overall complexity.

To make what we do next extremely clear, we assume for the moment that $N = 5$ so that $p(x_3, x_4)$ requires the following mathematical procedure:

$$p(x_3, x_4) = \sum_{x_1} \sum_{x_2} \sum_{x_5} \psi_{1,2}(x_1, x_2) \psi_{2,3}(x_2, x_3) \psi_{3,4}(x_3, x_4) \psi_{4,5}(x_4, x_5) \quad (6.23)$$

which is $O(r^5)$. The above computation can take advantage the distributive law over \mathbb{R} (i.e., $ab + ac =$

¹While a chain might seem simpler than a tree, it turns out that learning an optimal tree is solvable by a simple greedy algorithm, while learning an optimal chain is an NP-complete optimization problem.

$a(b + c)$), to improve the computation. The distributive law in this case states the following:

$$\begin{aligned} & \sum_{x_1, x_2, \dots, x_N} \left(\prod_{c \in \text{factors not involving } x_i} \psi_c \right) \left(\prod_{c \in \text{factors involving } x_i} \psi_c \right) \\ &= \sum_{x_1, \dots, x_{i-1}, x_{i+1}, \dots, x_N} \left(\prod_{c \in \text{factors not involving } x_i} \psi_c \right) \sum_{x_i} \left(\prod_{c \in \text{factors involving } x_i} \psi_c \right) \end{aligned} \quad (6.24)$$

Let us therefore use this property to distribute the first sum over x_1 into the product. There is some choice as to where it should go but Equation 6.24 suggests that it should involve no more than the factors that involve x_1 , or as far to the right as possible. Equation 6.23 becomes:

$$\begin{aligned} p(x_3, x_4) &= \sum_{x_2} \sum_{x_5} \psi_{3,4}(x_3, x_4) \psi_{4,5}(x_4, x_5) \psi_{2,3}(x_2, x_3) \underbrace{\sum_{x_1} \psi_{1,2}(x_1, x_2)}_{\phi_{1,2}(x_2)} \\ &= \sum_{x_2} \sum_{x_5} \psi_{3,4}(x_3, x_4) \psi_{4,5}(x_4, x_5) \psi_{2,3}(x_2, x_3) \phi_{1,2}(x_2) \end{aligned} \quad (6.25)$$

where $\phi_{1,2}(x_2)$ is a function of x_2 only. The notation $\not\in$ indicates that x_1 has been summed away.

It is said here that the node x_1 has been “eliminated” because once it is summed (or marginalized) away, it is no longer part of the computation — no later computation will involve such a node. In fact, the word “eliminate” is widely used in the context of summing away variables. As we will see, the operation of eliminating nodes corresponds directly to a strictly graphical operation of eliminating nodes (defined below) and is also the source of the name of the variable elimination algorithm.

Observe that constructing $\phi_{1,2}(x_2)$ costs $O(r^2)$ operations (one summation for every value of x_2).

We now, in Equation 6.25, have an expression that does not involve x_1 and we can further sum out the other variables. Lets next choose x_2

$$p(x_3, x_4) = \sum_{x_5} \psi_{3,4}(x_3, x_4) \psi_{4,5}(x_4, x_5) \underbrace{\sum_{x_2} \psi_{2,3}(x_2, x_3) \phi_{1,2}(x_2)}_{\phi_{1,2,3}(x_3)} \quad (6.26)$$

$$= \sum_{x_5} \psi_{3,4}(x_3, x_4) \psi_{4,5}(x_4, x_5) \phi_{1,2,3}(x_3) \quad (6.27)$$

In this case, $\phi_{1,2,3}(x_3)$ indicates that both variables x_1 and x_2 have been eliminated and that it is only a function of x_3 . Observe, the step to produce $\phi_{1,2,3}(x_3)$ cost again only $O(r^2)$ operations. There is only one additional variable to eliminate x_5 . We again move the sum in as far to the right as possible yielding

$$p(x_3, x_4) = \psi_{3,4}(x_3, x_4) \phi_{1,2,3}(x_3) \underbrace{\sum_{x_5} \psi_{4,5}(x_4, x_5)}_{\phi_{3,4}(x_4)} \quad (6.28)$$

$$= p(x_3, x_4) = \psi_{3,4}(x_3, x_4) \phi_{1,2,3}(x_3) \phi_{3,4}(x_4) \quad (6.29)$$

Once again, this last step requires $O(r^2)$ operations, and we end up with a function of only two variables, so the entire computation required $O(r^2)$ steps. In general, with a length- N chain, the computation can be done to require only $O(Nr^2)$ operations.

The above computation shows what happens if we eliminate the variables in the order $(1, 2, 5)$. Suppose, on the other hand, we had performed the elimination of the variables in a different order. For example, if

we eliminated in the order $(5, 1, 2)$ or $(1, 5, 2)$ we would still obtain a computation for $p(x_3, x_4)$, one which requires $O(r^2)$ operations. Not all of the orders have this nice property, though. What if we were to eliminate the variables in the order $(2, 1, 5)$. We go through the following steps:

$$p(x_3, x_4) = \sum_{x_1, x_5} \psi_{3,4}(x_3, x_4) \psi_{4,5}(x_4, x_5) \underbrace{\sum_{x_2} \psi_{1,2}(x_1, x_2) \psi_{2,3}(x_2, x_3)}_{\phi_{2,1,3}(x_1, x_3)} \quad (6.30)$$

$$= \sum_{x_5} \psi_{3,4}(x_3, x_4) \psi_{4,5}(x_4, x_5) \sum_{x_1} \phi_{2,1,3}(x_1, x_3) \quad (6.31)$$

$$= \psi_{3,4}(x_3, x_4) \phi_{2,1,3}(x_3) \sum_{x_5} \psi_{4,5}(x_4, x_5) \quad (6.32)$$

$$= \psi_{3,4}(x_3, x_4) \phi_{2,1,3}(x_3) \phi_{5,4}(x_4) \quad (6.33)$$

This all seems fine until one realizes that the sum over x_2 in Equation 6.30 is over two factor functions and given that one factor involves x_1, x_2 and the other involves x_2, x_3 , this requires $O(r^3)$ operations — for every set of values of the pair x_1, x_3 we need to compute the sum. The total complexity of the computation in the chain with this order is then $O(r^3)$ which, when r is large, much worse than $O(r^2)$.

It appears, therefore, that some of the possible node elimination orders might be much worse than others. How, then, can we ensure that we choose the best order?

Under some orders, we inextricably couple together some of the factors in a way so that they can never be decoupled again. The reason for this is that, for arbitrary functions $f_1(\cdot, \cdot)$ and $f_2(\cdot, \cdot)$, there are no functions $g(a)$ and $h(c)$ that constitute a factorization of a sum as in:

$$g(a)h(c) = \sum_b f_1(a, b) f_2(b, c) \quad (6.34)$$

The process of eliminating the variable b from the product has produced a new function of the pair a, c that does not factorize in general thereby removing the possibility of further use of the distributive law. In general, for disjoint variables $A, B, C \subseteq V$, the function

$$f(x_A, x_C) = \sum_{x_B} f_1(x_A, x_B) f_2(x_B, x_C) \quad (6.35)$$

does not factor. This means that there exists no functions g, h such that $f(x_A, x_C) = g(x_A)h(x_C)$. In order for the above equation to be true, both $f_1(x_A, x_B)$ and $f_2(x_B, x_C)$ must themselves factor with respect to their variables, but in a graphical model the presence of an edge is specifically meant to indicate that members of the corresponding family do not factor in general (i.e., are not required to factor) over the two variables connected by an edge. Given that we are starting with functions $f_1(x_A, x_B)$ and $f_2(x_B, x_C)$ before the summation, this suggests that the corresponding graphical model is such that both $G[A \cup B]$ and $G[B \cup C]$ are cliques. To capture the result $f(x_A, x_C)$ suggests that after the summation, $G[A \cup C]$ should also be a clique (if it is not already).

This is where the graph that defines the family can help. Consider the following purely graph-theoretic operation for eliminating a variable in a graph:

Definition 43. Elimination: *To eliminate a node $v \in V$ in an undirected graph G , we first connect all neighbors of v and then remove v and all v 's adjacent edges from the graph.*

Lets stay that starting from a graph $G = (V, E)$, eliminating node $v \in V(G)$ leads to a vertex-induced sub-graph $G' = (V', E')$ where $V' = V \setminus \{v\}$, and where $E' = (E \cap V' \times V') \cup \delta_G(v) \times \delta_G(v)$. This

means that once v has been eliminated from the graph, the neighbors of the (former) node v now form a clique. Figure 6.4 shows some examples of node elimination in both chains and other graphs.

When eliminating v from G we will sometimes use the notation $G_v = G'$. When we have a particular ordering of the nodes $\sigma = (\sigma_1, \dots, \sigma_N)$, we might eliminate the nodes in the graph in that order. We will say that $G_1 = G_{\sigma_1}$ is the result of eliminating σ_1 from G , that $G_2 = (G_{\sigma_1})_{\sigma_2}$ is the result of eliminating σ_2 starting from G_{σ_1} , that $G_3 = ((G_{\sigma_1})_{\sigma_2})_{\sigma_3}$ is the result of eliminating σ_3 starting from G_2 and so on. Therefore, the ordering σ defines a sequence of graphs $(G_0, G_1, G_2, \dots, G_{N-1})$ where $G_0 = G$ and where G_{N-1} consists of only one node σ_N . We will call this series of graphs *elimination graphs*.

It is no coincidence that both the graph-theoretic operation and the summation operation above are called elimination.

As might be expected, the graphical operation of eliminating nodes from a graph, and the computational operation of eliminating variables by summing them out of the equation correspond to each other in important ways. Specifically, the set of factors that are inextricably coupled by eliminating a variable via summation are precisely those that involve any of the nodes that are connected in the graphical version of the elimination of that variable. The reason for this is that when summing out a random variable, say x_i , those other variables that become inextricably coupled are those variables that are involved in factor with x_i (all other variables can be temporarily ignored due to the distributive property). Those variables that are involved in a factor with x_i are exactly the same variables that are adjacent to x_i in the covering graphical model. Once the random variable summation has occurred, we are left with a new factor involving all the variables that used to be in at least one factor with x_i (equivalently, all the nodes that had a neighbor with x_i). Since these variables are now inextricably intertwined (in general), this can only be represented by a single factor — in the graph, all these variables become connected in a clique thereby expressing any further lack of factorization amongst those variables.

Whenever a factor exists $f(x_A)$ over a set of variables A , any corresponding MRF must have that x_A forms a clique in the graph. That is, newly coupled variables can only be represented by a single factor. When forming the new factor $f(x_A, x_C)$ in

$$f(x_A, x_C) = \sum_{x_b} f_1(x_A, x_b) f_2(x_b, x_C) \quad (6.36)$$

by summing out x_b , the computational cost is $O(r^{|A \cup C|+1})$ for scalar sum over x_b . This is exponential cost in the size of resulting coupled variables. Graphically, the sets $A \cup C$ correspond to the nodes that are neighbors of $b \in V$ at the time of elimination. Thus, the graphical neighbors of a node determine the (exponential) cost of doing a variable elimination.

If therefore we wish to keep the coupling of factors to a minimum, we should choose nodes to eliminate that either:

1. have only zero or one neighbor (so that no new edges are added), or
2. have neighbors that are already connected so that eliminating the node will not add any new edges.

In the first case, when there are zero neighbors, the elimination step costs $O(r)$ and when there is one neighbor, the cost is $O(r^2)$.

In the second case, the the cost is going to be exponential in the size of the variable and the neighbor set (i.e., the cost will be $O(r^{|\delta(v)|+1})$ for eliminating variable v in the current graph), but this cost is unavoidable. That is, v along with its fully connected neighbors constitute a clique in G and the appearance of this clique means that any $p \in \mathcal{F}(G, \mathcal{M}^{(f)})$ is allowed to have a single non-decomposable factor of the form $f(x_v, x_{\delta(v)})$. At one point or another, we will encounter this factor when we eliminate all of the variables and at that time we will pay the $O(r^{|\delta(v)|+1})$ cost. If we are not able to find a node that satisfies case one, there will be no penalty to taking our medicine sooner rather than later and eliminating the first variable in this large factor.

Skipping forward a bit, we will see that this idea corresponds to the *min-fill* heuristic for choosing an elimination order in arbitrary graphs (see Section XXX), but for now we are still considering only trees.

Imagine now choosing a particular order of nodes σ to eliminate, eliminating those nodes in that order and adding edges to the graph as the process of elimination occurs, and then reconstituting the graph at the end, but adding back in all those edges that were added in the process. These additional edges are called *fill-in* edges, and we can denote the resulting graph $G_F = (V, E \cup F)$ where F is the set of additional edges that have been added during the process of elimination.

It is extremely important to realize the following fact. From the perspective of computation, we might as well have had those additional edges in the graph to begin with. That is, any set of neighbors $w, u \in \delta(v)$ that are not connected by an edge at the time we eliminate v might as well have been connected originally. Since those edges are inevitable, for the current order, so we can add them to the graph before running the elimination procedure. Then, running the elimination procedure with the same order will have the same computational properties. Stated another way, if we were to use the same ordering of the nodes, and eliminate them on a graph G_F we would not induce any additional fill-in edges because all the edges that are required to connect the neighbors of a node as it is eliminated are already in the graph. The reason is that on the second time around, when we eliminate on G_F , there are never any new edges adjacent to each node that weren't there when we eliminated a node in G — there is no way for latter elimination steps to add edges to a node that has already been eliminated.

Add more discussion here about what the “might as well have started with a larger family”. The various tradeoffs involve what a scientist might do, and how from a scientific perspective the scientist might be adding edges to a model only from the perspective of model accuracy, and not realizing the computational implications of those additional edges. For example, he might be withholding edges from the model, under the assumption that fewer edges always means faster exact computation, but in fact this might not be the case, since any elimination order might go ahead and add back in that edge. Therefore, the edge being withheld would not benefit computation and we might as well (from the computational perspective) have had that edge in the first place. If that edge was a useful edge to have because it improved the model accuracy and if the edge was removed only because of presumed improvement in computation, then this would have been a poor decision. In general, if one is interested in marginals corresponding to all factors (as one often is in machine learning contexts), and one wishes to do exact inference, then one might as well have started with a model where the computational algorithms do not “graduate” the model into a larger family. As we will see below, this family corresponds to those whose graph has a perfect elimination order, and this is also the class of triangulated graphs.

It is also important to note how the family changes when going from G to G_F . That is, $\mathcal{F}(G, \mathcal{M}^{(f)}) \subseteq \mathcal{F}(G_F, \mathcal{M}^{(f)})$. In fact, this is true for any set F even if it was not obtained from elimination. Therefore, we state and then prove this stronger theorem as follows:

Theorem 44. *Let $G = (V, E)$ be a graph with corresponding MRF family $\mathcal{F}(G, \mathcal{M}^{(f)})$. And let $F \subseteq V \times V$ be any set of pairs of nodes. Form a new graph $G_F = (V, E \cup F)$ by adding the pairs of nodes as edges to G to obtain G_F . Then $\mathcal{F}(G, \mathcal{M}^{(f)}) \subseteq \mathcal{F}(G_F, \mathcal{M}^{(f)})$.*

Proof. Take any $p \in \mathcal{F}(G, \mathcal{M}^{(f)})$. p factors w.r.t. the cliques in G . Take any clique C in G . Since G_F only has additional edges relative to G , C is a clique in G_F also but might be part of a larger clique. Therefore, any clique factor in G is either preserved in G_F or can be part of a larger factor in G_F , so p factors w.r.t. the cliques in G_F . \square

Since this is true for any F it is certainly true for the fill-in edges that we add during elimination.

Now, the key difference between the two orderings mentioned above is that in the first case, there were no fill-in edges added to the graph, while in the second case there was a fill-in edge added to the graph. Crucially, the largest clique in the original graph is of size 2, while in the second graph, the one with the

fill-in edge, the largest clique is of size 3. In fact, the number of nodes in the largest clique that is induced during the elimination process is the same as the exponent in the order of complexity. In general, when we are eliminating variables, the node that is currently being eliminated along with its neighbors that are subsequently fully connected constitute a clique, and we call this cluster of variables an *elimination clique*.

Definition 45 (elimination clique). *Given a graph $G = (V, E)$, a node elimination order $\sigma = (\sigma_1, \dots, \sigma_N)$, and a consequential series of elimination $(G_0, G_1, G_2, \dots, G_{N-1})$, the elimination cliques consist of a node v and its neighbors at the point it is eliminated, i.e., the set of cliques*

$$\bigcup_{i=1}^N \{\delta_{G_{i-1}}(\sigma_i)\} \quad (6.37)$$

The largest elimination clique size we encounter in the graphical elimination algorithm is equal to the largest dimensionality of the table size (exponent of r) during the summation process. That is, the complexity of eliminating all of the variables in a graphical model is exponential in the largest elimination clique. In general, therefore, a goal should be to find an elimination ordering that minimizes the size of the largest elimination clique in order to reduce the exponent of r .

We'll revisit this point later when we discuss generalized trees and arbitrary graphs, but for now there are two important points to be made from this discussion:

- The clique size of the resulting graph after elimination indicates the inherent complexity of the model.
- Chains are such that there is an obvious ordering that never adds any fill-in edges, namely, always eliminate one of the nodes at one of the two ends of a chain. Using such an order, the complexity is always $O(Nr^2)$.

In fact, trees in general have this property as well. Consider Figure 6.6. Lets say that for some computation (say we are interested in computing $p(x_3, x_4)$ where we are required to sum out variable x_1). If we were to eliminate node x_1 first, we would obtain the computation

$$\cdots \sum_{x_1} \psi_{1,2}(x_1, x_2) \psi_{1,5}(x_1, x_5) \psi_{1,7}(x_1, x_7) \psi_{1,9}(x_1, x_9) = \phi_{\gamma,2,5,7,9}(x_2, x_5, x_7, x_9) \quad (6.38)$$

any further computation would ultimately result in an $O(r^5)$ computation — x_1 is obviously a poor vertex to eliminate first.

On the other hand, consider the elimination order $(6, 5, 9, 8, 7, 1, 2)$. With this order, each summation never introduces any coupling of the factors, and correspondingly no edges are added to the tree when it is *reconstituted*. The reason for this is that each node, at the point that it is eliminated, is connected to one and only one neighbor.

From the perspective of summing out the variable in the equation for $p(x)$, at each elimination step, the sum of each variable can be moved to the right such that there is only one factor involved. That one factor only involves two variables, one of which is the variable being eliminated. After elimination, there is an additional factor involving only one variable which can be absorbed by any other factor involving that variable. Therefore, doing that summation in each case involves only an $O(r^2)$ computation.

From the perspective of graphical node elimination, each node when eliminated has only one neighbor, so no extra edges are added to the graph and the resulting re-constituted graph has zero fill-in edges. A *leaf node* in a tree is a node that has only one neighbor. Leaf nodes are also sometimes called *pendant* nodes. In fact, we see that any node order that only eliminates leaf nodes at each step in the tree is guaranteed to not add any edges in the graph. To make sure this is clear, after each elimination step, we have a reduced graph, a node that was formerly was not a leaf node might be a leaf node — we wish to eliminate nodes that at the time of elimination, in the corresponding partially reduced (or *current*) graph, is a leaf node.

The nice thing about trees is that there are always leaf nodes available to be eliminated as described by the following theorem.

Lemma 46. *A tree with more than one node always has at least two leaf nodes.*

Proof. Obviously true for $|V| = n = 2$ nodes. Assume true for $n - 1$ nodes and consider a tree with n nodes. The tree must have at least one leaf-node since if all nodes had two or more edges, we could find a cycle by traversing the nodes along the edges and marking the edges along the way — each node we encounter will either have an unmarked edge to allow the traversal to continue, or will have only marked edges implying the existence of a cycle, and eventually this latter condition will be reached since there are a finite number of nodes. The tree with $n - 1$ nodes induced by removing this leaf-node must itself have two leaf-nodes by induction, and at least one of those leaf-nodes is retained when adding back in the node to form the n -node tree. \square

Exercise 47. *Show that a graph is a tree iff every vertex-induced subgraph of the graph has at least two leaf nodes.*

Next, we note that eliminating a leaf node in a tree always yields a (sub-)tree since no fill-in is produced. In that sub-tree there will therefore always be two leaf-nodes as long as it has at least 2 nodes. Each time we eliminate a leaf-node, say i , that is connected to its neighbor $\{j\} = \delta(i)$, one of two things happen:

- Node j has only one neighbor other than i , so j becomes a new leaf node, and it is a candidate for elimination, or
- Node j has more than one neighbor other than i , but there still must be at least two other leaf nodes to eliminate in the sub-tree.

Eventually, node j will also become a leaf node, and it may at that time be eliminated.

Third, any elimination order in a tree that always eliminates next a leaf node in the current tree will never produce any fill in nodes (i.e., $F = \emptyset$) in the resulting reconstituted graph.

Fourth, it is easy to have a tree data-structure where the leaf-nodes are obvious. For example, in a node-adjacency list data structure, those nodes with only one neighbor are leaf nodes. When we eliminate a leaf node, we the neighbor of the node that has just been eliminated now only has one neighbor and if so add it to the set of current leaf nodes in the sub-tree.

Therefore, if we have a tree, we can easily choose an order of nodes to eliminate that is in some sense “perfect” since fill-in is never created — we always eliminate next a leaf-node in the current tree. Doing so will always yield a computation of the form $O(Nr^2)$.

Given that there is no ambiguity between an ordering of the nodes to be marginalized away and an ordering of the nodes to perform graph elimination, we can identify such an ordering using a permutation of $(1, 2, \dots, n)$.

6.3 Morphing from variable elimination to belief propagation

The elimination algorithm described above can be seen as a form of message passing procedure on a graph, where messages are passed along the edges of the graph. In this section, we look at this correspondence.

Consider the graph shown on the left in Figure 6.7 where we wish to compute $p(x_i, x_j)$ for an edge $(i, j) \in E(G)$. As is shown, we consider the tree to be “rooted” at the edge (i, j) — normally a tree is rooted at a single node but for this discussion we consider the tree to be rooted at an edge, specifically the two nodes i and j . Once a tree is rooted, we can talk about a nodes children and parents in the obvious way, except in this case both nodes i and j are at the root of the tree (i.e., they are neighbors but do not have

an ancestor/descendant relationship with respect to each other). Also we point out that there should be no confusion with parent/child relationships in a Bayesian network — here, we are talking about an undirected graph that happens to be a tree, and where we have designated an edge of the graph to be the root which automatically induces a directionality associated with the edges, where the directions are always pointing towards the root along the unique path to that root. This directionality does not change the semantics of the network (a Bayesian network uses the arrow directions to express families of models, so the arrow directions in Bayesian networks are used completely differently than the arrows we use here). Therefore, when we draw the graphs, the arrow directions will be drawn as arrows alongside the undirected edges of an undirected graph.

We start by eliminating leaf variables in the tree. This produces a sub-tree with new leaf variables which are next eliminated, and so on until we finally end up at the edge (i, j) at which point we have the marginal probability $p(x_i, x_j)$ that we need.

The figure shows an arrow along the edge for each variable that is eliminated. Again, the arrow does not correspond to a BN, rather the arrow is showing only the direction that the variables are being eliminated. We show the arrow $x_l \Rightarrow x_n$ if we eliminate a variable x_l that is a leaf node and its single neighbor at the time is x_n — a leaf node is determined relative to the *current* tree which might already have had some variables eliminated relative to the original tree. The arrows also place constraints on the order that variables must be eliminated. For example, if we see $x_5 \Rightarrow x_2 \Rightarrow x_8$ then this means that x_5 must be eliminated before x_2 which must be eliminated before x_8 . As was seen above, it is not required mathematically to eliminate variables in this order in producing $p(x_i, x_j)$, but doing so is computationally optimal since the order so displayed means that only leaf nodes at each step are eliminated.

Let us consider the arrows that are marked in blue in figure Figure 6.7 leading up to x_1 . The corresponding computations are:

$$\phi_{4,8}(x_8) = \sum_{x_{14}} \psi_{8,14}(x_8, x_{14}) \quad (6.39a)$$

$$\phi_{7,3}(x_3) = \sum_{x_7} \psi_{7,3}(x_7, x_3) \quad (6.39b)$$

$$\phi_{8,14,3}(x_3) = \sum_{x_8} \psi_{8,3}(x_8, x_3) \phi_{14,8}(x_8) \quad (6.39c)$$

$$\phi_{9,3}(x_3) = \sum_{x_9} \phi_{9,3}(x_9, x_3) \quad (6.39d)$$

$$\phi_{7,14,8,9,3,1}(x_1) = \sum_{x_3} \psi_{1,3}(x_1, x_3) \phi_{7,3}(x_3) \phi_{8,14,3}(x_3) \phi_{9,3}(x_3) \quad (6.39e)$$

$$\phi_{15,10}(x_{10}) = \sum_{x_{15}} \psi_{10,15}(x_{10}, x_{15}) \quad (6.39f)$$

$$\phi_{15,10,4}(x_4) = \sum_{x_{10}} \psi_{4,10}(x_4, x_{10}) \phi_{15,10}(x_{10}) \quad (6.39g)$$

$$\phi_{11,4}(x_4) = \sum_{x_{11}} \psi_{4,11}(x_4, x_{11}) \quad (6.39h)$$

$$\phi_{10,11,15,4,1}(x_1) = \sum_{x_4} \psi_{1,4}(x_1, x_4) \phi_{15,10,4}(x_4) \phi_{11,4}(x_4) \quad (6.39i)$$

More importantly, however, a pattern has emerged here — the pattern is that each node is receiving a form of *message* from its children in the rooted tree, and once it has received enough messages it may itself send out a message along the edge from itself to its parent. First, the blue arrows coming into a node in

Figure 6.7 indicate the message that has arrived at the node from its children. Then, for that node to send out a message, it uses the messages it has received, along with the potential relating it to the parent and destination of the message, and computes and “sends out” that message.

For example, consider node x_8 in Figure 6.7 and Equation (6.39a). Node x_{14} has no children to receive messages from (it is a leaf node in the original tree), and so it is ready to send a message to its parent which is formed by marginalizing away the factor between x_{14} and x_8 .

As a more complicated example, consider node x_3 in Figure 6.7 and Equation (??). Node x_3 receives messages from nodes x_7, x_8 (which has already received a message from node x_{14}), and x_9 and then relays those messages to node x_1 by forming a new message. This new message involves marginalizing away the messages x_3 received along with the factor function $\psi_{1,3}(x_1, x_3)$ and is shown diagrammatically in Figure 6.8, which is a blown-up version of the message passing around node x_3 in Figure 6.7.

In general, a node i may send a message to its parent j in the tree if i has received messages from all of i 's children. We can see this graphically as once i has received a message from its children (meaning all of i 's descendants have been eliminated from the elimination graph), that makes i a leaf node at which point it can be marginalized away.

The thing that determined the “parent” and “child” relationships in the undirected graph is the selection of the root edge. If we had chosen a different parent edge, then parent and child relationships might completely reverse. In general, however, for all possible root selections, the pattern we see is that when we want to send a message from i to j , we just have received messages from i 's other neighbors. This is known as the *message passing protocol*, it gives a partial ordering on the schedule of messages and we formalize this next.

Definition 48. Message passing protocol (MPP): A message may be sent from node i to a neighbor node j only after node i has received a message from all of its other neighbors.

Notationally, if $i \rightarrow j$ indicates a message from i to j , then the protocol may be written as $i \rightarrow j$ only when $\forall k \in \delta(i) \setminus \{j\}, \mu_{k \rightarrow i}(x_i)$. Note that any message i has received from j does not influence i 's ability to send a message to j . In order for i to send a message to j , i must have received a message from from all of i 's other neighbors, without needing to consider j . This means that i may send a message to j under two conditions, either:

- $\forall k \in \delta(i) \setminus \{j\}, k \rightarrow i$, or
- $\forall k \in \delta(i), k \rightarrow i$.

Note the second condition implies the first, but only the weaker (first) condition is required.

If the message passing protocol is followed but otherwise the ordering of the messages is arbitrary, then we are guaranteed that the end result will be the correct marginal. That is, the protocol specifies only a *partial* (rather than a total) order on messages.

As can be seen, this notation rapidly becomes long-winded, not to mention what would happen if we were to use it on a much larger graph with hundreds or even thousands of nodes. Therefore, rather than notationally keep track of the entire history of all sub-tree nodes that have been eliminated in the formulation of a message, as we have done above using constructs like $\phi_{15,10,4}(x_4)$, it is much more convenient to use a notation that indicates only the neighboring nodes that are participating in a message. We use $\mu_{i \rightarrow j}(x_j)$ to indicate a message coming from node i going to node j along the edge (i, j) and which is a function only of x_j (since x_i has been eliminated). In this new notation, Equation (6.39a) becomes:

$$\mu_{14 \rightarrow 8}(x_8) = \sum_{x_{14}} \psi_{8,14}(x_8, x_{14}) \quad (6.40)$$

and Equation (6.39d) becomes:

$$\mu_{3 \rightarrow 1}(x_1) = \sum_{x_3} \psi_{1,3}(x_1, x_3) \mu_{7 \rightarrow 3}(x_3) \mu_{8 \rightarrow 3}(x_3) \mu_{9 \rightarrow 3}(x_3) \quad (6.41)$$

The general form of a message is as follows: is:

$$\mu_{i \rightarrow j}(x_j) = \sum_{x_i} \left(\psi_{i,j}(x_i, x_j) \prod_{k \in \delta(i) \setminus \{j\}} \mu_{k \rightarrow i}(x_i) \right) \quad (6.42)$$

which indicates that a message is being passed from node i to node j , and it includes all incoming messages that have come in at node i except for the one that came in from node j . The message is always a function of the destination variable x_j , due to the selection $\delta(i) \setminus \{j\}$, as we do not include the message sent from j to i when sending a message from i to j . In some sense, j already knows about the information contained in the message it sends to i so it does not need to be reminded of that information. In the context that it is given here j is the parent variable in the currently rooted tree, but we will soon see that this message as given is general even when the tree is not rooted. The message is formed as follows:

1. First, collect messages from all neighbors of i other than j ,
2. next, incorporate these incoming messages by multiplying them in along with the factor $\psi_{i,j}(x_i, x_j)$,
3. the factor $\psi_{i,j}(x_i, x_j)$ relates x_i and x_j , and can be seen as a representation of a “communications channel” relating how the information x_i transforms into the information in x_j , thus motivating the terminology of a “message”, and
4. then finally marginalizing away x_i thus yielding the desired message to be delivered at the destination node x_j .

For example, the messages shown in Figure 6.8 are shown using this new notation in Figure 6.13. Note that when we have a tree like the above, x_j is x_i 's parent, and $\delta(i) \setminus \{j\}$ is i 's children, but we will see below that the construct will be more general than this, so we do not specify any directionality inherently associated with the graph in Equation (6.42) — rather, directionality is associated only with the direction of the message.

Also, messages of the form $\mu_{i \rightarrow j}(x_j)$ do not indicate the entire history of variables that have been eliminated before x_i unlike the notation we used earlier which would be something like $\phi_{\mathcal{M}, \mathcal{U}, i, j}(x_j)$. The indication of the history is not necessary since it is already implicitly defined by: 1) the tree, 2) the specification of a root in the tree, and 3) the message passing protocol which restricts when messages can be sent. That is, we are unable to send a message from i to j unless i has received a message from all its children, which means that all variables below i in the tree have been eliminated in their appropriate order. Moreover, the specific ordering has no mathematical effect as long as MPP has been respected — there could be many such orders.

Another reason for using this notation, as we will see later, is that for more general non-tree graphs, we may wish to construct and send a message $\mu_{i \rightarrow j}(x_j)$ without all previous messages being sent, or it might not even be well-defined what the notion of a “previous message” would be. This corresponds to loopy belief propagation (graphs that are not trees), as discussed in Section XXXX.

Before we move on, we consider further aspects of MPP.

MPP induces a partial order on the messages that might be sent, rather than a total order. Therefore, there is quite a bit of flexibility in the message orderings that are prescribed by MPP. According to MPP, we can always send a message from a leaf-node in the tree to its (necessarily single) neighbor, since a leaf node

has no other neighbors ($\delta(i) \setminus \{j\} = \emptyset$). Therefore, MPP requires that we start sending messages at the leaf nodes.

Each time we send a message from a leaf-node to its neighbor, this is equivalent to eliminating that leaf node in the graph, and the resulting remaining graph will still be a tree and will have at least two leaf nodes. Therefore, MPP will allow further messages to be sent. I.e., the notion of a node i having received a message from all $k \in \delta(i) \setminus \{j\}$ means that all nodes $k \in \delta(i) \setminus \{j\}$ have already been eliminated, and by transitivity, all subtrees rooted at such k have also been eliminated. Therefore, the MPP ensures that each time we send a message from a node to its parent in the graph, we are doing the equivalent of eliminating a leaf node in the graph which means each message will only cost $O(r^2)$.

If we were to not abide by MPP, what would be the consequences? Consider, for example, if we were to send message D before message A in Figure 6.8. We know that in order to be mathematically correct, we can not send the sum farther into the set of factors than possible thereby bypassing some factor that involves x_3 . That is, we are **not** allowed to perform a computation of the form:

$$\phi_{7,14,8,9,3,1}(x_1) = \sum_{x_3} \psi_{1,3}(x_1, x_3) \phi_{8,14,3}(x_3) \phi_{9,3}(x_3) \quad (6.43)$$

which doesn't involve the factor $\phi_{7,3}(x_3)$ since it is not available yet (it is only available after we have eliminated x_7 , equivalently sent message A). Therefore, the only valid way to interpret sending message D before message A would be in the following computation:

$$\phi_{14,8,9,3,7,1}(x_7, x_1) = \sum_{x_3} \psi_{7,3}(x_7, x_3) \psi_{1,3}(x_1, x_3) \phi_{8,14,3}(x_3) \phi_{9,3}(x_3) \quad (6.44)$$

$$(6.45)$$

which is a function of both x_7 and x_1 . We see immediately that this leads to a computation with cost $O(r^3)$ and, graphically, corresponds to eliminating node x_3 and inducing a fill-in edge between x_7 and x_1 , thus producing a clique of size 3 in the reconstituted graph. When we obey the message passing protocol, therefore, we are ensuring that we eliminate at each step only leaf nodes in the reduced graph so that the clique size is always 2 and no additional fill in edges are added. MPP therefore ensures that our computation is not only correct, but also is computationally optimal.

Exercise 49. Consider Figure 6.12, where we considered several possible message orders in order to reach a state where every edge had messages across it in both directions. In all three cases, the messages obeyed the MPP. In the left and middle case, there was a clear root. Is it the case that, whenever MPP is followed, and the result is that all edges have at least one message across it in both directions, then there must be some implied root? If so, prove it. Otherwise, give a counterexample.

6.4 Multiple Tree Queries

In the previous section, we considered only resulting queries of the form $p(x_S)$ for some S — in particular, S was one of the cliques (nodes adjacent to an edge) in the tree. For many problems of interest (e.g., most statistical and/or machine learning problems) there is a collection of non-disjoint subsets of V for which we wish to compute this quantity. I.e., we might have a set $\{S_1, S_2, \dots, S_k\} = \mathcal{S}$ and we wish to compute

$$p(x_{S_i}) \text{ for all } i \in \{1, 2, \dots, k\} \quad (6.46)$$

For example, with EM or gradient-based learning, we need to be able to compute expected sufficient statistics of the form

$$E[f(X_S)] = \sum p(x_S) f(x_S)$$

for all S where S is a clique in G . In our current case, we wish to compute all $p(x_i, x_j)$ where $(i, j) \in E(G)$.

One way to solve this problem is to run the elimination algorithm above k times, one for each S_k . That is, for each k , compute:

$$p(x_{S_k}) = \sum_{x_{V \setminus S_k}} p(x) \quad (6.47)$$

and we would find an appropriate elimination order for each query, resulting in a $O(kNr^2)$ computation.

If each of the sets S_k consists only of one edge in the graph, in which case $k = N - 1$, this computation can be reduced to $O(Nr^2)$ using dynamic programming (recall, for an N -node tree, there are always $n - 1$ edges). This means that this computation for all k queries does not depend on k and in fact is the same cost as if we were to do only one query.

How can this be possible? Clearly, it must be due to the multiple k computations above having a large amount of redundancy — in fact, it must be that there is a factor of k redundancy so that the factor of k can be removed in $O(kNr^2)$. We will see below that this is a property of all graphs to a certain extent if the sets $\{S_k : k\}$ consist of all maximal cliques in the graph (or are subsets of maxcliques), but for the immediate discussion we focus on trees.

We start our discussion by considering a second edge $(1, 3) \in E(G)$, and the goal is to compute both $p(x_1, x_2)$ and $p(x_1, x_3)$. A poor way to perform the computation would be to pay no attention to the fact that we just computed $p(x_1, x_2)$, and to root the tree at $(3, 1)$ and eliminate leaf nodes on up the tree until we reach $(3, 1)$ as shown on the left in Figure 6.10.

Lets first think about this in terms of a variable elimination order. For $p(x_1, x_2)$, the variable elimination ordering $(14, 7, 8, 9, 15, 10, 11, 4, 12, 13, 5, 6, 3)$ would suffice since we are always eliminating only leave nodes, leaving at the end the desired residual marginal nodes 1 and 2. This elimination order corresponds to computing and sending 13 messages: $\mu_{14 \rightarrow 8}(x_8), \mu_{7 \rightarrow 3}(x_3), \mu_{8 \rightarrow 3}(x_3), \mu_{9 \rightarrow 3}(x_3), \mu_{15 \rightarrow 10}(x_{10}), \mu_{10 \rightarrow 4}(x_4), \mu_{11 \rightarrow 4}(x_4), \mu_{4 \rightarrow 1}(x_1), \mu_{12 \rightarrow 6}(x_6), \mu_{13 \rightarrow 6}(x_6), \mu_{5 \rightarrow 2}(x_2), \mu_{6 \rightarrow 2}(x_2)$, and $\mu_{3 \rightarrow 1}(x_1)$. For the second quantity $p(x_1, x_3)$, the variable ordering $(14, 7, 8, 9, 15, 10, 11, 4, 12, 13, 5, 6, 2)$ would suffice, which corresponds to messages: $\mu_{14 \rightarrow 8}(x_8), \mu_{7 \rightarrow 3}(x_3), \mu_{8 \rightarrow 3}(x_3), \mu_{9 \rightarrow 3}(x_3), \mu_{15 \rightarrow 10}(x_{10}), \mu_{10 \rightarrow 4}(x_4), \mu_{11 \rightarrow 4}(x_4), \mu_{4 \rightarrow 1}(x_1), \mu_{12 \rightarrow 6}(x_6), \mu_{13 \rightarrow 6}(x_6), \mu_{5 \rightarrow 2}(x_2), \mu_{6 \rightarrow 2}(x_2)$, and $\mu_{2 \rightarrow 1}(x_1)$. It is crucial to realize that the first 12 variables in each order are identical, namely $(14, 7, 8, 9, 15, 10, 11, 4, 12, 13, 5, 6)$. Of course, the first 12 messages in each set of messages are also identical. Either results in the marginal $p(x_1, x_2, x_3)$. This means that if we have eliminated those nodes in computing $p(x_1, x_2)$ then we should utilize that work done for the second quantity $p(x_1, x_3)$.

Perhaps an even easier way to see this reuse is to look at the two trees that have been annotated with arrows indicating the elimination steps (equivalently the messages $\mu_{i \rightarrow j}(x_j)$ that have been passed in the graph). When we look at the messages that are required to compute the marginal at edge $(1, 2)$ (shown in blue) and also the messages that are required to compute the marginal at edge $(1, 3)$ (shown in red) as given on the right of Figure 6.10, we can see from the figure that almost all of the messages (or elimination steps) are the same, which is shown as an edge annotated with two different color arrows in the same direction. Therefore, given the portions of the computation needed to compute $p(x_1, x_2)$, computing $p(x_1, x_3)$ takes only a few more steps, and vice versa.

Depending on the ultimate quantities we wish to compute, different portions of the partial accumulations may be used in different ways. It is not always the case that the computations will be as redundant as the two mentioned above. For example, consider computing both $p(x_8, x_{14})$ and $p(x_6, x_{13})$. In this case, both computations may start with the elimination order $(7, 9, 15, 10, 11, 4, 5)$ corresponding to messages: $\mu_{7 \rightarrow 3}(x_3), \mu_{9 \rightarrow 3}(x_3), \mu_{15 \rightarrow 10}(x_{10}), \mu_{10 \rightarrow 4}(x_4), \mu_{11 \rightarrow 4}(x_4), \mu_{4 \rightarrow 1}(x_1), \mu_{5 \rightarrow 2}(x_2)$, and $\mu_{12 \rightarrow 6}(x_6)$. From the perspective of elimination, this leaves a chain $x_{14}, x_8, x_3, x_1, x_2, x_6, x_{13}$. From the perspective of redundant computation, computing the marginals only on both ends of a chain has the least redundancy of all possible pairs of marginal queries on a chain. Therefore, none of the remaining messages in the direction

from x_{14} to x_{13} and in the direction from x_{13} back to x_{14} may be reused. However, we certainly have saved potentially quite a bit by “trimming” off the portion of the tree in both cases that lead to the final chain.

There is further good news, however. Namely, as the number of required computations k increases, the potential for reuse also increases since there is a greater likelihood that some subset of the computations will require the same partial work. For example, suppose now that we wish to compute the marginals on *all* edges $p(x_i, x_j)$ for $(i, j) \in E(G)$. As mentioned above, this is the case that is often needed when wishing to learn the parameters of the model, namely we wish to compute $p(x_C)$ for all cliques $C \in \mathcal{C}$ in the graph. In the case of the tree, all cliques are pairs of nodes along an edge, so here $k = N - 1$. We claim that it is possible to compute all such marginals in $O(Nr^2)$ time, completely avoiding the extra factor of k .

Consider rooting the tree at all edges $(i, j) \in E(G)$ in turn, and sending the messages up to edge (i, j) and marking the edge with an arrow, as we have done, to indicate the direction of the message. When messages are redundant, rather than marking the edge with two arrows, we instead mark it with only one. More generally, if a message is required one or more times, it gets marked only with a single arrow. It is important to realize the definition of the message, which is given in Equation (6.42), keeping in mind that an outgoing message requires the incoming messages from all *other* neighbors. Once this is done, we will have a tree where each edge is marked with two messages, one in each direction. At that point, there is no reason to consider any edge as the root of the tree (all nodes could be considered a root), so we can thus re-draw the tree with all edges marked with both message directions as done in Figure 6.11.

It is clear that as long as the messages have been sent in such a way that the message passing protocol has been followed, each edge $(i, j) \in E(G)$ is now in possession of $\psi_{i,j}(x_i, x_j)$ as well as $\mu_{k \rightarrow i}(x_i)$ for all $k \in \delta(i) \setminus \{j\}$ as well as $\mu_{k \rightarrow j}(x_j)$ for all $k \in \delta(j) \setminus \{i\}$. With these quantities, we can compute the marginals for each edge as follows:

$$p(x_i, x_j) = \psi_{i,j}(x_i, x_j) \prod_{k \in \delta(i) \setminus \{j\}} \mu_{k \rightarrow i}(x_i) \prod_{k \in \delta(j) \setminus \{i\}} \mu_{k \rightarrow j}(x_j) \quad (6.48)$$

Since only $2(N - 1)$ messages were sent, each costing $O(r^2)$, the overall computation is $O(Nr^2)$. We moreover, have the following theorem.

Theorem 50. *Given a tree $G = (V, E)$ and some $p \in \mathcal{F}(G, \mathcal{M}^{(f)})$, if messages are sent obeying the message passing protocol so that all edges have two messages across them in each direction, then the computation given in Equation (6.48) will correctly produce all marginals for all edges in $E(G)$.*

Proof. Consider any edge $(i, j) \in E(G)$ and consider rooting the graph at that edge, as described above. Since all messages obey the MPP, the messages correspond to eliminating the variables in an order from leaf to root, which precisely gives $p(x_i, x_j)$. \square

When considering the nodes, it is also the case that all messages coming into a node constitute the node marginals. That is, we have, for all $i \in V(G)$,

$$p(x_i) = \prod_{k \in \delta(i)} \mu_{k \rightarrow i}(x_i) \quad (6.49)$$

The proof of this is identical to that of Theorem ???. In some sense, when G is a tree, each node $i \in V(G)$ “separates”, “splits”, or perhaps “shatters” the tree into multiple disconnected sub-trees, and considering each i as a root, each sub-tree may send messages to their root, corresponding precisely to the elimination algorithm starting at the leaf-nodes relative to i and eliminating only leaf nodes on the way up to the final variable x_i .

So now we know which messages are to be sent, when, and how, but we need an algorithm that sends these messages in their appropriate MPP abiding order. This is actually quite simple. First, we choose an

arbitrary node in $V(G)$ to be the root of the tree (we are choosing a node as the root, unlike above where we choose an edge). Next, we send all messages starting from the leaves up this root — sending messages in such a way guarantees that there will always be a message that can be sent that obeys the message passing protocol. Once all messages have been sent to the root, the root may start sending messages out to its children, which in turn may themselves start sending messages out to their children, and so on, until messages are propagated back to the leaves of the tree. See Figure 6.12. When done, all nodes will have received messages from all neighbors, and we may thus form the marginals as shown in Equation 6.48.

Of course, this is only an outline of an algorithm, we still need to choose which messages to send when, and we need to have two phases, one where we propagate messages up to the root, and another where we propagate messages back down to the leaves. We will call these two phases *collect evidence* and *distribute evidence* respectively and both are easy to define using recursive algorithms. The collect evidence procedure propagates messages from the leaves of the tree to the root of the tree by performing an post-order traversal of the tree, finishing all messages from the children to the parent before the parent sends a message to its parent. Distribute evidence does a pre-order tree traversal, sending messages from the parent to all children, before each child sends a message to its children.

Algorithm 3: CollectEvidence($c \rightarrow p$)

Input: A rooted tree $G = (V, E)$ with a child node $c \in V$ and its parent $p \in V$.

Result: A message propagated from c to p that obeys the message passing protocol.

- 1 **foreach** $u \in \text{child}(c)$ **do**
- 2 | call CollectEvidence($u \rightarrow c$)
- 3 Compute

$$\mu_{c \rightarrow p}(x_p) = \sum_{x_c} \psi_{c,p}(x_c, x_p) \prod_{u \in \text{child}(c)} \mu_{u \rightarrow c}(x_c)$$

Algorithm 4: DistributeEvidence($p \rightarrow c$)

Input: A rooted tree $G = (V, E)$ with a parent node $p \in V$ and a child $c \in \text{child}(p)$.

Result: A message propagated from p to c that obeys the message passing protocol.

- 1 Compute
 - 2 **foreach** $u \in \text{child}(c)$ **do**
 - 3 | call DistributeEvidence($c \rightarrow u$)
-

It is important to notice that the subroutines above define a $\text{child}()$ function, and also talk about parents children — these are all implicitly defined in graph G only once some node is designated the root, as is done on line 5 in Algorithm 5.

We can easily see that all messages obey the message passing protocol. At the collect evidence stage, a message is not sent to a node's (single) parent until it has received messages from all its children, so there is only one node it has not yet received a message from, namely the parent. At the distribute evidence stage, once a node has received a message from its parent, it has received a message from all of its neighbors (since it received a message from all its children earlier, during the collect evidence phase) so it is free to send a message to any child that it likes.

We reflect for a moment on why the algorithm is called collect and distribute *evidence* which may seem

Algorithm 5: CollectDistributeEvidence

Input: A tree graph $G = (V, E)$

Result: All messages propagated between all pairs of nodes so that we may compute the marginals on all edges $(i, j) \in E(G)$ as shown in Equation 6.48.

- 1 Designate an arbitrary node $r \in V$ as the root.
 - 2 **foreach** $c \in \text{child}(r)$ **do**
 - 3 | call CollectEvidence($c \rightarrow r$)
 - 4 **foreach** $c \in \text{child}(r)$ **do**
 - 5 | call DistributeEvidence($r \rightarrow c$)
-

curious since evidence is not explicitly mentioned in the algorithms. Recall as we mentioned in Section 6.1, each of the factors might have had delta functions attached, which provide a link to the evidence. In such a case, the result of Equation 6.48 in each case is actually not the marginal $p(x_i, x_j)$ but is that marginal joint with the evidence nodes E , or $p(x_i, x_j, \bar{x}_E)$ since this is precisely what the elimination algorithm would yield at each edge. Recall also that $p(x_i, \bar{x}_i) = \delta(x_i, \bar{x}_i)p(\bar{x}_i)$. The reason for this is described in Chapter EVIDENCE.

Furthermore, this can easily be turned into $p(x_i, x_j | \bar{x}_E)$ by normalizing over each resulting edge potential

$$p(x_i, x_j | \bar{x}_E) = \frac{p(x_i, x_j, \bar{x}_E)}{\sum_{x_i, x_j} p(x_i, x_j, \bar{x}_E)} \quad (6.50)$$

We note that the running time of the entire procedure is $O(Nr^2)$. The reason is that a message is calculated only twice for each edge in the tree, there are $N - 1$ edges in the tree, and that each message itself costs $O(r^2)$.

Are there other orderings possible? Yes, in fact it might be useful to perform a breadth-first search on the tree and then list all messages at a given tree depth simultaneously. The reason for this would be that each message is entirely separate from another, involving the least number of common nodes. In such case, it would be relatively easy to schedule each message on a separate processor, thereby leading to a simple parallel implementation of the algorithm. Of course, the amount of parallelism would diminish as we move up the tree, ultimately leading to only a single processor computing the messages into the root, but this is only a relatively short portion of the amount of time spending messages. In such case, assuming unlimited numbers of processors available, we would wish to choose the root that would lead to the shortest depth tree.

6.5 Alternate forms of messages - speed/memory issues

In the above discussion, we have considered one particular style of message propagation in a tree graph that has certain implications regarding the implementation, its speed, and memory requirements. Take as an example Figure 6.13 where we see that not only do we require storage for the table at edge $(3, 1)$ to store the factor $\psi_{3,1}(x_3, x_1)$ but for each incoming message at node x_3 we need to store the message and then use it once we are ready to send message $\mu_{3 \rightarrow 1}(x_1)$. Since the messages sent back to node x_7, x_8 , and x_9 are all different (i.e., each of them respectively does not include the message that came to x_3 from that node), each incoming message corresponding to x_3 's neighbors needs to be stored in a separate one-dimensional table so as to be able to appropriately combine them for whenever we compute an outgoing message.

More generally, when considering the message definition as in Equation (6.42), we need to have storage for a one-dimensional table for each node's neighbors. These tables store the incoming messages when

they come in. As the degree the nodes increase, the message storage required will correspondingly linearly increase.

An alternative approach would be to incorporate (i.e., multiply in) and then forget the message as soon as it arrives. We already know that all edge factors can hold node factors, so that when a message arrives, say $\mu_{k \rightarrow i}(x_i)$ we could incorporate (multiply in) and then forget that message as soon as it arrives without requiring additional permanent storage. We incorporate the message by multiplying it directly into the table $\psi_{i,j}(x_i, x_j)$, assuming that $k \in \delta(i) \setminus \{j\}$. The effect of new message would thus be:

$$\psi'_{i,j}(x_i, x_j) \leftarrow \psi_{i,j}(x_i, x_j) \mu_{k \rightarrow i}(x_i) \quad (6.51)$$

Such an update of the factor at the edge could occur at the arrival of every message as long as they arrive from nodes $k \in \delta(i) \setminus \{j\}$. The final factor, after multiplicatively updating the edge table with all the incoming messages at node i , would be of the form

$$\psi'_{i,j}(x_i, x_j) \leftarrow \psi_{i,j}(x_i, x_j) \prod_{k \in \delta(i) \setminus \{j\}} \mu_{k \rightarrow i}(x_i) \quad (6.52)$$

at which point we could send the message to node j as follows:

$$\mu_{i \rightarrow j}(x_j) = \sum_{x_i} \psi'_{i,j}(x_i, x_j). \quad (6.53)$$

Such an approach is advantageous in that it never require any storage associated with the nodes of the graph. Moreover, this can be quite useful for certain queries. For example, for computing **just** $p(x_i)$ for a particular node i , or $p(x_i, x_j)$ for a particular edge $(i, j) \in E(G)$, this works out well — consider Figure 6.14. As messages come into a node i , they are incorporated directly into the edge factor (i, j) of the edge where j is in the direction towards the root. Once messages have reached nodes 1 and 2, all node marginals required for computing $p(x_1, x_2)$ are available.

On the other hand, for most queries we are interested in, there is a potential problem which is that now the factor over the edge has been modified. At some point, messages will start arriving at x_j via nodes $k \in \delta(j) \setminus \{i\}$. If we were to use the updated (i, j) -edge table at this point and then send a message back to i , we would no longer be equivalent to the aforementioned elimination procedure since the messages that come into i via $\delta(i) \setminus \{j\}$ would be included in what is being sent back out from i to $\delta(i) \setminus \{j\}$. Recall our discussion of Equation (6.42) — when sending a message from i to j we do not include the message from j back to i since j and its attached sub-tree already knows this information. If we were to use the updated edge table, we would be sending back to j information that it already knows, and this would lead to it being double counted.

Mathematically, from the elimination perspective, this would be equivalent to squaring the marginal functions after they have been constructed (i.e., ϕ^2 rather than ϕ).

In order for the computation to be correct, therefore, the messages that came in to i from before must be “divided out” in some way, but at this point we would not know what to divide out since the former incoming messages have been multiplied into the edge table and are no longer available (since we have not allocated storage for them). Furthermore, we still want to keep the node storage at least bounded regardless of node degree.

6.5.1 Dividing out node functions from edge functions

One solution to this problem that does not require any node storage at all would be to divide out the outgoing message from an edge as soon as it is ready, so that when it comes back and is multiplied back in, the double counting is canceled out. Suppose that we are computing a message $i \rightarrow j$ towards a root node (so that we

are in the first phase of message passing, corresponding to the collect evidence procedure), and that $k \in \delta(j)$ is the neighbor of j one step closer to this root. The equations in the following algorithm give the message $\mu_{i \rightarrow j}(x_j)$ and then table updates for both the outgoing edge (j, k) and the source edge (i, j) .

Algorithm 6: First phase message update $\mu_{i \rightarrow j}(x_j)$

```

1  $\mu_{i \rightarrow j}(x_j) = \sum_{x_i} \psi_{i,j}(x_i, x_j) \prod_{k \in \delta(i) \setminus \{j\}} \mu_{k \rightarrow i}(x_i);$            /* message as normal */
2  $\psi'_{i,j}(x_i, x_j) \leftarrow \psi_{i,j}(x_i, x_j) / \mu_{i \rightarrow j}(x_j);$            /* table update - divide outgoing message out */
3 if  $j$  is not the root then
4   Let  $k \in \delta(j)$  be the neighbor of  $j$  towards the root;
5    $\psi'_{j,k}(x_j, x_k) \leftarrow \psi_{j,k}(x_j, x_k) \mu_{i \rightarrow j}(x_j);$            /* table update - multiply in incoming message */

```

By dividing out $\mu_{i \rightarrow j}(x_j)$ from $\psi_{i,j}(x_i, x_j)$, we are sure that the $\mu_{i \rightarrow j}(x_j)$ will not be double counted once it is multiplied back in from the message coming back from k in $\mu_{k \rightarrow j}(x_j)$. Obviously, once we reach the root of the tree, the above updates need to change as there is no longer an outgoing edge towards the root, and this is accounted for in the algorithm. Once all messages have been received at the root node, we may start sending messages back towards the leaves, and this is done using the normal definition of a message (Equation (6.42)). In each case, the marginal that was divided out in line 2 of the algorithm above will be multiplied back in and thus will be counted exactly one time. Thus, each edge will provide the appropriate marginal $p(x_i, x_j)$ for $(i, j) \in E(G)$. Note that the message schedule can be the same as any MPP abiding scheme – the only thing that is different is that the message definition needs to change depending on the phase of the message passing scheme: the first phase is the case before the root node has received messages from all of its neighbors, and the second phase is after the root node has received messages from all of its neighbors. Moreover, note that we can no longer send redundant messages along an edge. For example, message 7 after message 3 in Figure 6.12-right will no longer be valid since the marginal $\mu_{15 \rightarrow 10}(x_{10})$ will be divided out twice rather than once. Of course, such redundant messages are often not useful anyway.

Note that the new scheme of message passing is asymmetric, in that we do different messages during the collect vs. the distribute evidence phase of message passing. Our original scheme (Equation (6.42)) is perhaps simpler since it required only one type of message. Another drawback of this new scheme is that it requires division in the edge table (line 2). This will require $O(r^2)$ divisions in the worse case (or $O(r)$ divisions and $O(r^2)$ extra multiplications).

Exercise 51. It is often the case that as soon as division is used, we need to concern ourselves with the potential for divide by zero. Suppose distribution we are working with is not strictly positive, so that some of the messages might have zeros. Can you explain why this does not inherently cause problems in the message update equations? How would you modify line 2 in the above algorithms to deal with potential divide-by-zero issues?

The first form of message used a uniform message definition, and it did not require a two-phase collect/distribute evidence with respect to some root.

6.5.2 Maintaining distinct node separator functions

There is a third form of message that in some sense combines the above two forms. In this message, we once again utilize a table for each pair of edges that share a common node. That is, for each edge pair $(i, j) \in E(V)$ and $(j, k) \in E(V)$, we introduce a new node in the graph. This is depicted as a square node in Figure 6.15, where a unique square node is associated with each distinct pair of edges sharing a common node. Normally, each node in a standard tree separates (or shatters) the tree graph into two or more separate sub-trees. We note that each of these square nodes also corresponds to a separator in the original tree-graph, except that in this case, each square-node separator shatters the tree into precisely two (no more and no less)

separate components. For each of these square separator nodes, we store two extra tables $\phi_{i,j,k}(x_j)$ and $\phi'_{i,j,k}(x_j)$. For this reason, we call $\phi_{i,j,k}(x_j)$ and $\phi'_{i,j,k}(x_j)$ *separator potentials*, where x_j separates the tree into two sub-trees between edges (i, j) and (j, k) . The algorithm then follows the collect/distribute evidence message schedule, except that in this case the tree, once again, is rooted at an edge rather than a node. Before this begins, node tables are initialized to unity, $\phi_{i,j,k}(x_j) = \phi'_{i,j,k}(x_j) = 1$ for all $x_j \in D_{X_j}$. During the collect evidence phase of the algorithm, the messages take the following form, which are different depending on if we are in the collect-evidence or the distribute evidence form.

Algorithm 7: collect evidence message update $\mu_{i \rightarrow j}(x_j)$

- 1 $\phi_{i,j,k}(x_j) = \sum_{x_i} \psi_{i,j}(x_i, x_j);$ /* message as normal stored in node */
 - 2 $\psi_{j,k}(x_j, x_k) \leftarrow \psi_{j,k}(x_j, x_k) \phi_{i,j,k}(x_j);$ /* update (j, k) edge potential. */
-

Line 1 in Algorithm 7 compute and then stores the marginal into the separator potential between the two edges. We note that $\phi_{i,j,k}(x_j) = \mu_{i \rightarrow j}(x_j)$. Line 2 in Algorithm 7 multiplies in the separator potential to the outgoing edge (j, k) . Therefore, we must once again ensure that there is no double counting of $\phi_{i,j,k}(x_j)$ when we do the distribute evidence phase, which is given in the next algorithm.

Algorithm 8: distribute evidence message update $\mu_{i \rightarrow j}(x_j)$

- 1 $\phi'_{i,j,k}(x_j) = \sum_{x_i} \psi_{i,j}(x_i, x_j);$ /* message as normal stored in node */
 - 2 $\psi_{j,k}(x_j, x_k) \leftarrow \psi_{j,k}(x_j, x_k) \frac{\phi'_{i,j,k}(x_j)}{\phi_{i,j,k}(x_j)};$ /* update (j, k) edge potential. */
-

Line 1 in Algorithm 8 is just like the previous algorithm. Line 2, however, is where the double counting is avoided, since we divide out the previous separator potential table when we update the (j, k) edge function. Therefore, when all messages have been passed across both directions of the edges, we have the proper marginals.

The advantage of this strategy is that it can at least be made to appear to have uniform type of message. For example, line 2 in each of the algorithms could divide out the “other” separator potential values. Algorithm 8 already does this, as it divides out $\phi_{i,j,k}(x_j)$. Algorithm 7 could also do a similar operation, in that it could divide out $\phi'_{i,j,k}(x_j)$ which at that time would not have any effect since we initialized $\phi'_{i,j,k}(x_j)$ to be unity everywhere. On the other hand, we have once again used more storage associated with the nodes, two tables per edge pair. We can of course reduce cleanliness and use only one tables by using the following distribute evidence algorithm:

Algorithm 9: asymmetric distribute evidence message update $\mu_{i \rightarrow j}(x_j)$

- 1 **foreach** $x_j \in D_{X_j}$ **do**
 - 2 $\phi_{i,j,k}(x_j) \leftarrow \frac{1}{\phi_{i,j,k}(x_j)} \sum_{x_i} \psi_{i,j}(x_i, x_j);$ /* message as normal stored in node */
 - 3 $\psi_{j,k}(x_j, x_k) \leftarrow \psi_{j,k}(x_j, x_k) \phi_{i,j,k}(x_j);$ /* update (j, k) edge potential. */
-

The three different message styles we have described are called, respectively, the Shenoy-Shafer, the Lauritzen-Speigelhalter, and the Hugin message passing strategies. Normally, they are described in the context of junction-trees rather than standard trees as we have done above. They are nonetheless perfectly well defined in our context here. Moreover, having some understanding of them here will assist in their development below when we discuss junction trees.

Lastly, we note that these different message passing schemes may on the one hand be considered mere implementation issues as they have no effect on either the mathematical accuracy or the inherent complexity of the procedures. On the other hand, they can have real-world practical consequences related to: 1) redundancy of repeatedly re-multiplying incoming messages, especially if the degree of the graph and/or the cardinality of the nodes is very large; and 2) the existence of real division in the algorithm and/or the relief given by log-arithmetic implementations; 3) the amount of memory associated with each algorithm, in either case regarding what needs to be stored for each graph node.

Discuss historical aspects of this, including the few papers in the 1990s that compare implementations, and why they did this. Mention that this was done in the JT case that we have not yet defined, but it is beneficial to visit this issues right now, in the context of inference on simple trees since it is quite easy to understand in this context.

6.5.3 Message passing as state-space traversal

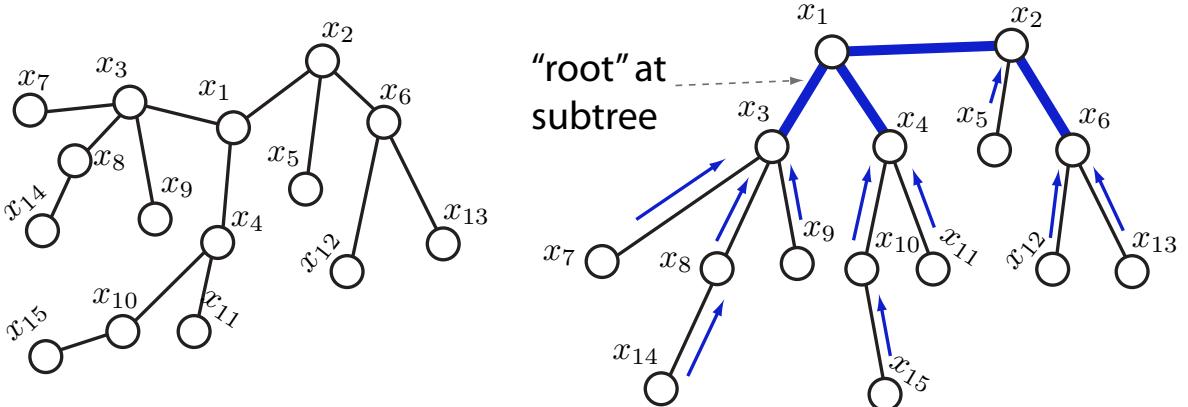
6.6 Tree Queries with arbitrary S

So far, we have described the case where the marginal query we wish to compute $S = (i, j)$ is one or more of the edges in a tree, but this is by no means the full set of queries we might wish to compute. Instead, we might require S as an arbitrary set of two or more nodes, and this can have a significant effect on the computation.

For example, suppose we have the simple 4-node Markov chain: $G = x_1 - x_2 - x_3 - x_4$ along with $p(x_1, x_2, x_3, x_4) \in \mathcal{F}(G, \mathcal{M}^{(f)})$. If our goal is to compute $p(x_1, x_2, x_3)$, then we can eliminate x_4 in

$$\sum_{x_4} \psi_{1,2}(x_1, x_2) \psi_{2,3}(x_2, x_3) \psi_{3,4}(x_3, x_4) \quad (6.54)$$

leading to an $O(r^2)$ computation since we have eliminated only a leaf node in the chain. In the general case, if S is a sub-tree in $G = (V, E)$ then can do the same trick above — the resulting $p(x_S)$ can be obtained with the same $O(r^2)$ computation “rooting” the tree at the subtree S .



In the above figure, $S = \{1, 2, 3, 4, 6\}$ induces a sub-tree in G , so all messages are sent towards nearest node inside of S starting from the leafs of each sub-tree that has been shattered by S . Once we have $p(x_S)$ we moreover have efficient representation of it, using only tables of size r^2 .

In the case of a simple 3-node Markov chain: $G = x_1 - x_2 - x_3$ with $p(x_1, x_2, x_3) \in \mathcal{F}(G, \mathcal{M}^{(f)})$. If we wish to compute any of $p(x_1), p(x_2), p(x_3), p(x_1, x_2), p(x_2, x_3)$ then we may proceed as we did above by sending messages starting at the leaf nodes.

On the other hand, if we wish to compute $p(x_1, x_3)$ then the only choice is to eliminate x_2 in the computation

$$\sum_{x_2} \psi_{1,2}(x_1, x_2) \psi_{2,3}(x_2, x_3) \quad (6.55)$$

which clearly is an $O(r^3)$ computation. We see that the choice of query can have an effect on the computation. In the above, this choice of query has, essentially, been one so that we might as well have been working on a member of the family for the graph where an additional fill-in edge has been added between x_1 and x_3 (i.e., the 3-clique).

In general, when we wish to compute the marginal $p(x_S)$, we need somehow to eliminate $x_{V \setminus S}$, which might introduce edges between nodes in S . Recall that $\sigma = (\sigma_1, \sigma_2, \dots, \sigma_N)$ is an ordering of the nodes. Also, let $\sigma^{-1}(v)$ for $v \in V(G)$ gives number that node v is eliminated by order σ . We have following theorem

Theorem 52. *Let $G = (V, E)$ be an undirected graph with a given elimination ordering σ that maps G to $G' = (V, E')$ where $E' = E \cup F_\sigma$, and where F_σ are the fill-in edges added during elimination with order σ . Then $(v, w) \in E'$ is an edge in G' iff there is a path in G with endpoints v and w , and where any nodes on the path other than v and w are eliminated before v and w in order σ . I.e., if there is a path $(v = v_1, v_2, \dots, v_{k+1} = w)$ in G such that*

$$\sigma^{-1}(v_i) < \min(\sigma^{-1}(v), \sigma^{-1}(w)), \text{ for } 2 \leq i \leq k \quad (6.56)$$

Proof. First part: Induction on $\ell = \min(\sigma^{-1}(v), \sigma^{-1}(w))$ that given any $(v, w) \in G'$ the equation holds. If $\ell = 1$ then $(v, w) \in E$ and $(v, w) \in E'$. Suppose holds for $\ell \leq \ell_0$ and consider $\ell = \ell_0 + 1$. If $(v, w) \in E$ then the equation holds. Otherwise, $(v, w) \in F_\sigma$, and by definition, we have an $x \in V$ with $\sigma^{-1}(x) \leq \min(\sigma^{-1}(v), \sigma^{-1}(w))$ and $x—v, x—w$ in G' . Induction hypothesis implies existence of x, v and x, w paths in G satisfying the equation, combining these chains gives the required v, w path.

Converse: Induction on k , length of path. If $k = 1$ clearly $(v, w) \in E'$. Suppose holds for $k \leq k_0$ and consider $k = k_0 + 1$. From path $(v = v_1, v_2, \dots, v_{k+1} = w)$, choose $x = v_i$ where $\sigma^{-1}(v_i) = \max\{\sigma^{-1}(v_j) | 2 \leq j \leq k\}$. Induction hypothesis implies $v—x$ and $x—w$ in G' . Therefore $v—w$ in G' . \square

Therefore, if there are $v, w \in S$ that are connected by a path strictly within $V \setminus S$, then v, w will be connected once elimination has run. In the worst case, S can become a clique, and computation will be exponential in $|S|$. This will happen when every variable in S is a leaf node in the tree. In the best case, for any $v, w \in S$ there is no path between them outside of S — this is the case in fact where S induces a tree (i.e., S shatters G into multiple disconnected sub-trees, or equivalently $G[S]$ is a tree). The typical case is somewhere in between. Depending on the graph and on the query, we might add additional edges which will necessarily lead to a more expensive computation. From the perspective of most machine learning algorithms that require potentials only along the edges, this does not present any problems. On the other hand, from the perspective of the scientist who might want to compute an arbitrary query, this could quite significantly

6.7 Certain Bayesian network queries can be optimized

Define and then explain what happens with polytrees, and how this is treated in more general graphs (each v-structure becomes a supernode).

The above section, give example of BN where all guys in S are mutually independent, this is optimally bad and would be one reason when moving to an MRF from a BN via moralization would be such that the lost independence statements could cause unboundedly bad things to happen to computation.

Note that for most applications, such as learning, inference, we are interested in S that are subsets of cliques in the original graph G , so this is ordinarily not a problem for machine learning, but it can be a big problem for scientific applications.

Describe what is lost when moving from BNs to MRFs (those two distant nodes might be independent), its consequences for inference, and how that depends on the query.

Can ignoring v-structures have any consequences? Yes, what if S is a set of unconnected parents of a node, we would do some redundant computation, given a BN, we can just remove factors easily since there is already a local normalization in many of the factors, so we know the summations would have to be unity. This can save an exponential amount of computation for certain (but not all) queries.

These types of queries are less widely used — for learning, need clique potential queries. But for other types of queries (i.e., understanding the implied relationship and statistical dependency between arbitrary sets of variables, or computing the mutual information therebetween) we might want arbitrary subsets of variables.

Note, this doesn't happen for the cases above, since all undirected trees could only come from BNs where each node has at most one parent. The issue here comes from BNs with V-structures, where we might lose something if we are marginalizing over everything other than the set of parents of a set of nodes.

Perhaps move this section later, after we've spoken about more general graphs.

6.8 Non-tree graphs

In the previous section, we saw that when G is a tree, it was possible to find an elimination ordering such that no fill-in was required — we could always eliminate next a leaf node of the tree w.r.t. some root node.

When the graph is not a tree, however, then we are no longer guaranteed that there exists an elimination order that produces no fill-in edges. To see this, all we need is a counter example, and the simple 4-cycle suffices as shown in Figure 6.16.

Looking at the four cycle, the elimination of any node means that there is a fill-in edge created that connects the neighbors of the node eliminated. Once that node is done, remaining nodes can be eliminated without any further fill-in. This means that, while a 4-cycle corresponds to a distribution that has clique potentials involving no more than two variables, there is no elimination procedure that is less costly than the $O(n^3)$ computation implied by the resulting clique. While there is no elimination order that does not produce a fill-in for a 4-cycle, we may still wish to compute a given marginal which means that such a process is inherently an $O(n^3)$ computation. Mathematically, this isn't a problem: we are free to eliminate nodes in any order up to the marginal we desire.

What would happen if we try to run the message passing algorithm defined on trees on a non-tree graph? As an example, again consider the 4-cycle. Here, however, we see that no message is ready to be sent from the perspective of MPP since it depends on receiving messages along the other edge. Due to the cycle, there is deadlock. Even if we were to start sending messages (by allowing one message arbitrarily to be sent first to break the deadlock), it is not clear what the resulting messages correspond to and if they would ever converge on something. And if they do converge, what they would converge to. As an example, consider the 5-cycle with the given potential functions , and if we keep sending messages along the cycle, we will oscillate and never converge. Whenever we had a tree, like in the previous section, once all messages have come into a node, messages no longer effect the “state” of the model in the sense that each additional message is the same. Unlike this example, however, performing message passing such as this on graphs with loops is an enormous part of approximate inference in graphical models and in many cases doing just this can be shown not only to converge (in some cases) but in some other cases it can converge to the right answer. We will address message passing on graphs with cycles in later sections. For now, we are still concerned with performing exact (i.e., guaranteed to be mathematically correct) inference on graphs with cycles. And we will do this starting again with the notion of the elimination algorithm.

In general, if we are able to always eliminate a node v where the neighbors $\delta(v)$ induce a complete subgraph of G , then there will be no additional fill-in, and no factors of any p in the corresponding family will be unnecessarily coupled together in performing the summation. This is a desirable property since the factor coupling is what we have seen in the tree case makes the computation more costly than necessary. When $\delta(v)$ does not induce a complete subgraph (i.e., if the neighbors of v are not completely connected to each other), then we will induce a bigger clique than what was in the original graph.

Note that in the 4-cycle case, if we had started with a graph that also had an edge x_2, x_3 and eliminated x_1 , there would be no fill in. Thus, there would be no penalty for eliminating x_1 first (although there

would still be a penalty for eliminating x_2 first). But the important point is that adding the extra edge has not increased the inherent computation of computing just with the pure 4-cycle (without the extra edge). Note that the 4-cycle states that any family member has both $X_1 \perp\!\!\!\perp X_4 | \{X_2, X_3\}$ and $X_2 \perp\!\!\!\perp X_3 | \{X_1, X_4\}$, while the graph with edge X_2, X_3 only requires family members to have the first property. So the extra independence properties of the 4-cycle does not help us in any way computationally.

Consider Figure 6.17-left, where we can see by considering all seven nodes that elimination necessarily produces some fill-in. Clearly, eliminating x_4 first would result in the worst of cases, producing a large 7-node clique, while eliminating any other node would do much better. On the right we see the result of an elimination order that produces a maximum clique of size 4. But like the above, there is no computational reason to have started from the similar family — we might as well have started with the larger family with the extra edges rather than the more restricted family. This means that, in some sense, we are solving the inference problem for a larger family than desired, but it is not a larger family than necessary because no elimination order produces a faster algorithm. In other words, this means that it is necessary to solve the inference problem on a larger family than the original one since there is no elimination order that does not require some fill-in.

Stated in another way, independence properties and factorization granted by the missing edges in a graph are not guaranteed to help computationally, and there are cases where we might as well have started with a larger family. Of course, there are limits to this property. For example, in the 4-cycle above it would have been wasteful to solve inference in a still larger family, i.e., the 4-clique which would cost $O(r^4)$. So clearly, factorization does help sometimes but not always.

We summarize the above in the following lemma.

Lemma 53. *The reconstituted graph after elimination is run corresponds to the family where inference occurs. If elimination produces fill-in edges, inference is solved in a family larger than that specified by the original graph, and we might as well have started with that family to begin with. If an elimination order produces no fill-in, we are solving the inference query optimally.*

Lemma 54. *Let $G = (V, E)$ be a graph, and $G' = (V, E \cup F_\sigma)$ be the reconstituted graph corresponding to elimination order σ . When elimination is run on G' with order σ , no new fill-in edges are added to G' . That is, σ is a fixed-point of G' , or $G'_\sigma = G$.*

Lemma 55. *Let $G = (V, E)$ be a graph, and $G' = (V, E \cup F_\sigma)$ be the reconstituted graph corresponding to elimination order σ . When elimination order σ is run on G' , the set of neighbors v at the time v is eliminated is the same in both the original and in the reconstituted graph. That is $\delta_{G_{i-1}}(\sigma_i) = \delta_{G'_{i-1}}(\sigma_i)$ for all i . Moreover, $\delta_{G'_{i-1}}(\sigma_i)$ is a clique in G'_{i-1} .*

Proof. Any neighbor of v in the reconstituted graph must be either an original-graph edge, or it must be due to a fill-in edge between v and some other node that is not an original graph neighbor. All of the fill-in neighbors must be due to elimination of nodes before v since after v is eliminated no new neighbors can be added to v . The point at which v is eliminated at the original graph and the point at which it v is eliminated in the reconstituted graph, the same previous set of nodes have been eliminated. Therefore, any neighbors of v in the reconstituted graph must have been already added to the original graph when v is eliminated in the original graph. Moreover, since the neighbors are the same, and the first time any neighbors are connected, we see that $\delta_{G'_{i-1}}(\sigma_i)$ must be a clique. \square

Definition 56 (perfect elimination order). *Order σ is called perfect for G if when we eliminate nodes in G according to σ , there are zero fill edges in the resulting reconstituted graph.*

A graph $G = (V, E)$ might or might not have a perfect elimination order. For example, the 4-cycle Figure 6.16 does not have a perfect elimination order, but the graph on the right of Figure 6.16 does have a perfect elimination order. Not all orders are perfect, though — consider starting with x_2 .

Exercise 57. Characterize the class of graphs such that all orders are perfect.

On the other hand, $G' = (V, E \cup F_\sigma)$ always has at least one perfect elimination order, namely σ . We are interested in these graphs since they are the ones that correspond to the family where inference is being performed. This is an important point so we state it again: when we run the elimination algorithm, we couple together variables by adding edges between them if there are any fill-in edges. We know that coupling together variables is an undesirable property since we are expanding the family of distributions the graph represents and also potentially significantly increasing the cost of inference compared to the case of no coupling. If we can find an order that does not require any fill in, we have not coupled variables and this is good. On the other hand, some graphs require unavoidable fill in edges, but once we have performed elimination, the reconstituted graph is one that does have a perfect elimination order. Since we will inevitably need to deal with family of distributions that correspond to graphs that have at least one perfect elimination order, it seems desirable to study such graphs. We might wish to ask questions such as: ok, we accept that there is going to be some fill in, but how can we find the order that results in the smallest fill in. Or perhaps fill-in is not what we wish to minimize, instead we may wish to minimize the inherent computational complexity associated with a given elimination order.

Indeed, given an ordering of the nodes, eliminating the nodes in that order results in complexity on the order of the largest elimination clique encountered. Moreover, the largest clique we encounter when running the elimination procedure, and the set of elimination cliques, have a strong relationship with respectively the overall complexity of running elimination, and the set of maximal clique in the resulting triangulated graph. We therefore characterize these properties next in the following lemmas.

Lemma 58. Given an elimination order, the computational complexity of the elimination process is $O(r^{k+1})$ where k is the largest set of neighbors encountered during elimination. This is the size of the largest clique in the reconstituted graph.

Proof. First, when we eliminate σ_i in G_{i-1} , eliminating variable v when it is in the context of its current neighbors will cost $O(r^\ell)$ where $\ell = |\delta_{G_{i-1}}(v)| + 1$ — thus the overall cost will be $O(r^{k+1})$.

Next, we show that largest clique in the reconstituted graph is equal to the complexity. Consider the reconstituted graph, and assume its largest clique is of size $k+1$. When we re-run elimination on this graph using the same order, there will be no fill in. However, the cost of the elimination step upon reaching the first vertex v of the clique of size $k+1$ will be $O(r^{k+1})$ since k of the variables of the clique will be neighbors of v , but no other nodes will be neighbors since it is a perfect elimination order in the reconstituted graph. This will be the same cost as what was incurred during the initial elimination procedure since v has the same set of neighbors. Therefore, the largest clique in the reconstituted graph is the complexity of doing elimination. \square

For lack of a better term (at the moment), let us call the class of graphs that contain at least one perfect elimination order the *perfect elimination graphs*.

Definition 59 (perfect elimination graph). A graph $G = (V, E)$ is a perfect elimination graph if there exists an ordering σ of the nodes such that eliminating nodes in G based on σ produces no fill-in edges.

In fact, the above lemma shows that any perfect elimination ordering on a perfect elimination graph will have complexity exponential in the size of the largest clique in that graph. Moreover, we can discover the maximum cliques in the resulting perfect elimination graph while we are performing elimination.

Recall that we define a *maxclique* as a maximal clique, and that this is distinct from a *clique* (any fully connected subgraph), and a *maximum clique* (no other clique in the graph has larger size). Also recall that a maximum clique is a maxclique, but not vice versa.

Lemma 60. *When running the elimination algorithm, all maxcliques in the resulting reconstituted graph are encountered as elimination cliques during elimination.*

Proof. Each elimination step produces a clique, but not necessarily a maxclique. But the set of maxcliques in the resulting reconstituted perfect elimination graph is a subset of the set of cliques encountered during elimination. This is because of the neighbor property proven above — if there was a maxclique in the reconstituted graph that was not one of the elimination cliques, that maxclique would be encountered on a run of elimination with the same order on the reconstituted graph, but for the first variable to encounter this maxclique, it would have the same set of neighbors in original graph, contradicting the fact that it was not one of the elimination cliques. \square

With this, the next lemma shows how to acquire the maxcliques of the reconstituted perfect elimination graph while running elimination on the original graph.

Lemma 61. *Given a graph G , an order σ , and a reconstituted graph G' , the elimination algorithm can produce set set of maxcliques in G' .*

Proof. Consider node v 's elimination clique c_v (i.e., v along with its neighbors $\delta(v)$ at the time of elimination of v). Since c_v is complete, either c_v is a maxclique or a subset of some maxclique. c_v can not be a subset of any subsequently encountered maxcliques since all such future maxcliques would not involve v . Therefore c_v must be a maxclique or a subset of some previously encountered maxclique. If c_v is not a subset of some previously encountered maxclique, it must be a maxclique (we add c_v to a list of maxcliques). Since all maxcliques are encountered as elimination cliques, all maxcliques are discovered in this way. \square

Corollary 62. *The first node eliminated in a graph, along with its neighbors, forms a maxclique.*

Note, that in a perfect elimination graph, there can be more than one maxclique containing a given variable v . As an example of this, consider Figure 6.16-right. The clearly has a perfect elimination order, and the maxcliques are $\{x_1, x_2, x_3\}$ and $\{x_2, x_3, x_4\}$, and we see that variables $\{x_2, x_3\}$ live in both maxcliques. There is no bound (other than $O(|V(G)|)$) on the number of maxcliques that a node might be a member of. Consider the star graph, the center node is a member of every maxclique.

As the family of models in which we solve the inference problem grows, the computation can never be reduced. Stated another way, as we add edges to a graphical model, the size of the largest clique can either stay the same or can grow. We recap the questions we want to solve. We start with a graph G corresponding to all $p \in \mathcal{F}((V, E), \mathcal{M}^{(f)})$. We know that to perform elimination, we must have some order σ and some fill in F_σ and work with the family $\mathcal{F}((V, E \cup F_\sigma), \mathcal{M}^{(f)})$ where $\mathcal{F}((V, E), \mathcal{M}^{(f)}) \subseteq \mathcal{F}((V, E \cup F_\sigma), \mathcal{M}^{(f)})$. We know that inference for this more general family can not be computationally easier than the original family. Therefore, this begets the following questions: 1) can we identify the smallest such larger family (best elimination order σ) in which inference is solved? 2) is there some property of this larger family that can be identified, i.e., since the family is always achieved by the result of elimination using some order, is there some property (other than having a perfect elimination order) that characterizes this family and might this help us to find a good elimination order based on just G . We will answer these questions below.

We may *embed* a graph into another graph. That is, any graph $G = (V, E)$ can be embedded into a graph $G' = (V, E')$ if G is a spanning subgraph of G' , meaning that $E \subseteq E'$. Note by embedding one graph into another graph, we only enlarge the family. That is, if G_1 can be embedded into G_2 then $\mathcal{F}(G_1, \mathcal{M}^{(f)}) \subseteq \mathcal{F}(G_2, \mathcal{M}^{(f)})$.

Any graph can be embedded into its reconstituted graph for any elimination order. This means that if σ is an ordering of the nodes and $G_\sigma = (V, E \cup F_\sigma)$ is the reconstituted graph resulting from eliminating nodes in order σ , then G can be embedded into G_σ .

In the elimination process, to decrease the complexity of the process, we care not so much about the number of fill-in edges encountered at each elimination step, but rather the size of the largest clique encountered during elimination (which we know from the above to be the largest clique). Therefore, given a graph G , we want to find the elimination order that results in a reconstituted embedding graph that has the smallest maximum clique size.

Before addressing the above problem of finding the optimal elimination order, it would be useful to more generally characterize graphs that have the property that there exists a fill-in free elimination ordering, since reconstituted elimination graphs are apparently members of this class of graphs. Clearly, not all graphs have a fill-in free order (take the 4-cycle). On the other hand, the 4-cycle with the extra edge (Figure 6.16-right) does have a fill-in free elimination order.

From the above, again, inference cost is always elevated to that of a class of graphs that have a fill-in free elimination order, the perfect elimination graphs, and we want to characterize such graphs.

On the one hand, these models seem attractive since those are the ones for which a perfect elimination order can be found and inference may be performed on the non-elevated graphical model family. For example, the trees we considered in the last section have this property (if we always eliminate leaf nodes, then no fill-in is ever produced). But there are many other non-tree graphs that have this property as well. Consider the examples given in Figure 6.18. In each case, there is a perfect elimination order of the nodes.

So one option would be to always operate on such families of graphs. That is, the graph designer (e.g., a scientist hand-designing a graph to fit a particular application domain, or an algorithm that is searching various structures based on how well they match a given data set) would never consider graphs that do not have a perfect elimination order.

One question is, might this put a restriction on the models in some way? Must we operate in this family rather than the family that we originally desired? Does this impose constraints on the models we are able to work with? And more importantly, does this in some way change the resulting quantities that are computed (e.g., a given probabilistic query)? To answer the last question first, no the resulting quantities are not changed, rather the results are exactly as if we were computing only on the smaller family, but it would use additional computational resources. Recall, computing on a larger family is always mathematically correct with respect to the smaller family. The same is not true the other way around, if we start with distribution $p \in \mathcal{F}(G, \mathcal{M}^{(f)})$ and then project in some way down to some spanning subgraph $G' = (V, E')$ where $E' \subset E$, and solve inference for $\mathcal{F}(G', \mathcal{M}^{(f)})$, then this could yield in arbitrarily inaccurate results (this touches on the issue of structure learning in graphical models, and also on the issue of various forms of approximate inference - in each case we wish to find an E' that is good in some way).

In any event, it seems safe, at least mathematically, to design the model that we wish to use for a given task, and then find a close model within the space of perfect elimination graphs work there.

6.8.1 Triangulated Graphs

Those graphs for which there exist perfect elimination orders are also called the class of *triangulated graphs*. Triangulated graphs are also sometimes referred to either as *chordal*, *rigid-circuit*, *monotone transitive*, or (as we saw above) *perfect elimination* graphs. We will use the terms chordal, triangulated, and perfect elimination interchangeably, once we prove that they are indeed identical. We first need a few definitions.

A *chord* with respect to a cycle in a graph G is an edge that connects two non-adjacent nodes in that cycle. Note a chord, if it exists, is always defined with respect to some cycle — that is, two connected nodes u, v are obviously adjacent in the graph, but might not be adjacent in the cycle if it is the case that the cycle (which comprises a set of nodes) does not list u and v in consecutive order. Therefore, a chord might or might not exist with respect to a given cycle (the adjacent nodes of the cycle don't count as a chord). The 4-cycle in Figure 6.16 does not have a chord. Figure 6.19 shows an example of a cycle in a graph and one

of its chords, as well as a graph that has chordless cycles.

Definition 63 (Triangulated graph). *A graph G is triangulated (equivalently chordal) if all cycles have a chord.*

This means that any cycles of length > 3 must have a chord. Cycles of length 3 have no non-adjacent vertices, so they cannot have a chord. Therefore, in a triangulated, any cycle large enough so that it could possibly have a chord does have a chord. We immediately can identify some classes of triangulated graphs:

1. a clique is a triangulated graph (all cycles have chord).
2. a tree is a triangulated graph, since there are no cycles that could disobey the chordal requirement.
3. a chain is a triangulated graph, since it is a tree.
4. a set of disconnected vertices is triangulated (since there are no cycles).

Lemma 64 (Hereditary property of triangulated graphs). *Any node-induced sub-graph of a triangulated graph is a triangulated graph.*

Proof. If a graph has no chordless cycles, then it has no chordless cycles involving any node v , and removing v only removes cycles involving v and so does not create any new chordless cycles. \square

Our goal is to find a perfect elimination order, or a graph for which one exists. In order for there not to be any fill-in for an elimination step, it means that at each point in elimination when we eliminate a node, the node's neighbors must be fully connected. Any node such that its neighbors are connected is called a *simplicial* node.

Definition 65 (Simplicial). *Let $\delta(v) = \{u : (u, v) \in E(G)\}$ be the set of node neighbors of v in $G = (V, E)$. Then we say that v is simplicial if the vertex induced subgraph $G[\delta(v)]$ is a complete graph.*

Note that once we have a graph that we know to be a perfect elimination graph, it is easy to find the perfect elimination order. We just choose an order that, at each step, does not produce any fill in. That is, we choose any simplicial node according to the current graph and eliminate it.

Theorem 66. *Given graph G , elimination order σ , and perfect elimination graph $G' = G_\sigma$ obtained by elimination on G . We may reconstruct a perfect elimination order (w.r.t. G_σ) from G_σ by repeatedly choosing any simplicial node and eliminating it. Call this new order σ' . Now σ' might not be the same order as σ , but both are perfect elimination orders for G' .*

Proof. If there is more than one possible order, we must reach a point at which there are two possible simplicial nodes $u, v \in G'$. Eliminating u does not render v non-simplicial since no edges are added and thus v has if anything only a reduced set of neighbors. Each time we eliminate a simplicial node, any other node that was simplicial in the original elimination order stays simplicial when it comes time to eliminate it. Repeat this argument for different u , until we eliminate v at which time it is still simplicial. \square

It turns out that triangulated graphs with at least 2 nodes always have at least two simplicial nodes. And since any vertex-induced subgraph of a triangulated graph is still triangulated, once we eliminate one of the simplicial graphs what remains will also be a triangulated graph with (if it has more than 2 nodes) two simplicial nodes. Therefore, we can produce a sequence of elimination steps without needing any fill in.

First, in order to easily show that triangulated graphs have two simplicial nodes, we first use an auxiliary lemma regarding triangulated graphs — namely that all minimal separators in a triangulated graph are

complete. This is an important theorem and will be used later in this chapter. First a few definitions and then the theorem.

Given $a, b \in V$, $a \neq b$, a set $S \subseteq V$ is an (a, b) -**separator** in G , if all paths from a to b must intersect some node in S . A **minimal** (a, b) -**separator** S is an (a, b) -separator such that any strict subset $S' \subset S$ is no longer a separator. A set S is a **separator** in $G = (V, E)$ if there exists $a, b \in V$ such that S is an (a, b) -separator. And lastly, a set S is a **minimal separator** if there exists an $a, b \in V$ such that S is a minimal (a, b) -separator.

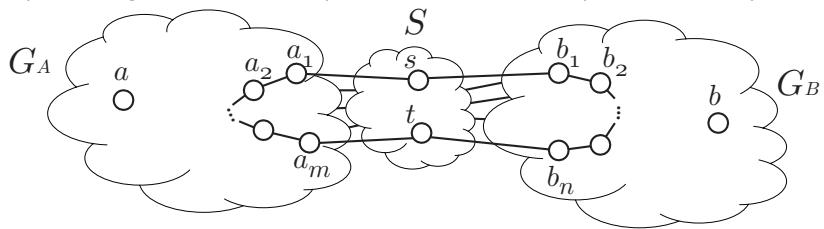
For example, consider Figure 6.20. Top row left, we see that both $\{x_3, x_4\}$ and $\{x_2, x_3, x_4\}$ is a (x_1, x_5) -separator, but only $\{x_3, x_4\}$ is a minimal (x_1, x_5) -separator since there is no path from x_1 to x_5 that can avoid traveling through either x_3 and x_4 . Top row middle, $\{x_3, x_4\}$ is no longer a separator at all due to the additional edge allowing the path from x_1 to x_5 via x_2 . The $\{x_2, x_3, x_4\}$ is now a minimal (x_1, x_5) -separator. Top row right: $\{x_2, x_4, x_6\}$ is a minimal (x_1, x_3) -separator, and $\{x_4, x_6\}$ is a minimal (x_5, x_7) -separator, so one minimal separator can include another. Bottom row: left - C, K is a minimal (B, E) -separator; middle - D, F, K is a non-minimal (B, E) -separator; right A, F, K is a minimal (B, E) -separator.

Lemma 67. Let S be a minimal (a, b) -separator in $G = (V, E)$ and let G_A, G_B be the connected components of G once S is removed (i.e., $(G_A, G_B) = G[V \setminus S]$) where $a \in V(G_A)$ and $b \in V(G_B)$. Then each $s \in S$ is adjacent both to some node in G_A and some node in G_B .

Proof. Suppose the contrary, that there exists a $v \in G_A$ not adjacent to some $s \in S$. In such case, $S \setminus \{s\}$ is still a separator contradicting the minimality of S . \square

Lemma 68. A graph $G = (V, E)$ is triangulated iff all minimal separators are complete.

Proof. First, suppose all minimal separators in $G = (V, E)$ are complete. Consider an arbitrary cycle $u, v, w, x_1, x_2, \dots, x_k, u$ of length 4 or greater starting and ending at node u , where $k \geq 1$. Then the pair v, x_i for some $i \in \{1, \dots, k\}$ must be part of a minimal (u, w) -separator, which is complete, so v and x_i are connected thereby creating a chord in the cycle. Thus, since the cycle is arbitrary, all cycles are chorded.



Next, suppose $G = (V, E)$ is triangulated, and let S be a minimal (a, b) -separator in G , and let G_A and G_B be the connected components of $G[V \setminus S]$ containing respectively a and b . Each $s \in S$ is connected to some $u \in V(G_A)$ and $v \in V(G_B)$. Therefore, since the components are connected, for each $s, t \in S$, there is a shortest path $s, a_1, a_2, \dots, a_m, t$ with $a_i \in V(G_A)$ for $i \in \{1, \dots, m\}$, and a shortest path $t, b_1, b_2, \dots, b_n, s$ with $b_j \in V(G_B)$ for $j \in \{1, \dots, n\}$ (see above figure). Only successive a_i 's in the path, and also s, a_1 and a_m, t , are adjacent as otherwise the path could be made shorter. The corresponding property is also true for the b_i 's. Also, no a_i is adjacent to any b_j since S is a separator. A cycle is formed by $s, a_1, a_2, \dots, a_m, t, b_1, b_2, \dots, b_n, s$ which must have a chord, and the only candidate chord left is s, t . Since s, t are chosen arbitrarily from S , all pairs of nodes in the minimal separator are connected, and it is thus complete. \square

Lemma 69. A triangulated graph on $n \geq 2$ nodes is either a clique, or there are two non-adjacent nodes that are simplicial.

Proof. First, any clique is triangulated since all cycles with non-adjacent nodes have a chord, so assume the graph is not a clique.

We prove this inductively. Any graph with $1 < n \leq 3$ is triangulated and has two simplicial nodes, and any chorded 4-cycle also has two simplicial nodes.

We next assume true for $n - 1$ nodes, and show for G with n nodes. Let a and b be two non-adjacent vertices and let S be a minimal (a, b) -separator which must be complete. Let G_A and G_B be the connected components of $G[V \setminus S]$ containing respectively a and b . Let $A = V(G_A)$ and $B = V(G_B)$. By induction, $G[A \cup S]$ and $G[B \cup S]$ are either cliques, or contain two non-adjacent simplicial vertices. In the first case, all nodes are simplicial, and in the second case both simplicial non-adjacent vertices cannot be in S since S is complete. In all cases, we may choose two non-adjacent simplicial vertices, one each in A and B , and these vertices are adjacent to no nodes other than $A \cup S$ and $B \cup S$ respectively. Therefore, these nodes remain simplicial and non-adjacent in G .

□

Does a graph being triangulated mean that all nodes are simplicial? No, consider the chorded 4-cycle (Figure 6.16-right). It is triangulated (and has two simplicial nodes) but the two other nodes are not simplicial.

In any event, the above theorem is important since it means that a triangulated graph always has a node that can be eliminated without adding any fill-in.

Corollary 70. *Eliminating any simplicial node in a triangulated graph yields a sub-graph that is still triangulated.*

Proof. Eliminating a simplicial node v is identical to the induced graph $G[V \setminus \{v\}]$, so the result follows from Lemma 64. □

Corollary 71. *For any triangulated graph, there exists an elimination order that does not produce any fill in.*

Once we eliminate a simplicial node, thereby creating no fill-in, if there are any nodes remaining, there will always be a simplicial node remaining, and this process can repeat until all nodes are eliminated.

In fact, the relationship is actually stronger than this. In fact, we have:

Lemma 72. *If G is a graph and there exists a perfect elimination order, then G is triangulated.*

Proof. By induction. It is obviously true for 1 and 2 nodes. Assume true for n nodes, and we are given an $n+1$ node graph. Since there exists an elimination order without fill-in, there exists a simplicial node, where chorded cycles can not exist through that node since all of its neighbors are connected. Once we eliminate that node, no fill-in is created, and induction step applies. □

We summarize the bijection as follows:

Theorem 73. *A graph G is triangulated iff there exists a perfect elimination order over the nodes in G .*

This is the reason why triangulated graphs are also called perfect elimination graphs.

Note that in any graph, once we have found an elimination order, then like in the tree-case above, if we run the same elimination order on the reconstituted graph, no fill-in will be produced.

Corollary 74. *Take any graph G and an elimination order σ , then the reconstituted graph $G' = (V, E \cup F_\sigma)$ is triangulated.*

This means, therefore, that it is always possible to generate any reconstituted elimination graph using the elimination order in reverse and constructing the graph node-by-node. We do this by starting with σ_N . We then add σ_i , for $i = N-1 \dots 1$ and connect σ_i to the complete set of nodes in $\{\sigma_N, \sigma_{N-1}, \dots, \sigma_{i+1}\}$ that were the neighbors of σ_i at the point at which σ_i was eliminated. In fact, we can generate any triangulated graph this way by first finding a perfect elimination order, and then applying the order in reverse. We simply recreate nodes one-by-one in each case giving that node the neighbors it had during the original forward-direction perfect elimination ordering, as shown in the following algorithm:

Algorithm 10: Regenerate triangulated graph.

Input: A triangulated graph $G = (V, E)$ and a perfect elimination order σ

Result: A new graph G' identical to G .

Corollary 75. All triangulated graphs can be generated by choosing a next node and connecting that node to some set of previously chosen nodes that are fully connected.

Of course, a tree can be generated this way, we take next node, and connected it to some one single previous node in the tree. We will always have the tree property (unique path between any two nodes) since a cycle can never be produced. The above theorem states that any triangulated graph can be obtained in a generative fashion using an elimination order in reverse.

It is elucidating to point out that while a reverse elimination order can generate any triangulated graph, this does not mean that all triangulations of a graph can be obtained by an elimination order on that graph. That is, let $G = (V, E)$ be any graph, and let $F \subset V \times V$ be a set of pairs of nodes so that $G' = (V, E \cup F)$ is triangulated. If F is the resulting fill-in due to an elimination order on G , then the triangulated graph G' can of course be obtained via an elimination order. On the other hand, some F 's cannot be obtained with any elimination order. As an example, consider a 4-cycle, and consider the triangulation of the 4-cycle corresponding to 4-clique. No elimination on the 4-cycle can yield the 4-clique since once any node is eliminated in the 4-cycle, no edge can be added between that node and the diagonally opposite node in the 4-cycle. Note, however, that the 4-clique is perhaps a special type of triangulated graph, in that there are edges that can be removed without destroying the chordality of the graph. In fact, we have the following theorem.

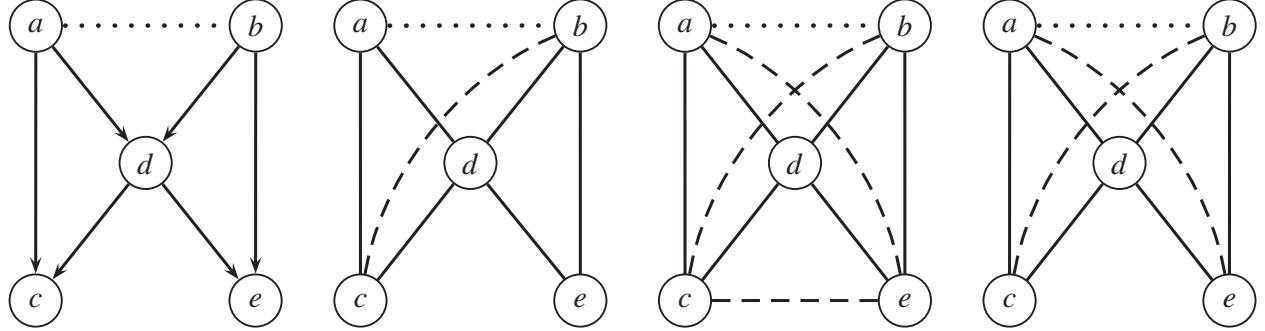
Theorem 76. Let $G = (V, E)$ be a graph and let $G' = (V, E \cup F)$ be a triangulation of G with F the required edge fill-in. If the triangulated graph is minimal in the sense that for any $F' \subset F$, the graph $G'' = (V, E \cup F')$ is no longer triangulated, then F can be obtained by the result of an elimination order. That is, the elimination algorithm and the various variable orderings may produce all minimal triangulations of a graph G .

Minimal triangulations are of interest since to find a triangulation with minimal maximum clique size, it is sufficient to consider a minimal triangulation (any non-minimal triangulation can only possibly have larger cliques). Moreover, if we consider state-space (i.e., the complexity of a clique when the variables all might not have the same domain sizes), then again a minimal triangulation is sufficient.

6.8.2 Nonminimal triangulations for big cliques in sparse models

When the distribution is not strictly positive, however, then sometimes it is desirable to have larger cliques (to take advantage of sparsity in the model). Therefore, in such cases non-minimal triangulations can be

beneficial. For example, consider a distribution $p(a, b, c, d)$ and let d be a deterministic function of a and b so that the distribution is not everywhere positive. Variables a, b, c variables have a size- r domain and d has a size- $r^2 - 1$ domain. Consider the following figure:



On the left, we see that the moralized graph is already triangulated, as there is a perfect elimination order of (c, a, b, d) . This will result in one maxclique with state space $O(r^2)$, and two cliques with $O(r^4)$. So the total state space cost is $O(r^4)$. On the left-middle, the elimination order is (a, c, b, d) , and the cost is still $O(r^4)$. On the middle-right, we start by eliminating node d leading to a cost that is still $O(r^4)$. Finally, on the right, we see a triangulation unobtainable using any elimination order and which has a cost of $O(r^3)$, which is unboundedly better than $O(r^4)$ for large r . Note that we will discuss aspects of inference that might deal with these issues, and there can be other graph transformation procedures that can also handle these sorts of situations at the loss of some semantic interpretability of the graph itself. Nonetheless, we see that non-minimal triangulations can lead to imperfect state space when the distributions non-positive. Further details on such non-minimal triangulations are given in [14].

For the rest of this discussion, however, we will assume positive distributions and thus minimal triangulations.

Exercise 77. Prove that minimal triangulations are state-space optimal.

6.8.3 Higher order trees and graph triangulation

Lets now summarize where we have been. We wish to run the elimination algorithm on a graph. We know that doing so produces a triangulated graph and we know that the elimination complexity is its largest clique. The cliques in the triangulated graph resulting from elimination are exactly the cliques encountered during the elimination process, and the complexity of elimination corresponds to the cliques in the resulting triangulated graphs. Therefore, we want to find the covering minimally triangulated graph that has the smallest largest maxclique size (i.e., the graph whose largest maxclique is smallest) — equivalently we want to find the optimal elimination order. Is this an easy or hard problem? To answer this question we define a generalization on trees.

The generative procedure for trees is mentioned in Theorem 41. This process can be modified to form generalizations of a tree as defined as follows:

Definition 78 (k -tree). A complete graph with $k + 1$ nodes is a k -tree. To construct a k tree with $n + 1$ nodes starting from a k -tree with n nodes, choose some size k complete sub-graph of the n -node k -tree and connect the $n + 1$ 'st node to all nodes in the k -node complete sub-graph.

This means that any complete graph of size n is an $(n - 1)$ -tree. Also, a standard tree is just a 1-tree. All of the trees mentioned above can be generated in this way — we may find a perfect elimination order, and then consider the order in reverse where we connect a node to a set of 1 (one) previously selected nodes.

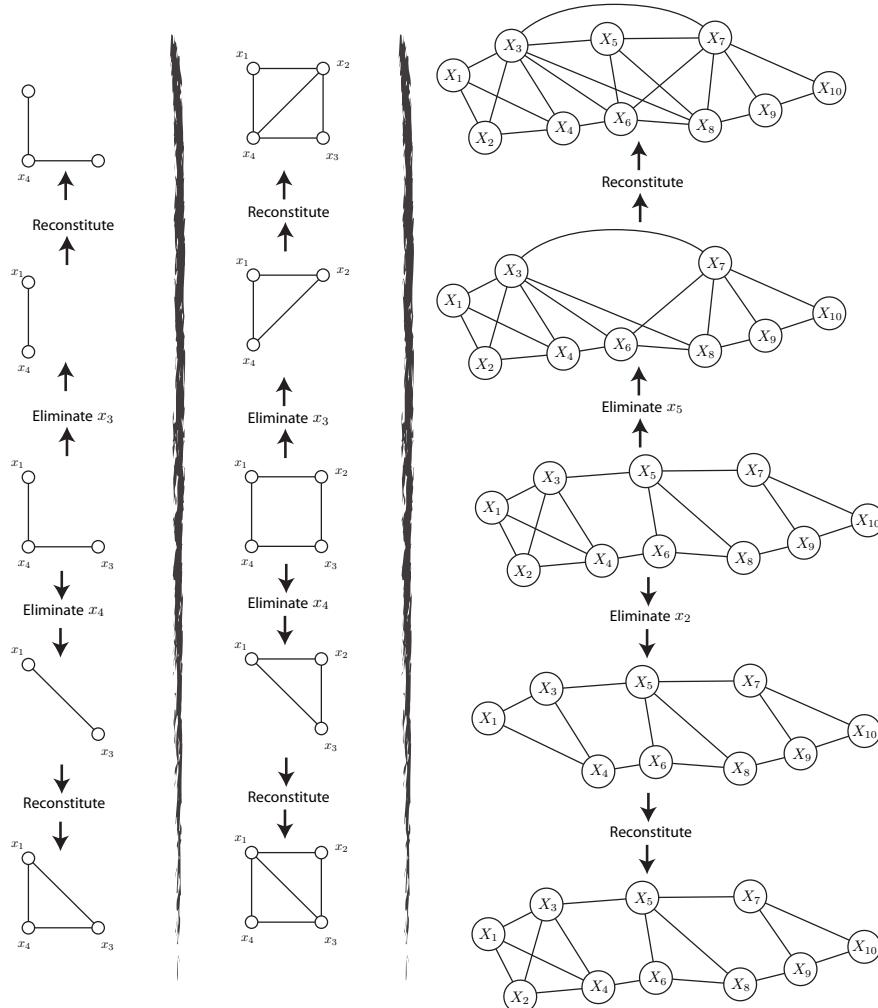


Figure 6.4: Examples of the process of graph elimination on various graphs and the corresponding reconstituted graph above or below it. On the left, we start with a tree. If we eliminate x_4 we produce a clique over x_1, x_3, x_2 while if we eliminate x_3 there are no fill-in edges. In the middle, we start with a 4-cycle. In this case, any variable that we eliminate will induce a fill-in edge. On the right, we start with a more complicated graph. Eliminating variable x_2 leads to no fill-in edges since x_2 's neighbors are already fully connected to each other, while eliminating x_5 leads to an elimination clique of size 5 consisting of the set of variables $\{x_3, x_5, x_6, x_7, x_8\}$.

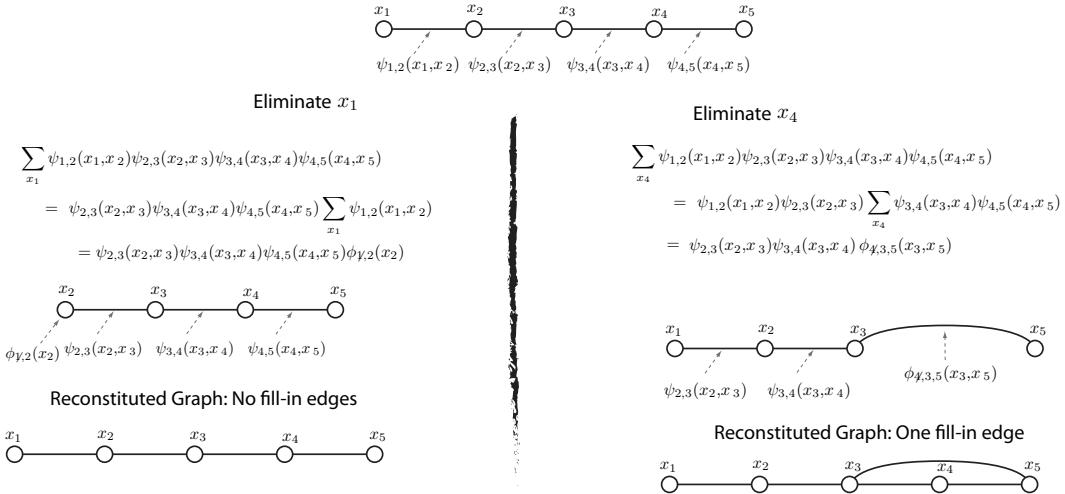


Figure 6.5: Examples of the process of graph elimination on a chain, and equations corresponding to the graphical operation. On the left side, we eliminate x_1 first and this leads to no additional fill-in edges in the reconstituted graph. On the right side, however, we eliminate variable x_4 which induces a fill-in edge between nodes x_3 and x_5 . In the reconstituted graph, the clique size is now 3 which means that the computation resulting from first eliminating x_4 is at least $O(r^3)$.

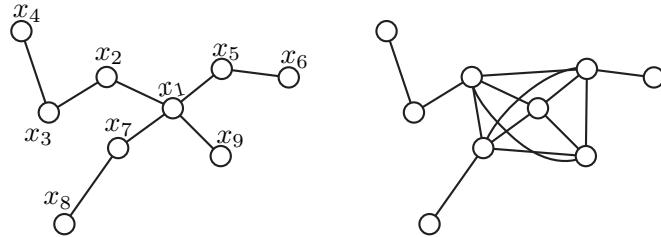


Figure 6.6: Left: Example of a tree graph. If we eliminate starting with node x_1 then that will couple together the factors to for all intents and purposes create a factor that involves all the variables in the clique shown at the right (which has a clique of size 5). It is much better to eliminate the nodes of the tree starting at the leaves which is guaranteed not to add any fill-in edges.

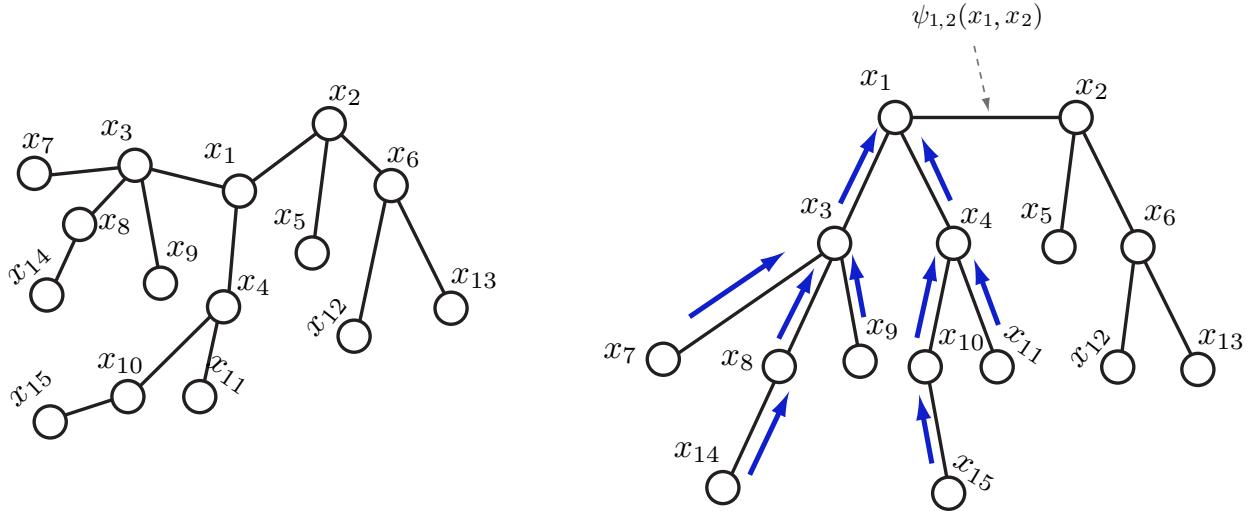


Figure 6.7: A tree (on the left) that is redisplayed on the right so that the “root” is at the edge $(i, j) = (1, 2)$. The edge is marked by the factor at that edge. The blue arrows define a message order so that only leaf nodes are eliminated.

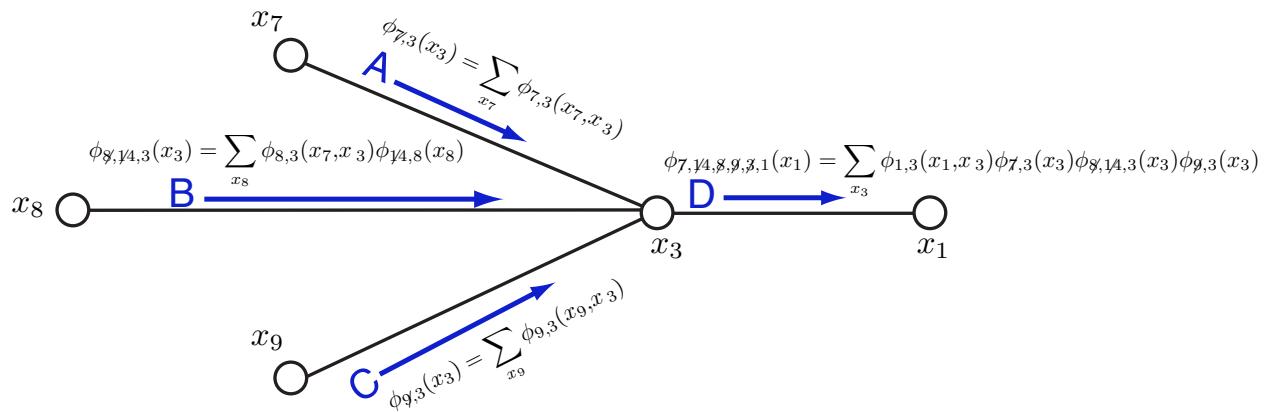


Figure 6.8: Expanded graph showing the incoming messages into node x_3 from nodes x_7 , x_8 , and x_9 and then x_3 's message sent out to its destination x_1 . Note that the four blue “message” arrows (A, B, C, and D) in the graph correspond respectively (and precisely) to Equations (6.39b), (6.39c), (6.39d), and (6.39e). The message passing protocol simply states, for example, that we should not execute the computation of Equation (6.39e) (message D) until we have finished with the computation of Equations (6.39b), (6.39c), and (6.39d) (messages A, B, and C). Compare this figure with Figure 6.13.

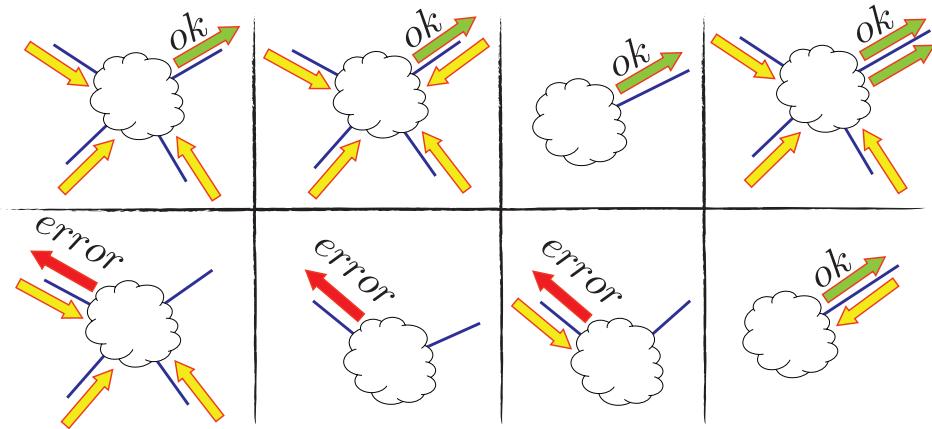


Figure 6.9: Examples of cases where messages are allowed according to MPP (green outgoing arrows) vs. cases where messages are disallowed by MPP (red outgoing arrows).

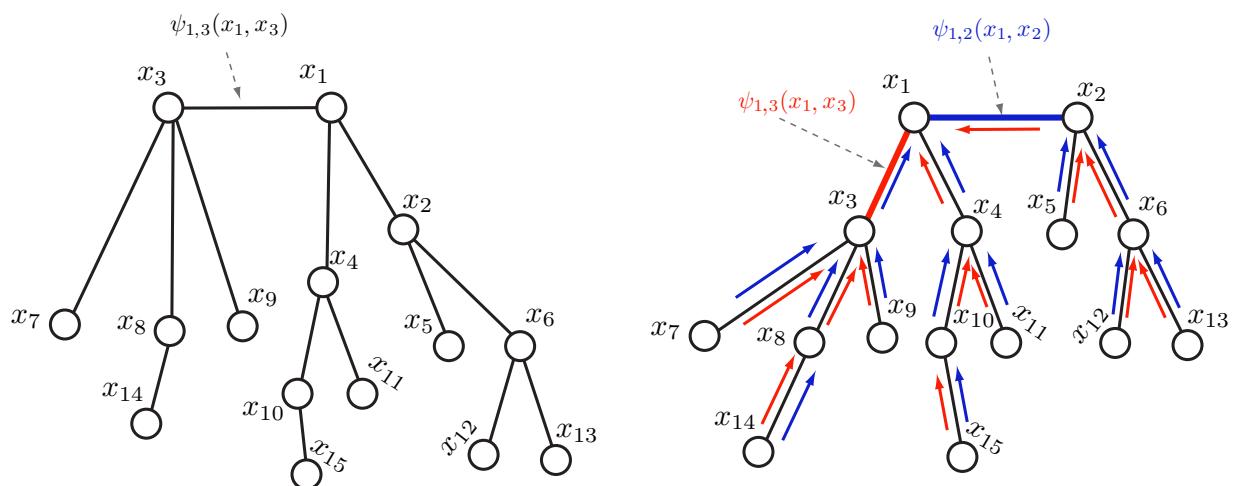


Figure 6.10: Left: The tree rooted at edge $(3,1)$. Right: the messages that are needed to compute the marginal at the edge $(1,2)$ are shown as blue arrows, and the messages that are needed to compute the marginal $(3,1)$ are shown in red. Most of the messages are the same (indicated by an edge having both a blue and a red arrow) implying that we should not send them more than once.

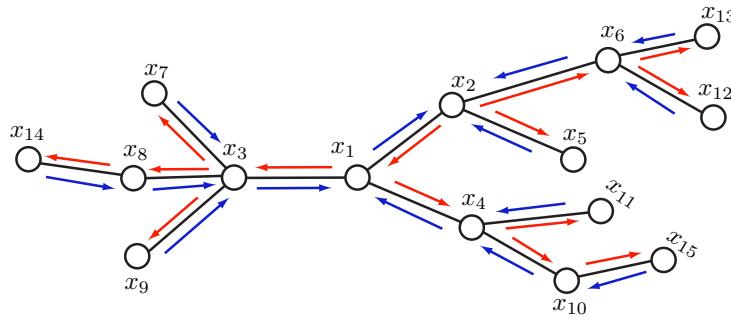


Figure 6.11: The same tree in recent figures drawn without an obvious root, and marked with all messages going from each node to its neighbors. One root node, however, is designated x_2 and the blue messages are sent from the leaves to x_2 and the red messages are sent from x_2 back down to its descendants.

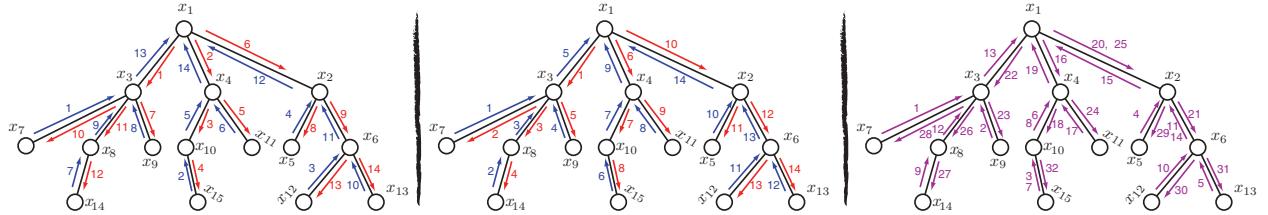


Figure 6.12: Examples of the messages as ordered by MPP. Left and middle: the blue arrows indicate messages towards a arbitrary root node (x_1 in this case), and the red arrow indicate messages away from the root node. The numbers next to each arrow indicate the order of the message. Right: another set of messages obeying MPP, but in this case some of the messages have been sent redundantly (i.e., messages 3 and 7 are the same; so are 11 and 14, 6 and 8, and 20 and 25). All figures show possible message orderings that abide by the message passing protocol, but only the middle could have come from the collect/distribute evidence algorithms with root x_1 .

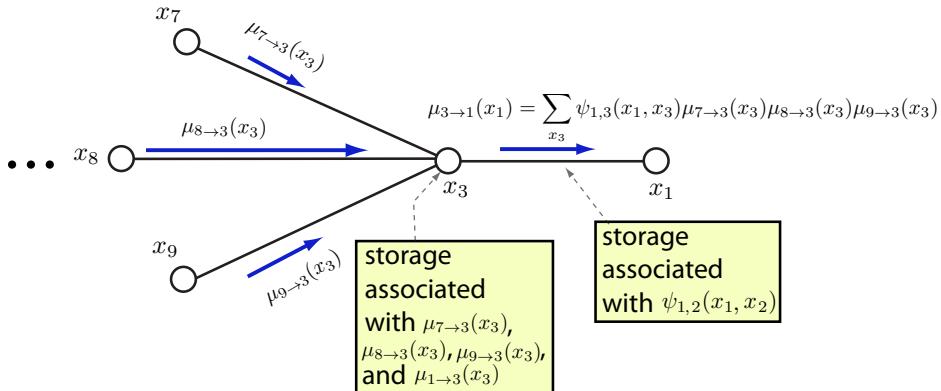


Figure 6.13: The standard message passing algorithm in trees suggests that for each edge (i, j) there is storage associated both with the edge itself, as in $\psi_{i,j}(x_i, x_j)$ but also with all of the incoming messages, $\mu_{k \rightarrow i}(x_i)$ for all $k \in \delta(i) \setminus \{j\}$. This figure shows this for $(i, j) = (3, 1)$ and for $\delta(i) = \{7, 8, 9, 1\}$, and so we need to store three 1-dimensional tables at x_3 for the incoming messages that need to be sent to x_1 , and for the one that ultimately comes back from x_1 . To see how generic messages correspond to the elimination algorithm, compare this figure with Figure 6.8.

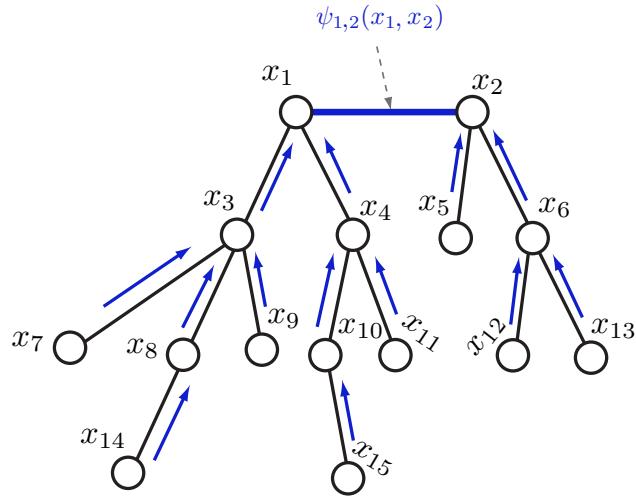


Figure 6.14: A tree rooted at edge $(1, 2)$ and all messages required for computing the marginal $p(x_1, x_2)$.

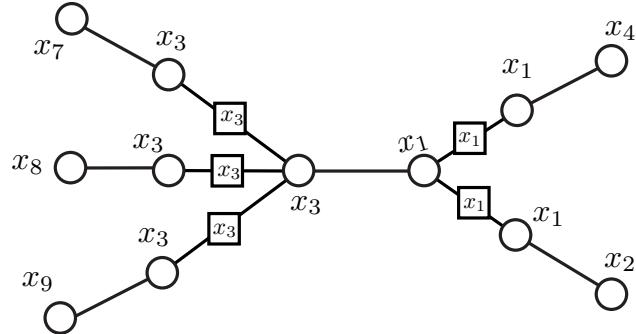


Figure 6.15: Portion of the tree where every pair of edges that shares a common node has an extra node potential (shown as a square node) corresponding to that common node. The common node separates the tree into two separate sub-trees. For example, edge $(7, 3)$ and $(3, 1)$ share the common node 3 and so there is a distinct square x_3 node corresponding to the edge pair $((7, 3), (3, 1))$ and separator potential function $\phi_{7,3,1}(x_3)$.

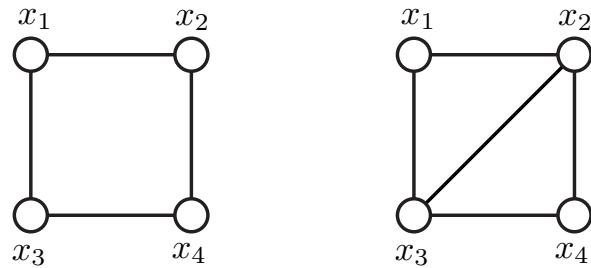


Figure 6.16: The four-cycle (left) shows that there is no node that can be eliminated without producing some fill-in edge. As soon as we add an edge to the four cycle (right) then there is an elimination order w.r.t. this new graph that does not produce fill-in.

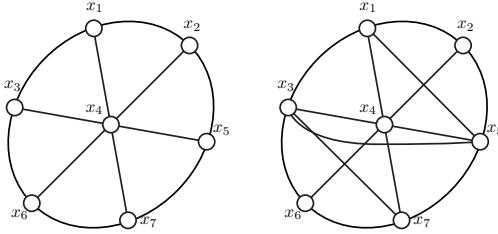


Figure 6.17: This seven node graph also has the property that no node can be eliminated without causing some fill-in to occur.

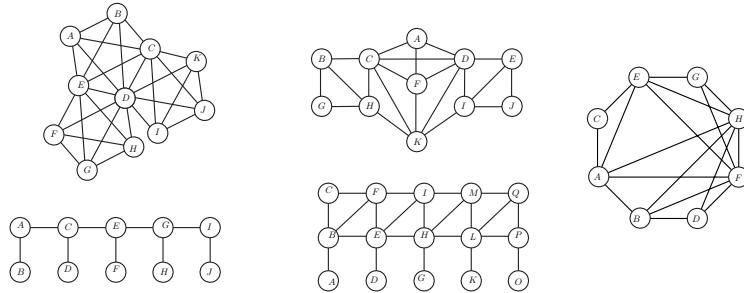


Figure 6.18: Graphs for which there exists a perfect elimination order. These are also examples of chordal/triangulated graphs

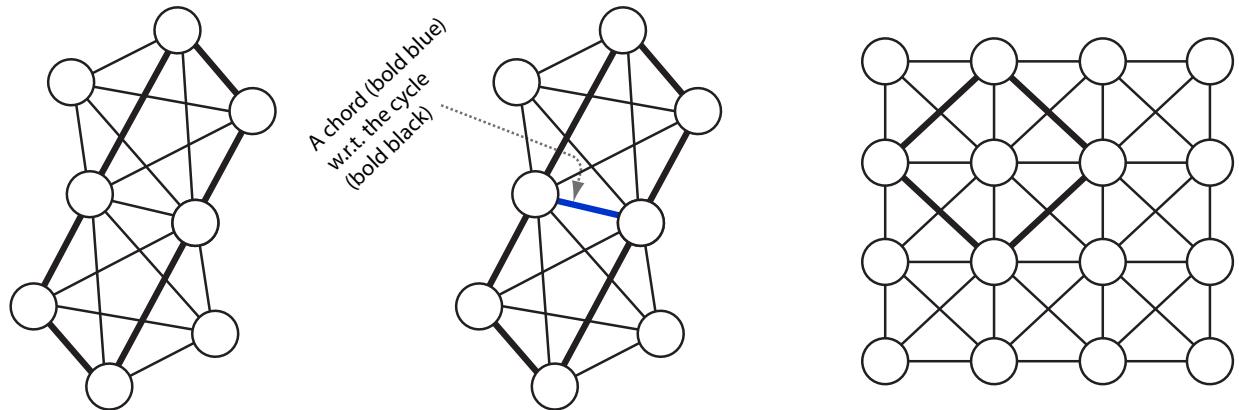


Figure 6.19: Left: A graph with a cycle (in bold), the same graph with one of the cycle's chords (middle, with chord in bold blue). Right: a graph that has many chordless cycles, one of which is shown in bold.

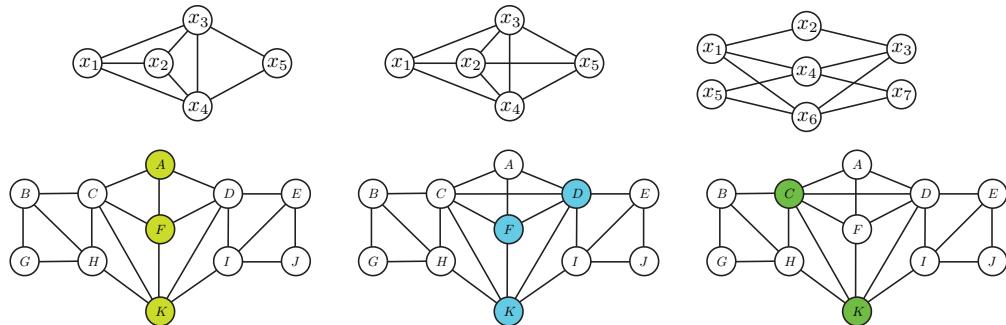


Figure 6.20: Example graphs and separators.

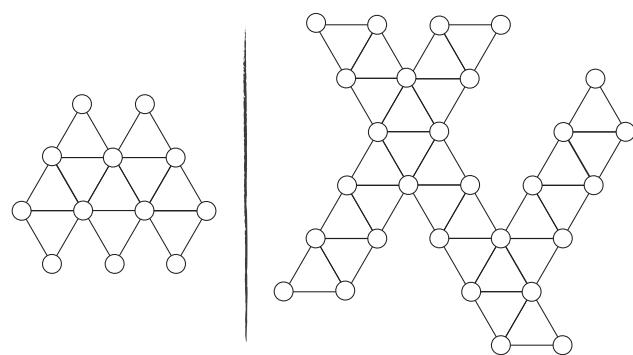


Figure 6.21: Two examples of 2-trees.

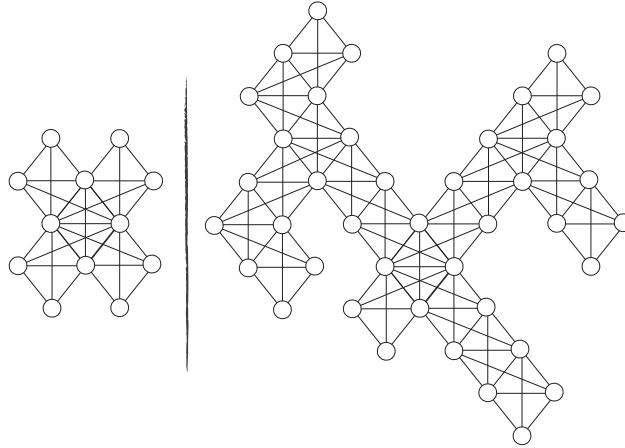


Figure 6.22: Two examples of 3-trees.

A 2-tree is generated by starting with 2 adjacent nodes, and then adding a new node by connecting it to two existing connecting nodes. Figure 6.21 shows a few examples of 2-trees. A 3-tree is similar, connect any new node to a 3-clique of existing nodes. Figure 6.22 shows examples of 3-trees. In these examples, it can be seen that there is a perfect elimination order since it is always easy to find a sequence of simplicial nodes to eliminate (consider the order in reverse of the generative process). Therefore, all k -trees are triangulated.

Moreover, the graphs in the figures are drawn to show how “tree-like” they are, and they might even be seen as being “fat” trees.

Note that in a tree (a 1-tree), every minimal separator is of size 1. This is because all non-leaf nodes are minimal separators, and any set of two or more nodes can be shrunk to any of the non-leaf nodes it contains while retaining the separator property.

Generalizing, in a k -tree, every minimal separator is of size k , and since it is triangulated, is a k -clique.

Exercise 79. Show that in a k -tree, every minimal separator is of size k .

Corollary 80. In a k -tree, the largest clique has size $k + 1$.

Proof. Consider the k -tree generative process. Each newly added node, along with its neighbors, produces a clique of size $k + 1$. Inductively, each additional node never makes adjacent more than a set of nodes of size $k + 1$. \square

Since in a k -tree, the largest clique has size $k+1$, when performing elimination algorithm, the complexity of inference will be $O(r^{k+1})$.

In fact, we have an even stronger theorem regarding k -trees and minimal separators.

Lemma 81. A graph $G = (V, E)$ is a k -tree iff

- G is connected
- G 's maximum clique is of size $k + 1$
- Every minimal separator of G is a k -clique.

Proof. Add proof here. \square

If we start with a k -tree and remove edges, we obtain what is called a *partial k -tree*, which is a spanning subgraph of a k -tree. Any partial k -tree may be embedded into a k tree, so inference in a partial k -tree is

also at most $O(r^{k+1})$. A partial k -tree, however, might be embeddable into a ℓ -tree for $\ell < k$ (more on this below).

A k -tree is necessarily triangulated (due to the generative process above which gives a reverse perfect elimination order for the resulting k -tree), but not all triangulated graphs are necessarily k trees (See for example Figure 6.18).

Any triangulated graph, however, is a spanning subgraph of some k -tree for a large enough k . In fact, since any graph, after running elimination and reconstituting, is triangulated, any graph can be embedded into a k -tree of suitably large k . We can see this since, trivially, an n -clique which is a $(k - 1)$ -tree embeds any subgraph.

This is clearly not a good upper bound, however, into which to place a given triangulated graph. A better upper bound keeps track of the largest clique we encounter during elimination. As we generate the triangulated graph by eliminating, we keep track of maximum clique size encountered, and that becomes the $k + 1$ of the k -tree into which G can be embedded into. The reason this is so is described by the following lemma:

Lemma 82. *If G is a triangulated graph with at least $k + 1$ vertices and has a maximum clique of size at most $k + 1$, then G can be embedded into a k -tree.*

Proof. Let $\sigma = (\sigma_1, \dots, \sigma_n)$ be a perfect elimination node order for G . We can make an embedding of G into a k -tree by adding edges to G in such a way that the same ordering is still perfect in the k -tree. We form the embedding inductively as follows.

For the base case, any set of $k + 1$ vertices can be embedded into a k tree by making those vertices a clique. In an elimination order, we can do this by adding edges to the last $k + 1$ vertices to be eliminated, that is we add edges such that the subgraph induced by $\{\sigma_{n-k}, \sigma_{n-k+1}, \dots, \sigma_n\}$ is a $k + 1$ -clique.

For the induction step, assume the subgraph with vertices $\{\sigma_{i+1}, \dots, \sigma_n\}$ has been embedded into a k -tree T_{i+1} . Since the maximum clique size of G is $k + 1$, in G vertex σ_i is adjacent to a clique c with no more than k vertices in $\{\sigma_{i+1}, \dots, \sigma_n\}$. In the k -tree T_{i+1} , c is contained in a k -clique c' . When we make σ_i adjacent to all of the vertices of c' , we obtain a k -tree T_i since σ_i is still simplicial in T_i (all its neighbors are complete). Repeating this down to σ_1 , we obtain a supergraph of G that is still triangulated (since the order is still perfect). \square

Given a triangulated graph obtained by running the elimination algorithm, we know the size of the largest clique $k + 1$ in the graph, and the above states that the resulting graph can be embedded into a k -tree. In a k tree, all cliques are of size $k + 1$, but in an elimination graph, some cliques might not be that large. This means that in practice, some of the elimination steps will not cost as much as $O(r^{k+1})$, but from the perspective of understanding the complexity, there is no penalty to assuming that all cliques have the same $k + 1$ size.

Exercise 83. *Given a triangulated graph created via elimination, where the largest clique is of size $k + 1$, describe an algorithm to construct a k -tree such that the graph can be embedded into it.*

Any partial k -tree is embeddable into a k -tree.

Of course, using the maximum clique encountered during the elimination order to decide the k into which the resulting triangulated graph can be embedded into is better than embedding into an n -clique, but is this the best k ? We want much more than to just embed a graph into a k -tree for the k we found during one particular elimination order. We want to find the elimination order (out of all $n!$ such orders) that results in the smallest maximum clique. This is important since the resulting computation is exponential in the largest clique size, so even reducing the clique size by one can yield enormous benefits.

What we want, however, is in some sense to find a form of minimal “chordal cover” of the graph. Given a graph $G = (V, E)$, let $\mathcal{F} = \{F_1, F_2, \dots\}$ be the set of fill-in edges such that $G_i = (V, E \cup F_i)$ is chordal.

We want to find the F_i that makes G chordal but that minimizes the largest clique in G_i . Clearly, the size of the largest clique is a lower bound on $k + 1$ in any k -tree embedding. Moreover, if we have a k -tree embedding, this means that the size of the largest clique in the original graph can be no more than $k + 1$. In other words, to find an elimination order that yields the smallest size maximum clique in the resulting triangulated graph, we can find the smallest k such that G can be embedded into a k tree — we wish to find the best “chordal cover” for a graph G .

Once done, we have a chordal graph and can run elimination on simplicial nodes, thereby achieving the best possible overall complexity.

Unfortunately, finding the minimal embedding is an NP-hard optimization problem.

Theorem 84. *For an arbitrary graph $G = (V, E)$, finding the smallest k such that G can be embedded into a k -tree is an NP-complete optimization problem (i.e., the decision version of the problem, asking if G can be embedded into a k -tree of size k , is NP-complete).*

Proof. Reduction is from Minimum Cut Linear Arrangement. I.e., let σ be an ordering of the nodes V , then the *linear cut value* of G w.r.t. order σ is defined as:

$$c_\ell(G) = \max_{1 \leq i \leq |V|} |\{(u, v) \in E(G) : \sigma^{-1}(u) \leq i \leq \sigma^{-1}(v)\}| \quad (6.57)$$

The Minimum Cut Linear Arrangement (MCLA) problem is to decide if a given graph has $c_\ell(G) \leq k$ for some given k . The reduction starts with an arbitrary graph, then constructs a new bipartite graph, and shows that if we can answer the optimal k -tree embedding problem, then we can answer the original MCLA problem. Note also that, given a candidate k -tree embedding, checking if it is one is easy (must have a perfect elimination order, max clique has to be no more than $k + 1$, and has to be a superset of the edges).

The full proof is long, include it here? □

This therefore means that finding the optimal elimination order, the one that yields the smallest clique, is NP-complete since if we could, we would be able to identify the k threshold that answers the above question either affirmatively or negatively.

Note further that the problem of finding the smallest k is equivalent to the problem of finding the optimal triangulation (the one that has the smallest max clique). Therefore, finding the optimal triangulation is NP-hard.

If we knew the smallest k such that the graph can be embedded into a k tree, then this is possible in polynomial time in n , for given the constant k (and usually the algorithm runs exponential in k but since only n is part of the input, exponential in k is considered a constant).

Aarnborg reference - . Also mention Geiger paper and any more recent results on this.

There are a number of heuristics that attempt to find the best elimination ordering that work quite well in practice, although they provide no guarantees, either constant-factor or approximate, that they are good. They are, in roughly increasing order of complexity.

1. **min fill-in heuristic:** Eliminate next the node n that would result in the smallest number of fill-in edges at that step. Break ties arbitrarily.
2. **min size heuristic:** Eliminate next the node that would result in the smallest clique when eliminated (i.e., choose the node as one with the smallest edge degree). Break ties arbitrarily.
3. **min weight heuristic:** If the nodes have non-uniform domain sizes, then we choose next the node that would result in the clique with the smallest state space, which is defined as the product of the domain sizes. Break ties arbitrarily.

Note that there are a number of variants and improvements to the above three.

1. **tie-breaking:** When one heuristic has a tie on a vertex elimination, then choose one of the other heuristics to break that tie.
2. **non-greedy:** Rather than choosing the vertex that greedily looks best at the moment, take the m -best vertices (e.g., the $m < n$ nodes that would result in, say, the smallest fill-in) and eliminate one of them.
3. **random next step:** Create a distribution over the m -best vertices at any given step, where the probability of eliminating that vertex is either 1) uniform, or 2) inversely proportional to the greedy score (e.g., say the inverse of the required fill-in), and draw from that distribution to choose what node to eliminate next.
4. **random repeats:** Run any of the above complete heuristics multiple times, each time producing a different elimination order. Choose the one that results in the smallest maximum clique size.

In practice, these heuristics can work quite well. Moreover, it is possible to use the maximum cardinality search algorithm (described below) to find a lower bound on the maximum clique size (Lucena 2003 reference). Given such a lower bound, if any of the above heuristics reach that lower bound, we know we would have, in such instances, found the lowest max clique size).

There are a class of related problems that equivalently indicate the difficulty we're in, including the maximum clique problem.

Theorem 85. *Given an arbitrary graph $G = (V, E)$, find the largest clique $C \subseteq V(G)$, where large is measured in terms of $|C|$ is an NP-complete optimization problem.*

If we could find the smallest k such that it could be embedded in a k tree, we could identify the maximum clique in the graph. How? Starting from G , for each of the cliques C in the K -tree cover G' (an $O(n)$ step), we search for the largest complete set in $G[C]$, which is exponential in k but still polynomial in n . That gives us a simple poly-time (in n) algorithm for finding the largest clique, although it is exponential in k , the tree width.

While we're at it, recall that one measure of if we have a triangulated graph is if it has a perfect elimination order, and any order that produces no fill-in edges is perfect. We might consider finding the order that minimizes the number of fill-in edges, but even this is difficult.

Theorem 86. *Given an arbitrary graph $G = (V, E)$, and $G' = (V, E \cup F)$ is a triangulation of G , finding the smallest such F is an NP-complete optimization problem.*

6.8.4 How to identify triangulated graphs

While finding a triangulation of a graph G that minimizes the maximum clique is an NP-complete optimization problem, there is some good news. Specifically, identifying if a given graph is triangulated or not is relatively easy. One way would be to keep eliminating simplicial nodes as long as we can, and if output “not-triangulated” only if we ever run out of simplicial nodes to eliminate. A naive implementation of this would be to find the fill-in of each node, and eliminate the one with no fill-in yielding an $O(n^3)$ algorithm.

There are two common much better methods than this, however, and they are known as *maximum cardinality search* and lexicographic breadth-first search. Both of these methods identify an node ordering in the graph that, if the graph is triangulated, is the reverse of a perfect elimination order.

The first method, maximum cardinality search does the following: All vertices of the graph are given a binary flag indicating if they are labeled or not. We define an ordering of the vertices by the order in which they are chosen by the algorithm. Initially all vertices are unlabeled. We repeatedly choose as a next vertex the one that has the greatest number of previously labeled neighbors (i.e., we choose the vertex

whose set of previously labeled neighbors has the maximum cardinality). Once a variable is chosen it is considered labeled. If at any time these neighbors are not a clique, the graph is not triangulated. Otherwise, the algorithm generates a perfect elimination order in reverse, since considering the the order in reverse, each node so chosen will be simplicial.

Algorithm 11: Maximum Cardinality Search: Determines if a graph G is triangulated.

Input: An undirected graph $G = (V, E)$ with $n = |V|$.
Result: An indication if the graph is triangulated, and if so, a an MCS ordering $\sigma = (v_1, \dots, v_n)$ of the nodes, and the maxcliques in r.i.p. order.

```

1  $L \leftarrow \emptyset ; i \leftarrow 1 ; \mathcal{C} \leftarrow \emptyset ;$ 
2 while  $|V \setminus L| > 0$  do
3   Choose  $v_i \in \operatorname{argmax}_{u \in V \setminus L} |\delta(u) \cap L|$ ;           /*  $v_i$ 's previously labeled neighbors has max
   cardinality. */
4    $c_i \leftarrow \delta(v_i) \cap L$ ;                                /*  $c_i$  is  $v_i$ 's neighbors in the reverse elimination order. */
5   if  $\{v_i\} \cup c_i$  is not complete in  $G$  then
6     return "not triangulated";
7   if  $|c_i| \leq |c_{i-1}|$  then
8      $\mathcal{C} \leftarrow (\mathcal{C}, \{c_{i-1} \cup \{v_{i-1}\}\})$ ;          /* Append the next maxclique to list  $\mathcal{C}$ . */
9   if  $i = n$  then
10     $\mathcal{C} \leftarrow (\mathcal{C}, \{c_i \cup \{v_i\}\})$ ;          /* Append the last maxclique to list  $\mathcal{C}$ . */
11     $L \leftarrow L \cup \{v_i\}$   $i \leftarrow i + 1$ ;
12 return "triangulated", the ordering  $\sigma$ , and the set of maxcliques  $\mathcal{C}$  which are in r.i.p. order.

```

MCS is described in detail in Algorithm 11. Note that other properties of MCS are: MCS can be made to force an elimination order by completing c_i at each stage in the algorithm. This does not tend to produce a good elimination order though. Also, if the graph is chordal, MCS will produce a reverse of a perfect elimination order for that chordal graph. The complexity of the algorithm can be made to run in $O(|V| + |E|)$ (via amortized analysis [96]) using a Fibonacci heap to determine the next label (the one that has maximum cardinality of the set of previously labeled neighbors). Why is the algorithm called maximum cardinality search? First, on line 3, it chooses next the node whose set of previously labeled neighbors has “maximum cardinality.” It is called a “search” since it is a node traversal of the graph, just like breadth-first or depth-first search, except for here the priority queue of nodes which determines which node is next chosen is ordered by the cardinality of the set of previously labeled neighbors.

Theorem 87. A graphical G is triangulated iff in the MCS algorithm, at each point when a vertex is marked, that vertices previously marked neighbors form a complete subgraph of G .

Proof. TODO: add proof here □

Corollary 88. Every maximum cardinality search of a triangulated graph G corresponds to a reverse perfect eliminating order of G .

Note that not all perfect elimination orders can be reverse MCS orders.

Exercise 89. Give an example of a graph where MCS produces a poor elimination order compared to the heuristics. Give an example of a graph where the heuristics produce an unboundedly bad elimination order.

Lexicographic breadth-first search (LBFS) is an alternative (earlier developed) method to identify triangulated graphs and provide a perfect ordering. Again the nodes are ordered in the order that they are

chosen by the algorithm. Again, each vertex might or might not have a label, but if it does, the label consists of a sorted (in decreasing order) list of non-negative integers. All vertices start out without a label, like above. Vertices are chosen by the one that has the largest lexicographic label (breaking ties arbitrarily). Lexicographical order is really “dictionary”, or element-by-element order. To compare two lists, the first number in each list is compared and if unequal determines the order. If the first number is equal, the second number determines the order, and so on until some position down the list yields unequal numbers (thereby determining the order), or one list runs out of numbers. If the lists are identical, they are tied. If one list is a prefix of the other, the longer list is greater. Once a vertex is chosen, its vertex number (the order it is chosen) gets added to the list of all unchosen neighbors, and the process repeats.

Algorithm 12: Lexicographical breadth-first search: Determines if a graph G is triangulated.

Input: An undirected graph $G = (V, E)$ with $n = |V|$.
Result: An indication if the graph is triangulated, and if so, an LBFS ordering $\sigma = (v_1, \dots, v_n)$ of the nodes.

```

1  $L \leftarrow \emptyset ; i \leftarrow 1 ;$ 
2 for  $i = 1$  to  $n$  do
3   Choose as  $v_i$  an unnumbered vertex with lexicographically largest label ;
4    $c_i \leftarrow \delta(v_i) \cap L ;$            /*  $c_i$  is  $v_i$ 's neighbors in the reverse elimination order. */
5   if  $\{v_i\} \cup c_i$  is not complete in  $G$  then
6     return “not triangulated” ;
7   Number  $v$  with  $i$  ;
8   foreach  $w \in \delta(v) \cap L$  do
9     Add  $i$  to  $w$ 's label ;           /* We append  $i$  to the  $w$ 's list and resort the list */
10     $L \leftarrow L \cup \{v_i\}$ 
11 return “triangulated”, the ordering  $\sigma$ , and the set of maxcliques  $\mathcal{C}$  which are in r.i.p. order.

```

MCS looks like a more involved algorithm, only because it does more. Namely, if the graph is triangulated, it produces the sequence of maxcliques in r.i.p. order (see Definition 98). Having the cliques in this order makes it trivial to produce a data structure that is often useful for performing general inference on graphical models. All of this will be described later in this chapter.

These two algorithms both also generate an ordering of the graphs. If the graphs are already triangulated, the orders (in reverse) will be perfect. Otherwise, they can be used to triangulate the graphs. It has not been the case that these algorithms, however, have been shown to produce particularly good orderings. The elimination order heuristics described above tend to do a much better job in practice.

Both algorithms can be run in time $O(|V| + |E|)$ using appropriate data structures. Usually MCS is preferred in practice.

6.9 Multiple Queries

Elimination is sufficient only to produce the result of one particular query, i.e., we wish to sum away some subset of the variables $L \subseteq V$ to obtain the marginal $p(x_{V \setminus L})$.

In many cases we wish to produce multiple marginals at the same time. For example, to perform most machine learning tasks, we will need to have a marginal for every maxclique in the graph. That is, for a given graphical model $p \in \mathcal{F}(G, \mathcal{M}^{(F)})$ and set of maxcliques \mathcal{C} of G we have

$$p(x) = \prod_{c \in \mathcal{C}} \psi_c(x_c)$$

and we desire the marginal $p(x_c)$ for every $c \in \mathcal{C}$.

In Section 6.4, we did something quite similar on trees, where we wished to compute the marginal over every clique which in that case corresponds to each edge in the tree. We saw how we could run a message passing algorithm such that messages for the potential on one edge are re-used for the other edges. The messages in that case corresponded to elimination steps, and were more easily seen as messages when we “forgot” the history of messages leading up to a particular message. For the messages to produce the correct set of marginals, the messages order needed to obey a set of constraints and enough of those messages needed to be sent. The paradigm of messages, rather than elimination steps, moreover allowed us to see how the messages for one query could be reused for another query, and how the amount of message re-use would grow with the number of different queries needed.

The question now is, is it possible to do this on more general graphs than just trees?

First, we know that even to do one elimination procedure, we must elevate the model to the class of triangulated graphs. Since we will need to be in this class anyway even for one query, there is no computational reason to consider anything other than the set of triangulated graphs when computing the desired queries.

On the other hand, the next question might be: is it sufficient to consider only one triangulated cover of a graph for all queries, or is there a benefit to have multiple triangulated graphs?

First, since a triangulated graph is a cover of G , any clique in G will still be a clique in a triangulation G' of G , so given a clique $c \in \mathcal{C}(G)$, there exists $c' \in \mathcal{C}(G')$ with $c \in c'$. Given $p(x_{c'})$ we can always compute $p(x_c) = \sum_{c' \setminus c} p(x_c)$ at a cost no more than the $O(r^{|c'|})$, which is the same cost as the triangulated graph.

Next, given two queries, might one triangulated graph be better for one query than for another? We saw above that the optimal k -tree embedding for G is one that minimizes the maximum clique for any triangulation of G , so if we have found this embedding, this will be optimal for any original-graph clique marginal.

Therefore, even if we found a “good” elimination order (one that produces a maxclique of reasonable size) for one particular query, this order can be shared for other clique queries. Therefore, stage of finding a good elimination order need not be performed each time for each original graph clique or maxclique query.

On the other hand, if we desire non-clique queries, then computation can get worse. Computing $p(x_L)$ for arbitrary L will turn x_L into a clique in the worst case (Rose’s theorem). If x_L is not clique in G' , then G' is not valid for $p(x_L)$. In such case we would need to triangulate a modified graph that has the additional edges necessary to make x_L a clique. Again, for most machine learning applications, however, we desire only clique queries which is what we assume.

Therefore, the work that we do to find a good elimination order for one query is not wasted — we can search for elimination orders using the above heuristics, produce a triangulated graph, and then run perfect elimination on that triangulated graph for each query $p(x_c)$ we wish to compute for every clique c in the original graph.

On the other hand, it seems like the “sharing” only of the process that triangulated the graph, but not of the actual perfect elimination procedures could be quite wasteful. Intuitively, since we now know that a triangulated graph can be embedded into a k -tree, and we’ve seen how “tree like” k -trees are, it seems that many of the actual numerical elimination steps for one query could be reused for another query. Can we view the problem of inference as a more general tree inference?

This indeed turns out to be the case, and the answer ideally would be at least intuitively clear why from the above. We will formalize in our journey through the next several sections. Along the way, we will need a number of additional theorems and concepts that will help to unify our understanding of how all graphical models can be seen as generalizations of trees. These theorems will also help us to prove our main results.

We start with the notion of a decomposable model.

Definition 90 (Decomposition of G). A decomposition of a graph $G = (V, E)$ (if it exists) is a partition

(A, B, C) of V such that:

- C separates A from B in G .
- C is a clique.

if A and B are both non-empty, then the decomposition is called proper.

If G has a decomposition, what does this mean for the family $\mathcal{F}(G, \mathcal{M}^{(f)})$? Since C separates A from B , this means that $X_A \perp\!\!\!\perp X_B | X_C$ for any $p \in \mathcal{F}(G, \mathcal{M}^{(f)})$, which moreover means we can write the joint distribution in a particular form.

$$p(x_A, x_B, x_C) = \frac{p(x_A, x_C)p(x_B, x_C)}{p(x_C)} \quad (6.58)$$

Note that we take the above factorization to be zero if $p(x_C) = 0$ since both the numerators factors and denominator will be zero in this case.

We moreover can define the class of graphs which have recursive decomposition, with the base case being cliques. These are otherwise known as *decomposable graphs*:

Definition 91. A graph $G = (V, E)$ is decomposable if either: 1) G is a clique, or 2) G possesses a proper decomposition (A, B, C) s.t. both subgraphs $G[A \cup C]$ and $G[B \cup C]$ are decomposable.

Stated another way, a decomposable graph means that we can partition V as $V = A \cup B \cup C$ so that A , B and C are disjoint, A and B are non-empty, C is a clique, C separates A from B in G , and each vertex-induced subgraph $G[A \cup C]$ and $G[B \cup C]$ are decomposable (and this applies recursively). Note that the separator is contained within the subgraphs: i.e., $G[A \cup C]$ is what is required to recursively be decomposable, not, say, $G[A]$.

A decomposable graph can be represented by a binary tree where each node of the tree represents the application of the theorem one time. This is called a *binary decomposition tree* (see Figure 6.23). Each internal node of the decomposition tree corresponds to a separator, and each leave node corresponds to a clique. There are many different binary decomposition trees for a given decomposable graph.

Note that in the second part in the definition of a decomposable model, it uses the word “possesses.” This means only that it must be *possible* to find such a division — there could be many such divisions that allow for a recursive decomposition almost but not entirely the way to the bottom case. Due to the recursive definition, not finding a decomposition way down at the bottom of the decomposition tree can affect the finding of such a proper recursively decomposable decomposition at the top of the tree.

If G is decomposable, what does this mean for members of the corresponding family $\mathcal{F}(G, \mathcal{M}^{(f)})$? We see from Figure 6.23-middle that each of the decompositions forming the decomposable graph correspond to internal nodes in the decomposition tree, and the leaves correspond to cliques. Moreover, the internal nodes correspond to separators in the graph that are also required to be cliques. The leaf nodes in the decomposition tree are required to be maxcliques, since that is the base-case of a decomposition (see below). We can use this recursive decomposition to expand on the joint as we do in the following:

$$p(A, B, C, D, E, F, G, H, I, J, K) = \frac{p(A, C, D, F)p(B, C, D, E, F, G, H, I, J, K)}{p(C, D, F)} \quad (6.59)$$

$$= \frac{p(A, C, D, F)}{p(C, D, F)} \left(\frac{p(B, C, G, H)p(C, D, E, F, H, I, J, K)}{p(C, H)} \right) \quad (6.60)$$

$$= \dots \quad (6.61)$$

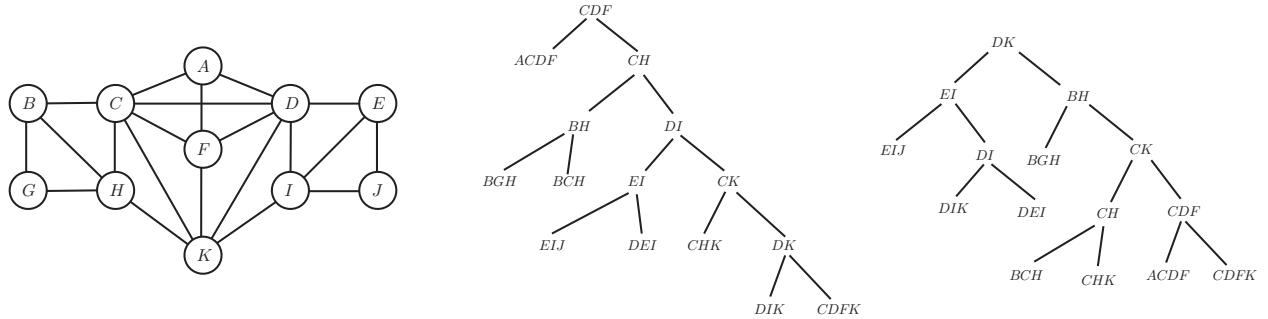


Figure 6.23: A decomposable graph on the left, and a two tree representations of the decomposition of this graph. In each case, the root of the tree is the first set of nodes that splits the graph, and then we recursively traverse on the left and the right forming decompositions for each subgraph, until we reach the leaf nodes which are all cliques. If it was the case that at any point a decomposition could not be found, the graph would not be decomposable. There are many decompositions, only two of which are shown here.

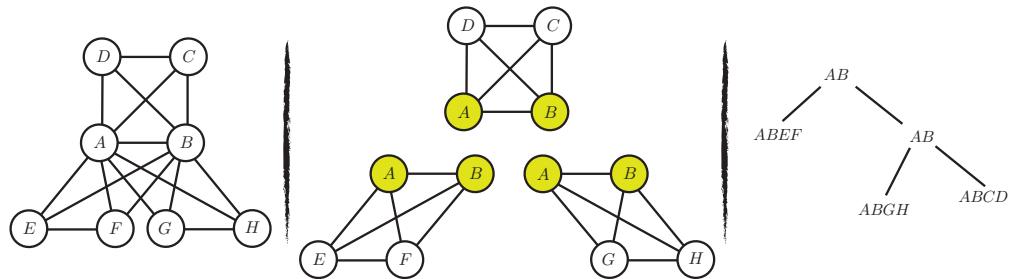


Figure 6.24: On the left, we see a decomposable model with one minimal separator A, B . This separator is used multiple times in the binary decomposition tree on the right. The separator S has shattering coefficient $d(S) = 3$ since it separates the graph into three connected components (shown at the center where the separator is included in each component).

$$= \frac{p(A, C, D, F)p(B, G, H)p(C, B, H)p(I, E, J)p(E, I, D)p(C, K, H)p(D, K, I)p(D, K, F, C)}{p(C, D, F)p(C, H)p(B, H)p(D, I)p(E, I)p(C, K)p(D, K)} \quad (6.62)$$

By the nature of S being a separator, we know that S will break the graph into connected components. That is $G[V \setminus S]$ consists of at least two connected components, namely those components that are separated by S . A given S however might break a graph into more than two pieces (Figure 6.24), however. We use the term *shatter* to refer to this concept. That is a separator is said to shatter a graph into some number of connected components, and we use the notation $d(S)$ to refer to the number of components S shatters G into. We will call $d(S)$ the *shattering coefficient* of separator S . Note that we always have $d(S) \geq 2$.

We note that separator in a binary decomposition tree can be used multiple times and this depends on S 's shattering coefficient. That is, it could be that S shatters the graph in to more than 2 connected components implying that on at least one side of a decomposition, the given separator S is still a separator of a proper decomposition, requiring S to be used more than once in the tree (we note that a given S must be used at some point since the existence of a complete S that creates a decomposition means we have not yet reached the base case. Figure 6.24 shows a case where $d(S) = 3$, and where $d(S)$ is used twice in the binary decomposition tree.

When $d(S) > 2$, it moreover means that the separator marginal would be used more than once in the denominator in the factorization. Specifically, the number of times that separator is used in the denominator is $d(S) - 1$ and this it becomes the exponent. The general form of the factorization becomes:

$$p(x) = \frac{\prod_{C \in \mathcal{C}(G)} p(x_C)}{\prod_{S \in \mathcal{S}(G)} p(x_S)^{d(S)-1}} \quad (6.63)$$

In general, we see that for any p in the family of decomposable models, we can write the probability as the ratio of leaf cliques factors in the decomposition tree, to the non-leaf-node clique factors in the decomposition tree.

Note that the 4-cycle is not decomposable since it does not have a decomposition. In fact, notice that there are two independent properties of the 4 cycle, but they can't both be used at the same time with regard to factorization. That is, considering Figure 6.16, we have that $X_1 \perp\!\!\!\perp X_4 | \{X_2, X_3\}$ and $X_2 \perp\!\!\!\perp X_3 | \{X_1, X_4\}$ which means we can factor as either of the following two forms:

$$p(x_1, x_2, x_3, x_4) = \frac{p(x_1, x_2, x_4)p(x_1, x_3, x_4)}{p(x_1, x_4)} = \frac{p(x_1, x_2, x_3)p(x_2, x_3, x_4)}{p(x_2, x_3)} \quad (6.64)$$

The first form uses the first independence property and the second form uses the second, but note that there is no way to factor the joint using both forms simultaneously. This seems suspiciously similar to the property of the four cycle we saw above with regard to elimination.

So it seems like the graphs that are decomposable are such that the marginals can be expressed in a special way, where marginals over the maxcliques are in the numerator, and with marginals over separators in the denominator.

In fact, decomposable models are ones for which all implied conditional independence statements made by the graph may be used simultaneously to define a factorization of the graph as in the above form.

Notice that all of the maxcliques in a graph lie on the leaf nodes of the binary decomposition tree (the base case).

Proposition 92. *All of the maxcliques in a graph lie on the leaf nodes of the binary decomposition tree*

Proof. For a decomposable model, the base case (leaf node) is a clique, otherwise it would not be decomposable. If a leaf was not a maxclique, then that means it is contained in a maxclique, and got split by a separator corresponding to that leaf's parent, but this is impossible since a maxcliques have no separator. \square

Proposition 93. *The (nec. unique) set of all minimal separators of graph are included in the non-leaf nodes of the binary decomposition tree, with $d(S) - 1$ being the number of times the minimal separator S appears as a given non-leaf node.*

What lies at the interior nodes of the tree are the separators because of the definition. In fact, we note that all of the separators are complete by definition. We noted above that triangulated graphs have all minimal separators complete. Also, note that as soon as we chord the 4-cycle (thereby triangulating it), then it becomes decomposable

In fact the connection is much stronger than this as we show in the next theorem.

Before doing so, we introduce a bit of notation. First, as we know, $G = (V, E) = (V(G), E(G))$ is a graph. Suppose that $C \subset V$ is a separator, that means removing C from G breaks (or *shatters*) G into $d(C) \geq 2$ pieces, each of which is connected. We represent the union of these pieces as $G[V \setminus C]$ which is the vertex-induced subgraph of G , induced by vertices $V \setminus C$. Each of these pieces are called the *connected components* of $G[V \setminus C]$. In fact, we may define a separator as a set C such that $G[V \setminus C]$ is disconnected. Suppose that the subgraphs G_1, G_2, \dots, G_ℓ are the connected components of $G[V \setminus C]$, which means that $G_1 \cup G_2 \cup \dots \cup G_\ell = G[V \setminus C]$ where we use the obvious notion of graph union. Next, suppose that $a \in V(G_i)$ for some i . Then we define notation that allows us to use a to refer to the connected component containing a in $G[V \setminus C]$ as $G[V \setminus C](a)$. That is, $G[V \setminus C](a) = G_i$.

We next show that the class of decomposable graphs is the same as the class of triangulated graphs.

Theorem 94. *A given graph $G = (V, E)$ is triangulated iff it is decomposable.*

Proof. First prove that decomposability implies triangulated using induction. We then prove that if every minimal separator is complete in G (which we know to be equivalent to triangulated by Theorem ?? above), then G is decomposable.

Assume G is decomposable. If G is complete then it is triangulated. If it is not complete then there exists a proper decomposition (A, B, C) into decomposable subgraphs $G[A \cup C]$ and $G[B \cup C]$ both of which have fewer vertices, meaning $|A \cup C| < |V|$ and $|B \cup C| < |V|$. By the induction hypothesis, both $G[A \cup C]$ and $G[B \cup C]$ are chordal. Any potential chordless cycle, therefore, can't be contained in one of the sub-components, so if it exist in G must intersect both A and B . Since C separates A from B , the purported chordless cycle would intersect C twice, but C is complete the cycle has a chord.

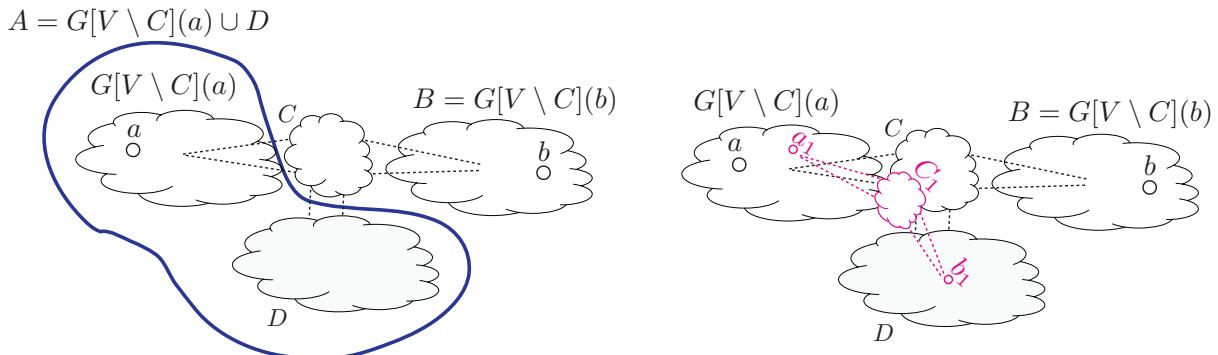


Figure 6.25: Left: (A, B, C) forms a decomposition. Right: C is still a minimum (a, b) -separator in G .

Next, assume that all minimum (a, b) separators are complete in G . If G is complete then it is decomposable. Otherwise, there exists two non-adjacent vertices $a, b \in V$ in G with a necessarily complete minimal separator C forming a partition $G[V \setminus C](a), G[V \setminus C](b)$, and all of the remaining components of $G[V \setminus C]$. We merge the connected components together to form only two components as follows:

let $A = G[V \setminus C](a) \cup D$ and $B = G[V \setminus C](b)$. Since C is complete, we see that (A, B, C) form a decomposition of G , but we still need that $G[A \cup C]$ and $G[B \cup C]$ to be decomposable.

Let C_1 be a minimal (a_1, b_1) separator in $G[A \cup C]$. But then C_1 is also a minimal (a_1, b_1) separator in G since, once we add B back to $G[A \cup C]$ to regenerate G , there still cannot be any new paths from a_1 to b_1 circumventing C_1 . This is because any such path would involve nodes in B (the only new nodes) which, to reach B and return, requires going through C (which is complete) twice. Such a path cannot bypass C_1 since if it did, a shorter path not involving B would bypass C_1 . Therefore, C_1 is complete in G , and an inductive argument says that $G[A \cup C]$ is decomposable. The same argument holds for $G[B \cup C]$. Therefore, G is decomposable. □

A further implication is that hat triangulated graphs always have cliques as leaf nodes in the decomposition tree. Basically, if the graph is chordal, eventually we get down to an leaf that is not decomposable (i.e., we can't find a clique separator). If the leaf was not a clique, we could find a clique separator. If the leave had a chordless cycle, then that cycle would be chordless in the entire graph since the parent "separator" is a clique and the chordless cycle could not go through that separator to the rest of the graph and come back (since that would give it a chord) — in fact, the cycle could only have one edge in that clique. Therefore, the only remaining option is for the leaf to be a clique. Moreover, the leaves must be maximal cliques, since it if a leaf was not a maximal clique it could not have a corresponding parent separator (cliques do not have separators).

So why are we spending so much time on graph decomposability? Because they will help us show that while an arbitrary graph is not a tree, it is possible to, in some sense, embed an arbitrary graph into another graph that is tree-like enough so that essentially everything that we showed in Section 6.2 is still true. The way we do this is to essentially cluster multiple nodes in the original graph into a type of "hyper" node in the new graph (contrast this with Lemma 82). In fact, these new graphs have a representation that is exactly a tree and on which the very same message passing algorithm, messaging sharing, and dynamic programming, can be defined. It is not the case, however, that any clustering of the nodes and the stitching of those clusters together into a tree will work with the tree-inference algorithms mentioned earlier. When we discussed regular tree graphs, every node x_i was its own representative in the graph — messages came into the node, and messages left that node, but there is only one location in the graph where the "definitive" answer regarding the state of the variable x_i is located, and that is the node x_i . Moreover, when we considered the edges adjacent to node x_i and all of the edge factor functions associated with those edges, the variable x_i was always present and involved in all message communications involving x_i . In some sense, there is no way in a standard tree graph for the various locations where information about x_i is represented (i.e., node x_i and the edge functions adjacent to x_i) could get out of sync.

When we cluster variables, however, it will be necessary for nodes to reside in more than one cluster. The reason for this is that, if this was not the case, i.e., the set of clusters partitioned the nodes into disjoint subsets, then there would be no way for neighboring clusters to interact. In the tree graph, for example, one node interacts with another node far away in the tree indirectly via the nodes that lie along the unique paths between those nodes. Messages are only defined along edges, which are the conduit, or communications channel, through which nodes interact. If clusters had no nodes in common, that would sever that communications channel. On the other hand, the key property that a cluster of nodes will need to have is that as information from one node x_i to another node x_j travels along clusters, that there is no break in the representation of a node x_i until it is finally forgotten. This means that there cannot be a way for the multiple representations of a given node to get out of sync with respect to each other. We will make this more precise below (the reader should consider returning to this section once having read the definitions below). For now, the representation (or hyper-tree, if you will) that satisfies this property is called a junction tree. Before defining a junction tree, we have a few subsidiary definitions to get out of the way.

Definition 95 (Cluster graph). Consider forming a new graph based on G where the new graph has nodes that correspond to clusters in the original G , and has edges existing between two (cluster) nodes only when the corresponding clusters have a non-zero intersection. That is, let $\mathcal{C}(G) = \{C_1, C_2, \dots, C_{|I|}\}$ be a set of $|I|$ clusters of nodes $V(G)$, where $C_i \subseteq V(G), i \in I$. Consider a new graph $\mathcal{J} = (I, \mathcal{E})$ where each node in \mathcal{J} corresponds to a set of nodes in G , and where edge $(i, j) \in \mathcal{E}$ if $C_i \cap C_j \neq \emptyset$. We will also use $S_{ij} = C_i \cap C_j$ as notation.

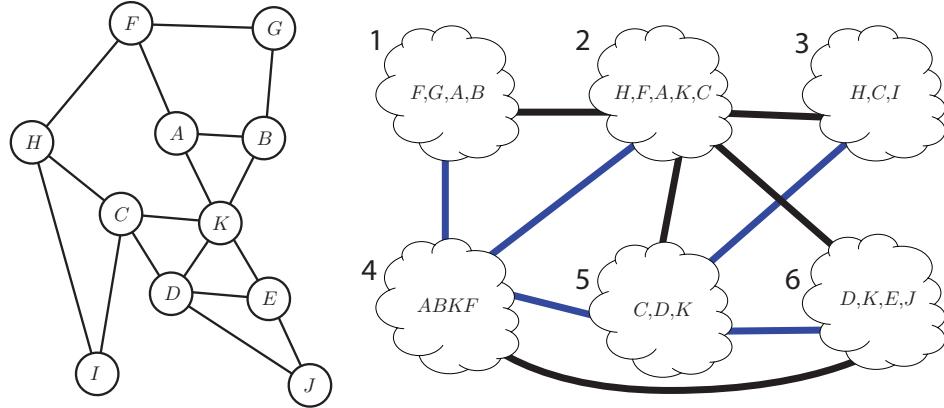


Figure 6.26: Left: a graph. Right: A cluster graph with $|I| = 6$ clusters, where $C_1 = \{F, G, A, B\}, C_2 = \{H, F, A, K, C\}, \dots$. There is an edge $(1, 2)$ since $C_1 \cap C_2 = \{F, A\} \neq \emptyset$. If we remove either the blue or the black edges, then we get a cluster tree.

If we relax the definition a bit (i.e., drop the requirement that an edge exists whenever there is intersection between the corresponding two clusters), and the graph is also a tree, then we have what is called a cluster tree.

Definition 96 (Cluster Tree). Let $\mathcal{C} = \{C_1, C_2, \dots, C_{|I|}\}$ be a set of node clusters of graph $G = (V, E)$. A cluster tree is a tree $\mathcal{T} = (I, \mathcal{E}_T)$ with vertices corresponding to clusters in \mathcal{C} and edges corresponding to pairs of clusters $C_1, C_2 \in \mathcal{C}$. We can label each vertex in $i \in I$ by the set of graph nodes in the corresponding cluster in G , and we label each edge $(i, j) \in \mathcal{E}_T$ by the cluster intersection, i.e., $S_{ij} = C_i \cap C_j$.

Note that in the above, we have deliberately not defined a cluster in the above graph either to be a clique or a maxclique. Indeed, in some cases each cluster might correspond to a clique and in other cases, each cluster might correspond to a maxclique, and at other times arbitrary sets of subsets of V are used. The reason we do not do this here is that the crucial properties we wish to study do not require either that all cliques in G are represented nor that the nodes of the clique graph are maxcliques. Also, once we require additional properties of a cluster tree, it might not be possible to form such a cluster tree where clusters correspond to cliques or maxcliques — to do so would require modifying the graph.

For most of the computations, that exploit these properties, however, it will be the case that the graph has nodes that correspond to all maxcliques of G .

A *junction tree* is a cluster tree that (if it exists) has a special property that, among other things, allows the message passing algorithm on trees that we have seen above to produce mathematically valid probabilistic inference for any distribution that is in the original graph family. We define a number properties of sets of clusters.

Definition 97 (Cluster Intersection Property (c.i.p.)). We are given a cluster tree $\mathcal{T} = (I, \mathcal{E}_T)$, and let C_1, C_2 be two clusters in the tree. Then the cluster intersection property states that $C_1 \cap C_2 \subseteq C_i$ for all C_i on the (necessarily unique) path between C_1 and C_2 in the tree \mathcal{T} .

A given cluster tree might or might not have this property. The above property that the intersection of any two nodes in the tree is contained in the (by the definition of a tree, necessarily) unique path between those two nodes is identical to the *running intersection property*, or just r.i.p. Given a set of clusters or subsets of nodes, r.i.p. means that the clusters can be ordered in a particular way as defined next.

Definition 98 (Running Intersection Property (r.i.p.)). *Let C_1, C_2, \dots, C_ℓ be an ordered sequence of subsets of $V(G)$. Then the ordering obeys the running intersection property (r.i.p.) property if for all $i > 1$, there exists $j < i$ such that $C_i \cap (\cup_{k < i} C_k) = C_i \cap C_j$.*

Note that the running intersection property is defined base not necessarily on cliques or even maxcliques in the graph, but based on any subsets of the vertices $V(G)$ of G .

It is perhaps easier to understand r.i.p. by expanding out and explaining individually various terms. Suppose we are given a sequence of cliques C_1, C_2, \dots, C_ℓ . First, we define the history (or accumulation) of the sequence at position i as:

$$H_i = C_1 \cup C_2 \cup \dots \cup C_i. \quad (6.65)$$

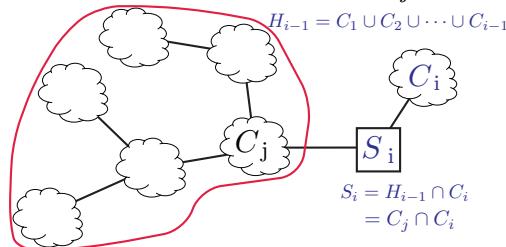
Next, we define the innovation, or residual, or new nodes in C_i not encountered in the previous history, as:

$$R_i = C_i \setminus H_{i-1}. \quad (6.66)$$

Lastly, we define the non-innovation, commonality, or separation elements between the new i^{th} clique and the previous history

$$S_i = C_i \cap H_{i-1} \quad (6.67)$$

Note that $C_i = R_i \cup S_i$, so the i^{th} clique consists of the innovation R_i and the commonality S_i . The cliques are in r.i.p. order if the commonality S_i between the new clique and the history is fully contained in one element of that history. I.e., there exists an $j < i$ such that $S_i \subseteq C_j$.



Now imagine constructing a tree with clusters as nodes by starting with C_1 , and then for each position $i = 1 \dots \ell$, connecting an edge between C_i and the corresponding C_j . Clearly, this will produce a tree (consider the generative property of trees mentioned in Theorem 41). Moreover, this order will produce a tree that satisfies the cluster intersection property – take any C_i and C_k and suppose that C_i is later in the order than C_k . By an inductive argument, we know that S_i summarizes everything common between H_{i-1} and C_i , and so $C_i \cap C_k \subseteq S_i$. Recursively this argument for the latter node of C_j and C_i means that $C_i \cap C_k$ is contained on the unique path between them in this tree.

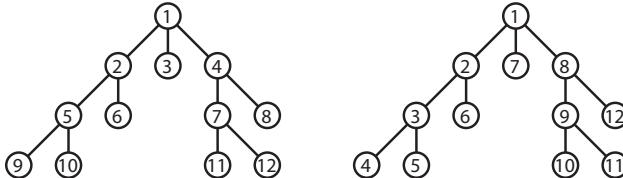
The converse is true as well, as if we perform an traversal (e.g., a depth or breadth first search, where we visit and then number the parent node before any of the children) of a cluster tree that satisfies the cluster intersection property to produce an ordered sequence of cliques, we have that for every C_i , and C_j with $j < i$, $C_i \cap C_j$ is a subset of every node on the unique path from C_i to C_j . If $C_{\pi(i)}$ is C_i 's parent in the junction tree, then we must have that for any $j < i$, $C_i \cap C_j \subseteq C_i \cap C_{\pi(i)}$

Lemma 99. *The cluster intersection and running intersection properties are identical.*

Proof. Starting with clusters in r.i.p. order, construct cluster tree by connecting each i to its corresponding j node. This is a tree. Also, take any pair C_k, C_i and assume w.l.o.g. that $k < i$ and hence $C_k \subseteq H_{i-1}$.

Then $C_i \cap C_k \subseteq C_i \cap H_{i-1} = S_i \subseteq C_j$. Note that C_j is one node closer to C_k on the path. Repeat this process, but with pair C_k, C_j (if $k < j$) or C_j, C_k (if $j < k$) which decreases the path by one edge, until we get adjacent clusters. This shows c.i.p.

Conversely, perform a tree traversal (depth or breadth first search) on cluster tree to produce node ordering, as shown.



Then by c.i.p., for any i in that order, and any $k < i$, $C_i \cap C_k \subseteq C_j$ for any j on the unique path between k and i . In particular, $C_i \cap C_k \subseteq C_j$ for $j < i$ being i 's neighbor in the tree. Then $\bigcup_{k < i} (C_i \cap C_k) \subseteq C_j$ implying $C_i \cap \bigcup_{k < i} C_k \subseteq C_j$ and so $C_i \cap \bigcup_{k < i} C_k \subseteq C_i \cap C_j$. On the other hand, we always have that $C_i \cap C_j \subseteq C_i \cap \bigcup_{k < i} C_k$, and the two together give us r.i.p. \square



Figure 6.27: Shown is an example of a set of node clusters (within the cloud-like shapes) arranged in a tree that satisfies the r.i.p. and also the cluster intersection property. The intersections between neighboring node clusters are shown in the figure as square boxes. Consider the path or $\{B, E, H\} \cap \{B, D, F\} = \{B\}$.

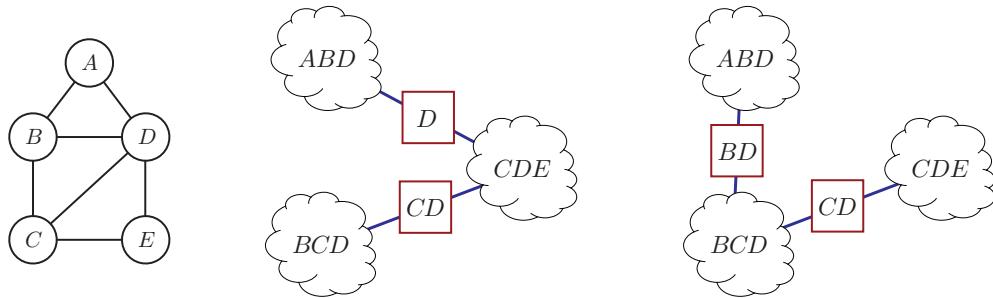


Figure 6.28: This example shows that not all trees of maxcliques satisfy the r.i.p. and therefore not all trees of maxcliques are junction trees. Consider the graph on the left which has three maxcliques. The middle tree of maxcliques is not a junction tree since a traversal of the tree does not produce a r.i.p. order. For example, $\{A, B, D\} \cap (\{B, C, D\} \cup \{C, D, E\}) \neq \{A, B, D\} \cap \{C, D, E\}$. The reason why this shows that r.i.p. is not satisfied is that $\{C, D, E\}$, which is what is connected to $\{A, B, D\}$ in the junction tree, is not representative of everything in the history in this traversal (which is $\{B, C, D\} \cup \{C, D, E\}$). Equivalently, we see that $\{B, D\} = \{A, B, D\} \cap \{B, C, D\} \not\subseteq \{C, D, E\}$ meaning that the intersection is not a subset of all cliques on the unique path between two nodes. The right tree, however, is a junction tree since it satisfies the required properties.

Figure 6.27 shows an example of a set of clusters of nodes (as clouds) that satisfy r.i.p. and the junction tree property. As can be seen, the intersection of any pair of clusters is a subset of all of the clusters on the unique path between those two clusters. For example, $\{B, E, H\} \cap \{B, D, F\} = \{B\}$ is within every cluster in the tree. Another few examples corresponding to a graph is given in Figure 6.28. The graph is on

the left, and this time the clusters correspond to maxcliques in the graph. The middle tree is not a junction tree while the right tree is.

Yet another way of stating the junction tree property is that if we take any $v \in V$, and then consider the clusters $C \in \mathcal{C}$ such that $v \in C$, then those clusters will produce a sub-tree (necessarily connected). This key property, called the *induced sub-tree property*, means that it is never possible to “forget” about any node with respect to constructing marginals using only neighboring clusters (more on this below when we discuss message passing on a junction tree).

Definition 100 (Induced Sub-tree Property). *Given a cluster tree \mathcal{T} for graph G , the induced sub-tree property is true if for all $v \in V$, the set of cliques $C \in \mathcal{C}$ such that $v \in C$ induced a sub-tree $\mathcal{T}(v)$ of \mathcal{T} .*

If the induced sub-tree property holds then the clique intersection property holds. This is because if we take all $v \in C_i \cap C_j$, then each such v induces a sub-tree of \mathcal{T} , and all of these sub-trees overlap on the unique path between C_i and C_j in \mathcal{T} . On the other hand, if the clique intersection property holds, then the induced sub-tree property holds as well. Given a $v \in V$, consider all maxcliques that contain v , $\mathcal{C}(v) = \{C \in \mathcal{C} : v \in C\}$. For any pair $C_1, C_2 \in \mathcal{C}(v)$, we have that $C_1 \cap C_2$ exists on the unique path between C_1 and C_2 in \mathcal{T} , and since $v \in C_1 \cap C_2$, v always exists on each of these paths. These paths, considered as a union together, cannot form a cycle (since they are paths on a tree). Moreover, these paths unioned together form a tree (they’re connected) since for any $C_1, C_2, C_3 \in \mathcal{C}(v)$, the path between C_1 and C_2 is connected to the path between C_2 and C_3 via v .

Lemma 101. *Induced sub-tree property holds whenever cluster intersection property holds*

Therefore, we see that the clique intersection property, the running intersection property, and the induced sub-tree property are identical. We will henceforth refer to all of them as r.i.p.

A tree decomposition is a cluster tree that satisfies induced sub-tree property (e.g., r.i.p. and c.i.p. as well). That is:

Definition 102 (tree decomposition). *Given a graph $G = (V, E)$, a tree-decomposition of a graph is a pair $(\{C_i : i \in I\}, T)$ where $T = (I, \mathcal{E}_T)$ is a tree with node index set I , edge set \mathcal{E}_T , and $\{C_i\}_i$ (one for each $i \in I$) is a collection of clusters (subsets) of $V(G)$ such that:*

1. $\cup_{i \in I} C_i = V$
2. for any edge $(u, v) \in E(G)$, there exists $i \in I$ with $u, v \in C_i$
3. (r.i.p.) for any $v \in V$, the set $\{i \in I : v \in C_i\}$ forms a (nec. connected) subtree of T

The tree-width of the tree-decomposition is the size of the largest C_i minus one (i.e., $\max_{i \in I} |C_i| - 1$). Finding a tree-decomposition with minimal tree-width, however, is NP-complete.

Theorem 103. *Given graph $G = (V, E)$, finding the tree decomposition $T = (I, F)$ of G that minimizes the tree width ($\max_{i \in I} |C_i| - 1$) is an NP-complete optimization problem.*

Note that this is (multiplicatively) approximable within $O(\log |V|)$ but it is not possible to (additively) do better than $|V|^{1-\epsilon}$ for any $\epsilon > 0$.

How does this relate to our problem though? It relates via our next definition, the junction tree, a tree of cliques that posses r.i.p.

Definition 104. *Given a graph $G = (V, E)$, a junction tree corresponding to G (if it exists) is a cluster tree $\mathcal{T} = (\mathcal{C}, \mathcal{E}_T)$ having the clique intersection property over the clusters, and where those nodes u, v adjacent to every edge $(u, v) \in E(G)$ are together in at least one cluster.*

Note that a junction tree could be defined on the cliques or it could be defined on the maxcliques (as we have done). For example, given a junction tree of maxcliques, we'll still have a junction tree if we take each maxclique, and form a chain of subsets of every maxclique and appending the junction tree with that chain. That is, r.i.p. will still hold. We can do the other direction as well by merging non-max cliques into neighboring superset cliques until we have only maxcliques left (note, the intersection graph example below does this to some extent, add a pointer to that).

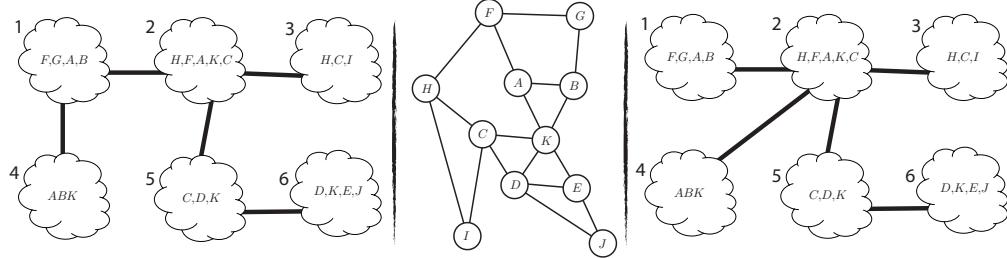


Figure 6.29: Given the middle graph, we have two examples (left and right) of cluster trees. Neither of the cluster trees are junction trees for this graph because a junction tree's nodes correspond to cliques in the original graph. Note that even if we were to make each cluster a clique (by completing all of the nodes in the graph on the left) the two trees would not be junction trees since r.i.p. is not satisfied.

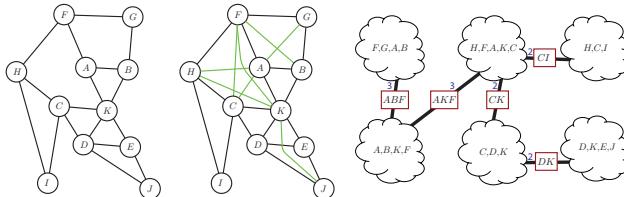


Figure 6.30: Given the graph on the left, the right example is junction tree if we were to complete all of the nodes in each cluster.

Figure 6.28 shows that not all trees of maxcliques satisfy the r.i.p. and that not all trees of maxcliques are junction trees as there is no r.i.p. order in middle case.

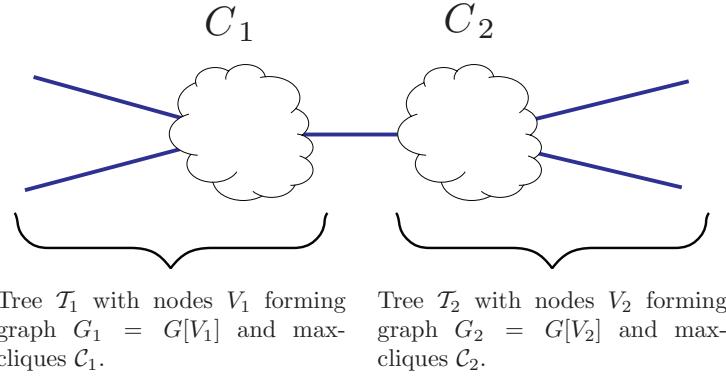
So far we have defined a junction tree as being one that might exist on any set of cliques G .

As you might have guessed, we have the following key theorem that relates the decomposability of G to a junction tree over *all* of the maxcliques in G .

Theorem 105. *A graph $G = (V, E)$ is decomposable iff a junction tree of maxcliques for G exists.*

Proof. a junction tree exists \Leftrightarrow decomposable: We prove this using induction on the number of maxcliques. Clearly, if G has only one maxclique then it is both a junction tree and decomposable. This is similarly true for G having exactly two maxcliques. Assume it is true for $\leq k$ maxcliques and we show it for $k + 1$.

a junction tree exists \Rightarrow decomposable: Let \mathcal{T} be a junction tree of maxcliques \mathcal{C} , and let C_1, C_2 be adjacent in \mathcal{T} . The edge C_1, C_2 in the tree separates \mathcal{T} into two sub-trees \mathcal{T}_1 and \mathcal{T}_2 , with V_i being the nodes in \mathcal{T}_i , $G_i = G[V_i]$ being the subgraph of G corresponding to \mathcal{T}_i , and \mathcal{C}_i being the set of maxcliques in \mathcal{T}_i , for $i = 1, 2$. Thus $V(G) = V_1 \cup V_2$, and $\mathcal{C} = \mathcal{C}_1 \cup \mathcal{C}_2$. Note that $\mathcal{C}_1 \cap \mathcal{C}_2 = \emptyset$. We also let $S = V_1 \cap V_2$ which is the intersection of all the nodes in each of the two trees.



Also, the nodes in T_i are maxcliques in G_i and T_i is a junction tree for G_i since r.i.p. still holds in the subtrees of a junction tree. Therefore, by induction, G_i is decomposable. To show that G is decomposable, we need to show that: 1) $S = V_1 \cap V_2$ is complete, and 2) that S separates $G[V_1 \setminus S]$ from $G[V_2 \setminus S]$.

If $v \in S$, then for each G_i ($i = 1, 2$), there exists a clique C'_i with $v \in C'_i$, and the path in T joining C'_1 and C'_2 passes through both C_1 and C_2 . Because of the r.i.p., we thus have that $v \in C_1$ and $v \in C_2$ and so $v \in C_1 \cap C_2$. This means that $V_1 \cap V_2 \subseteq C_1 \cap C_2$. But $C_i \subseteq V_i$ since C_i is a clique in the corresponding tree T_i . Therefore $C_1 \cap C_2 \subseteq V_1 \cap V_2 = S$, so that $S = C_1 \cap C_2$. This means that S contains all nodes that are common among the two subgraphs and moreover that S is complete as desired.

Next, to show that S is a separator, we take $u \in V_1 \setminus S$ and $v \in V_2 \setminus S$ (note that such choices mean $u \notin V_2$ and $v \notin V_1$ due to the commonality property of S). Suppose the contrary that S does not separate V_1 from V_2 , which means there exists a path $u, w_1, w_2, \dots, w_k, v$ for the given u, v with $w_i \notin S$ for all i . Therefore, there is a clique $C \in \mathcal{C}$ containing the set $\{u, w_1\}$. We must have $C \notin \mathcal{C}_2$ since $u \notin V_2$, which means $C \in \mathcal{C}_1$ or $C \subseteq V_1$ implying that $w_1 \in V_1$ and moreover that $w_1 \in V_1 \setminus S$. We repeat this argument with w_1 taking the place of u and w_2 taking the place of w_1 in the path, and so on until we end up with $v \in V_1 \setminus S$ which is a contradiction. Therefore, S must separate V_1 from V_2 . We have thus formed a decomposition of G as $(V_1 \setminus S, V_2 \setminus S, S)$ and since G_i is decomposable (by induction), we have that G is decomposable.

decomposable \Rightarrow a junction tree exists: Since G is decomposable, let (W_1, W_2, S) be a proper decomposition of G into decomposable subsets $G_1 = G[W_1]$ and $G_2 = G[W_2]$ with $V_i = W_i \cup S$. By induction, since G_1 and G_2 are decomposable, there exists a junction tree T_1 and T_2 corresponding to maxcliques in G_1 and G_2 . Since this is a decomposition, with separator S , we can form all maxcliques $\mathcal{C} = \mathcal{C}_1 \cup \mathcal{C}_2$ with \mathcal{C}_i maxcliques of V_i for tree T_i . Choose $C_1 \in \mathcal{C}_1$ and $C_2 \in \mathcal{C}_2$ such that $S \subseteq C_1$ and $S \subseteq C_2$ which is possible since S is complete, and must be contained in some maxclique in both T_1 and T_2 . We form a new tree T by linking $C_1 \in T_1$ with $C_2 \in T_2$. We need next to ensure that this new junction tree satisfies r.i.p.

Let $v \in V$. If $v \notin V_2$, then all cliques containing v are in \mathcal{C}_1 and those cliques form a connected tree by the junction tree property since T_1 is a junction tree. The same is true if $v \notin V_1$. Otherwise, if $v \in S$ (meaning that $v \in V_1 \cap V_2$), then the cliques in \mathcal{C}_i containing v are connected in T_i including C_i for $i = 1, 2$. But by forming T by connecting C_1 and C_2 , and since v is arbitrary, we have retained the junction tree property. Thus, T is a junction tree. □

Obviously, this means that all triangulated graphs have junction trees of max-cliques and vice versa. Intuitively, a junction tree expresses things in terms of a decomposition, or factorization, where the separators in a junction tree correspond to the separators that are conditioned on, all of which are used when forming the decomposition of the distribution.

Lemma 106. *A junction tree of maxcliques for graph $G = (V, E)$ exists iff a junction tree of cliques for*

graph $G = (V, E)$ exists.

Proof.

Corollary 107. A graph G is triangulated iff a junction tree for G exists.

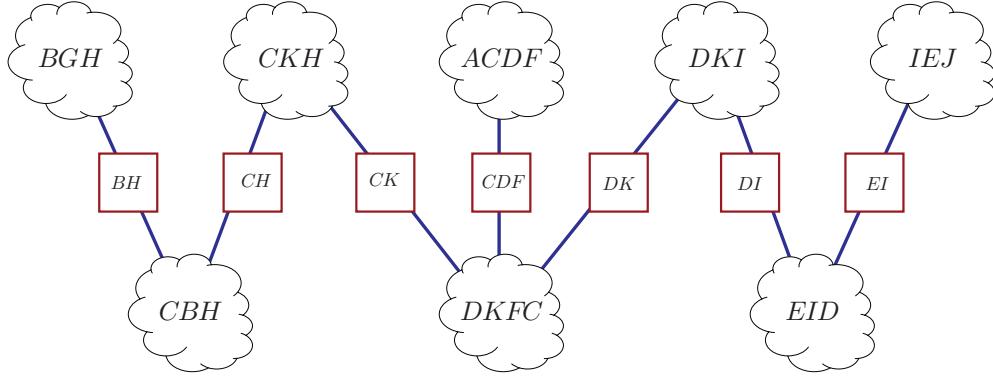


Figure 6.31: The junction tree shown here is a junction tree of maxcliques corresponding to the decomposable graph given in Figure 6.23

In fact, the junction tree of cliques for Figure 6.23 is shown in Figure 6.31. Once should verify that the clique intersection property, the r.i.p., and the induced sub-tree property all hold.

Therefore, once we have a triangulated graph, we can form a junction tree. Now why does this help us? Notice that the theorem is a bijection. So while any graph is such that we can cluster the nodes and form a tree from those clusters, only a triangulated graph is one where we can form a junction tree that satisfies r.i.p. This will allow us to treat any graph as a tree, by first triangulating that graph, and then forming a junction tree. But more importantly, the resulting tree will be one where we can perform inference that is essentially identical to what we did in Section 6.4 and have it be correct.

Intuitively, since a triangulated graph is such that its each variable induces a connected tree in the graph's junction tree, each variable can be seen as induces is a tree-shaped "spine" in the tree, so that as long as information is passed around in the tree in the appropriate order (e.g., the message passing order that we introduced above), information regarding that variable (such as marginal distributions over that variable) will not get become inconsistent. Moreover, it is the case that consistency can be ensured over the entire tree just by ensuring consistency locally (as we will see). If the induced sub-tree property was not true, then a variable might become inconsistent after message passing. This is somewhat hand-wavy at the moment, and we will be more precise below, but before that, lets first talk about ways to construct a junction tree from a triangulated graph. The reason, as we will see, is that a junction tree is not just an interesting graph-theoretic alternative representation of a triangulated graph, but it also yields a useful data-structure that can be used by a computer program to construct a system that can do general exact inference on graphical models.

Recall that not all trees of maxcliques are JTs. For example, Figure 6.28 shows two trees of maxcliques, one of which is not a junction tree since it does not satisfy r.i.p. order.

In general, there are two ways to form a junction tree from a triangulated graph. The first way is to use the MCS algorithm. In fact, we have the following theorem.

Theorem 108. If the input graph to MCS is triangulated, then it produces a list of maxcliques in a r.i.p.-respecting order.

Once we have the cliques in r.i.p. order, we can form a junction tree by using the tree-generative procedure mentioned above where we connect an edge between C_i and the corresponding C_j with the appropriate $j < i$.

On the other hand, we may wish to run MCS since we know we can use the elimination algorithm in modified form which can be used to provide us with the set of maxcliques of the reconstituted and triangulated graph. Given the set of maxcliques of a triangulated graph, we can form a junction tree by first constructing a junction graph of maxcliques, where nodes in the graph correspond to cliques, where edges exist between two nodes only if the corresponding cliques have a nonempty intersection, and edges are weighted by the size of this intersection (meaning all weights are integers greater than 0). We have:

Theorem 109. *A tree of maxcliques \mathcal{T} is a junction tree iff it is a maximum spanning tree on the max clique graph, with edge weights set according to the cardinality of the separator between the two maxcliques.*

Proof. Consider a forest (a set of trees) \mathcal{T} consisting of M nodes C_i and $M - 1$ separators between cliques S_j , and consider some element $v \in V$. The number of times v appears in the set of separators is $\sum_{i=1}^{M-1} \mathbf{1}\{v \in S_i\}$. The number of times v appears in the maxcliques is $\sum_{i=1}^M \mathbf{1}\{v \in C_i\}$. In a forest, moreover, we have that

$$\sum_{i=1}^{M-1} \mathbf{1}\{v \in S_i\} \leq \sum_{i=1}^M \mathbf{1}\{v \in C_i\} - 1 \quad (6.68)$$

since whenever $v \in S_i$ then v is also in the cliques neighboring S_i . The inequality becomes an equality only when the subgraph of \mathcal{T} induced by v is a tree (which is the induced sub-tree property of a junction tree). Considering the total weight of the tree, meaning the sum of the cardinalities of the edges weighted by separator cardinality, we have:

$$\begin{aligned} w(\mathcal{T}) &= \sum_{j=1}^{M-1} |S_j| = \sum_{j=1}^{M-1} \sum_{k=1}^N \mathbf{1}\{x_k \in S_j\} \\ &= \sum_{k=1}^N \sum_{j=1}^{M-1} \mathbf{1}\{x_k \in S_j\} \leq \sum_{k=1}^N \left[\sum_{j=1}^M \mathbf{1}\{x_k \in C_i\} - 1 \right] \\ &= \sum_{i=1}^M \sum_{k=1}^N \mathbf{1}\{x_k \in C_i\} - N = \sum_{i=1}^M |C_i| - N \end{aligned}$$

We can see that $w(\mathcal{T})$ is maximized when the one inequality becomes an equality, and that happens when all nodes induce a tree in the graph of cliques (again the induced sub-tree property). Therefore, finding the maximum spanning tree in the graph of cliques forms a junction tree. \square

Note the graph must be triangulated. I.e., maximum spanning tree of a cluster graph where the clusters are maxcliques but the graph is not triangulated will clearly not produce a junction tree.

This is a very standard way to get a junction tree in real inference code since the elimination algorithm can easily produce the set of maxcliques, and there are a number of simple but fast greedy strategies to form a maximal spanning tree, including Kruskal's or Prim's MST algorithm (see Cormen et al.). Prim's algorithm in fact can be made to run in $O(|E| + |V| \log |V|)$ using a Fibonacci heap.

It is interesting to note that a junction tree is created, either explicitly or implicitly, whenever we wish to do exact probabilistic inference optimally. This is even true for the search based methods where perhaps value-specific junction trees might be formed. We will see this when we discuss time-space trade offs in Section ??).

It is also important to realize that, given a junction tree, it is possible to recover the original triangulated graph. To do this, we start with a set of nodes $V = \cup_{C \in \mathcal{C}} C$. Then we add an edge between $u, v \in V$ if there exists a $C \in \mathcal{C}$ such that $u, v \in C$. The maxcliques of the junction tree, therefore, form a type of edge

cover for the graph (i.e., the union of the cliques in the junction tree should correspond to all edges in the triangulated graph).

Theorem 110. *The above reconstruction from a junction tree back to a graph yields the same triangulated graph that we started from.*

Proof. The theorem is really stating that the set of maxcliques of a graph constitute an edge clique cover of that graph. In fact, whenever we have a set of all cliques in a graph, that set constitutes an edge clique cover. And since all cliques are contained in a maxclique, if we have a set of all maxcliques, then all edges and only the edges in G are covered by some clique. \square

A junction tree of maxcliques is of course unique in that any junction tree of all maxcliques will always contain the same maxcliques . However, there can be multiple junction trees of maxcliques since there might be several ways of connecting together the maxcliques into a tree such that r.i.p. still holds. This can be seen by considering that there could be multiple equivalent weight maximum spanning trees of the graph of cliques, and since they each have maximum weight, they each constitute a junction tree.

Junction trees also are related to the binary decomposition trees we saw above. It should be clear that the binary decomposition trees are not junction trees as they need not satisfy the junction tree properties — once we find a non-leaf node separator to produce a decomposition, the next set of separators might have no intersection with the parent separator (see the examples given in Figure 6.23). On the other hand, the set of leaf nodes of the decomposition trees correspond to the nodes of the junction tree. And the set of separators in the binary decomposition tree are the same set of separators in the junction tree. Moreover, as long as each separator in the junction tree corresponds to exactly two maxcliques (see below where this might not be the case), then the set of edges in the junction tree are in one-to-one correspondence with the non-leaf nodes in the decomposition tree. This set of separators, moreover, corresponds precisely to all minimal separators in the corresponding triangulated graph. It should be clear, therefore, that going from a junction tree to a binary decomposition tree is as simple as selecting edges from a junction tree to partition the tree, and repeating this process recursively.

It should be noted that there are various possible forms of S . That is, regarding the separators and their identities as a subset of vertices, we could have: 1) the separators might be used only once in the junction tree, and all separators are unique; 2) the separators in the junction tree might be uniquely used in that each separator is the intersection of and connected to exactly two cliques, but there might be some sets of separators that are identical in their vertex identities; and 3) the separators in the junction tree might be used multiple times in that they shatter the junction tree into more than 2 connected subtrees. In either of the last two cases, we have that $d(S) > 2$. In the third case, however, we have a junction tree where a separator connects more than two maxcliques. Such a junction tree is shown in Figure ??.

There is one remaining interesting property of triangulated graphs and junction trees that is worth mentioning and that helps to gain intuition regarding these structures, and that is how junction trees and triangulated graphs correspond to the notion of an intersection graph.

We first review a bit of terminology. A “set cover” is a set of sets $\{S_i\}_i$ that cover some ground (or universal) set S , so a set cover could be called a ground set cover. A “vertex cover” is a set of vertices must cover all the edges in some graph, so it could be called an edge vertex cover. An “edge cover” is a set of edges must cover the vertices of a graph, so it could be called a “vertex edge cover.” Therefore, a “clique cover” is considered a set of cliques that must cover the edges, and is called an “edge clique cover.” The theorem above stated that the maxcliques of a junction tree for a triangulated graph G' constitute an edge clique cover for the G' - we start with set of nodes $V = \cup_{C \in \mathcal{C}} C$. Add edge between $u, v \in V$ if exists a $C \in \mathcal{C}$ such $u, v \in C$. Going from G' to a junction tree and back to the graph yields the same graph.

Intersection graphs and the notion of edge covers are closely related, as we next see.

Definition 111 (Intersection Graph). An intersection graph is a graph $G = (V, E)$ where each vertex $v \in V(G)$ corresponds to a set U_v and each edge $(u, v) \in E(G)$ exists only if $U_u \cap U_v \neq \emptyset$.

Another way of saying this is that there is some underlying set of objects U and a multiset of subsets of U of the form $\mathcal{U} = \{U_1, U_2, \dots, U_n\}$ with $U_i \subseteq U$. We note that this is a multiset rather than set since we might have some i, j where $U_i = U_j$. From this multiset of sets, we can form a graph $G = (V, E)$ with n vertices, where there is a one-to-one correspondence between nodes in G and the integers $\{1 \dots n\}$, and where an edge exists between vertex u and v if $U_u \cap U_v \neq \emptyset$. If a graph can be represented in this way, then we say that it is an intersection graph. It turns out, in fact, that any graph has such a representation:

Theorem 112. Every graph is an intersection graph.

Informally, we can see that this must be true since for any finite graph $G = (V, E)$ (i.e., with a finite number of nodes), we can form the set U as the set of all nodes (so $U = V$). Then for any node $v \in V$, we form the corresponding subsets U_v so that both $v \in U_v$ and for each $u \in \delta(v)$, we also have that $u \in U_v$. Then, any edge $u, v \in E(G)$ will lead to a non-empty intersection between the corresponding two subsets since $u \in U_v$ and $v \in U_u$.

Of course, where this gets a bit more interesting is when the set U and the subsets have some structure. To motivate this by an example, consider the subsets that constitute the multiset as being generated based on closed line segments, where each node $v \in V(G)$ corresponds to one line segment L_v , and where an edge u, v exists only if the corresponding line segments overlap at least at a point, as shown in the following figure



The left of the figure shows a set of intersecting closed line segments $\{a, b, c, d, e\}$ each of which corresponds to a node in a graph. The right figure shows the corresponding graph, where we see, for example, that $a, b \in E(G)$ since line segments a and b overlap on the horizontal axis. Graphs that can be formed in this way are called *interval graphs*.

We can make the structures a little bit more sophisticated still. For example, let's say that we are given some underlying tree $T = (U, E_T)$, and also a set of subtrees of this tree $\{T_1, T_2, \dots, T_n\}$. Note that each T_i is a sub-tree of T which means it is connected. We can define the intersection between two subtrees T_i and T_j to be non-empty if any vertices of T_i and T_j are common. Given this notion of tree intersection, we can form a graph G of n nodes, each node corresponding to one of the subtrees, and where an edge exists in G between two nodes if the corresponding subtrees have a non-empty intersection. Any graph that can be formed in this way is called a *sub-tree graph*. Note that sub-tree graphs indicate an interesting property of intersection graphs: in general, the constraints in an intersection graph need not be how intersection is done (in sub-tree graphs, we are still using intersection of sets to determine edges) but instead how the subsets U_1, U_2, \dots, U_n of U are formed. In this case, the subsets must be subtrees of some underlying tree.

Figure 6.32 shows an example. On the left we have a large tree (black bold) and a set of sub-trees of that tree, each of which is labeled with a sub-tree index T_i . The middle figure corresponds to the sub-tree graph corresponding to these sub-trees. The right figure shows another tree and set of sub-trees which generates the same sub-tree graph in the middle. In general, there are many sub-tree representations of sub-tree graphs. We're interested, however, in how these graphs relate to triangulated graphs. When we consider the induced sub-tree property of a junction tree corresponding to a triangulated graph, it should be clear that sub-tree graphs seem to be related. In fact the relationship is quite strong, as indicated by the following theorem.

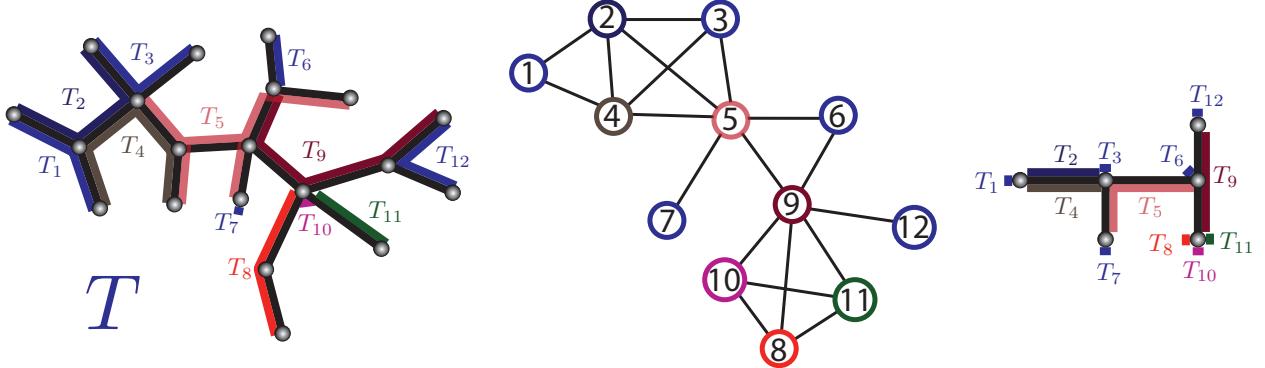


Figure 6.32: An example of tree $T = (U, E_T)$ (left), a set of subtrees of that tree $\{T_i : i = 1 \dots 12\}$, and the corresponding graph $G = (V, E)$ (middle) where for each $v \in V = \{1 \dots 12\}$ we have a subtree T_v of T , and where $u, v \in E(G)$ iff $T_u \cap T_v \neq \emptyset$, where intersection in this case correspond to tree intersection. Note that this tree is not unique, as another tree (right) generates the same graph. These pictures also show that the subtrees of a tree might even be a single node (e.g., T_7 and T_{10} on the left), and an intersection exists between two subtrees if they have any node in common.

Theorem 113. *A graph $G = (V, E)$ is triangulated iff it corresponds to a sub-tree graph (i.e., an intersection graph on subtrees of some tree).*

Proof. triangulated \Rightarrow sub-tree graph: Given a triangulated graph G , form a junction tree \mathcal{T} of maxcliques of G . Consider, further, two nodes $u, v \in V(G)$ and the corresponding two induced subtrees of \mathcal{T} , namely \mathcal{T}_u and \mathcal{T}_v . An edge $u, v \in G$ exists iff there exists a maxclique in the nodes of \mathcal{T} containing both u and v . If subtrees \mathcal{T}_u and \mathcal{T}_v have an intersection, then there must be at least one clique containing both u and v yielding this edge.

sub-tree graph \Rightarrow triangulated: Let $G = (V, E)$ be a sub-tree graph with underlying tree T and subtrees T_v for each $v \in V(G)$. Suppose G were not triangulated. Then there exists an unchorded cycle $v_1, v_2, \dots, v_k, v_1$ where $k \geq 4$. Therefore, $T_{v_i} \cap T_{v_{i+1}} \neq \emptyset$ for all $1 \leq i \leq k$ (where we take $k+1$ to wrap around back to 1) since the intersections correspond to edges in the cycle. We also have that $T_{v_i} \cap T_{v_j} = \emptyset$ for anything other than the successive nodes in the cycle (since it is a chordless cycle).

Therefore, in T , there is some path from some tree node $t_a \in T_1$ to some other tree node $t_b \in T_k$ going through nodes of T via the subtrees T_1, T_2, \dots, T_k in that order. Let $t_a, t_1, \dots, t_\ell, t_b$ be this path in T . Thus, this path must intersect each subtree T_i for $1 < i < k$. But we also have that $T_1 \cap T_k \neq \emptyset$ and moreover we must have that $t_i \notin T_1 \cap T_k$ for $1 < i < \ell$ since otherwise there would be a chord. Therefore, $T_1 \cap T_k \neq \emptyset$ can not be due to these two subtrees extending all the way around the path $t_a, t_1, \dots, t_\ell, t_b$ so there must be some other common path between nodes in the two subtrees, but this leads to a contradiction since T is a tree. Therefore, G must be chordal. \square

It is interesting to note that the junction tree itself can act as the tree T corresponding to triangulated graph G . Any node v in G carves out a sub-tree of the junction tree, and two nodes in G are connected by an edge iff the two corresponding sub-trees in the junction tree intersect. In fact, in Figure 6.32, we see a tree T corresponding to the graph that is exactly a junction tree of maxcliques in the graph. Each node of the tree corresponds to a maxclique of nodes corresponding to all trees that intersect that node — the junction tree for this graph is given below:

What the above sub-tree graph description of triangulated graphs tell us, however, is that there are many trees that can “generate” a given triangulated graph besides the junction tree. Each such tree has the property

that the node of a tree corresponds to a clique in the graph, but not necessarily a max clique. For example, in Figure 6.32-left, the nodes of the tree T yields the cliques, moving roughly from left to right, $\{1, 2\}$, $\{1, 4\}$, $\{1, 2, 4\}$, $\{2, 3\}$, $\{3\}$, $\{2, 3, 4, 5\}$, $\{4, 5\}$ and so on. The junction tree, of course, is the only one that consists of the maxcliques, although as we have seen, even there there could be different trees over these maxcliques.

So, now that we have a junction tree, we have a representation of the probability distribution $p(x)$ in terms of marginals on cliques and separators. That is, we have that if G is a triangulated graph, then for all $p \in \mathcal{F}(G, \mathcal{M}^{(f)})$, and corresponding junction tree $\mathcal{T} = (\mathcal{C}, \mathcal{S})$ where \mathcal{C} is the set of maxcliques in G and \mathcal{S} is the set of separators in the corresponding junction tree, we can write the factorization as in Equation 6.63.

Given the above factorization of $p(x)$, it is possible for a junction tree's nodes and separators to represent the joint distribution by assigning the marginal $p(x_C)$ to the junction tree node $C \in \mathcal{C}$ and assigning the marginal $p(x_S)$ to $S \in \mathcal{S}$. That is, we can think of each node and each separator in the junction tree as having an associated potential function, and by making these marginal assignments to the potential function, we can represent the joint distribution $p(x)$.

On the other hand, it is typically the case that we do not initially have these marginals and in fact the goal of inference is to compute these marginals for each maxclique (for the purposes of learning as described early in this chapter TODO: where). That is, suppose we started from an undirected graph G that is not necessarily triangulated. Given a $p \in \mathcal{F}(G, \mathcal{M}^{(f)})$, we have a set of potentials associated with p . Once we triangulate G into G^t we know that the nodes of the corresponding junction tree are such that any clique in G is a subset of a maxclique in G^t . Moreover, any factor ϕ of $p \in \mathcal{F}(G, \mathcal{M}^{(f)})$ over a set of variables has a maxclique C in G^t that contains all its variables. To be very precise about this point, $p \in \mathcal{F}(G, \mathcal{M}^{(f)})$ can be represented as:

$$p(x) = \prod_{C \in \mathcal{C}(G)} \psi_C(x_C) \quad (6.69)$$

where in this case $\mathcal{C}(G)$ correspond to the cliques (or maxcliques if we have them, it does not matter, see chapter on UGMs, Section XXX) of G . Our goal is to represent this factorization as an initial assignment to the maxclique and separators of the junction tree. That is, let \mathcal{T} be the junction tree corresponding to G^t and let $\mathcal{C}'(\mathcal{T})$ be the maxcliques of G^t (equivalently the nodes of the junction tree), and let \mathcal{S} be the set of separators in the junction tree.

Our problem is to assign the potentials in the junction tree so that the junction tree factorization equals Equation 6.69. That is, for each $C' \in \mathcal{C}'(\mathcal{T}) = \mathcal{C}'(G^t)$, let $\psi_{C'}(x_{C'})$ be the potential associated with the maxclique C' , and for each separator $S \in \mathcal{S}(\mathcal{T})$ (which corresponds to complete minimal separators in G^t), let $\phi_S(x_S)$ be the potential function associated with that separator. That is, we wish to initiate $\{\psi_{C'}(x_{C'}) : C' \in \mathcal{C}'\}$ and $\{\phi_S(x_S) : S \in \mathcal{S}\}$ so that:

$$p(x) = \prod_{C \in \mathcal{C}(G)} \psi_C(x_C) = \frac{\prod_{C \in \mathcal{C}'} \psi_{C'}(x_{C'})}{\prod_{S \in \mathcal{S}(G)} \phi_S(x_S)} \quad (6.70)$$

For each maxclique C of G , we can find a maxclique $C' \in V(\mathcal{T})$ of the corresponding junction tree of a triangulation of G such that $C \subseteq C'$. Therefore, this potential function $\phi_C(x_C)$ can be assigned to the potential function at junction tree node C' . Note that there might be more than one C' such that $C \subseteq C'$ but we only need assign it to one (in fact it is incorrect to assign it to more than one since then that potential would be multiplied in twice in the numerator in Equation 6.63). This therefore gives us a way of initializing the potential functions associated with the nodes of the junction tree: for each $C \in \mathcal{C}(G)$ find a $C' \in \mathcal{C}(\mathcal{T})$ such that $C \subseteq C'$ and multiply $\phi_C(x_C)$ into the existing potential function at that node. That is, given a set of subsets A_1, A_2, \dots, A_n of C and corresponding potential functions $\phi_{A_i}(x_{A_i})$ which are factors of p , we can form a potential function over just C as

$$\phi_C(x_C) = \prod_i \phi_{A_i}(x_{A_i}) \quad (6.71)$$

Note that it might be that $A = \cup_i A_i = C$, meaning that all variables in the resulting potential function $\phi_C(x_C)$ have a corresponding factor. On the other hand, it might also be the case that $A = \cup_i A_i \subset C$. This can happen, for example, if some factor whose variables are fully contained within C are assigned to some other clique's potential function. In this case, for any $j \in C \setminus A$, changing the value of x_v in $\phi_C(x_C)$ has no effect on $\phi_C(x_C)$'s value (i.e., $\phi_C(x_{C \setminus \{j\}}, x_j) = \phi_C(x_{C \setminus \{j\}}, x'_j)$ for any $x_j \neq x'_j$). Even so, the assignment is still mathematically valid.

If, moreover, there is some maxclique in the junction tree, $C' \in \mathcal{C}(\mathcal{T})$, that has a $v \in C'$ such that no potential has been assigned that involves v in that maxclique, then there will be some other potential in $\mathcal{C}(G)$ that involves v and that factor it will be assigned to some other junction tree node.

For the separators of the junction tree, moreover, we just initialize them to one. We make this a little bit more precise with the following algorithm.

Algorithm 13: Algorithm to initialize the potentials in a junction tree starting from an UGM.

Input: An undirected graph $G = (V, E)$ and corresponding $p(x) \in \mathcal{F}(G, \mathcal{M}^{(f)})$ with potential functions $p_C(x_C)$ for each clique in G ($C \in \mathcal{C}(G)$). A triangulation $G^t = (V, E \cup F)$ of G and a corresponding junction tree $\mathcal{T} = (\mathcal{C}', \mathcal{S})$ for G^t with nodes of \mathcal{T} corresponding maxcliques of G^t , edges of \mathcal{T} corresponding to complete minimal separators of G^t , and corresponding potential functions for each node and each edge (separator) of \mathcal{T} .

Result: An assignment to both the nodes and edges of \mathcal{T} such that $p(x)$ is exactly represented using Equation 6.63.

```

1 foreach  $C' \in \mathcal{C}'(G^t)$  do
2    $\psi_{C'}(x_{C'}) \leftarrow 1$ ; /* Initialize the potential to unity for all values of the variables  $x_{C'}$ . */
3 foreach  $S \in \mathcal{S}(G^t)$  do
4    $\phi_S(x_S) \leftarrow 1$ ; /* Initialize the separator potential to unity. */
5 foreach  $C \in \mathcal{C}(G)$  do
6   Find a  $C' \in \mathcal{C}'$  such that  $C \subseteq C'$ ;
7    $\psi_{C'}(x_{C'}) \leftarrow \psi_{C'}(x_{C'})\psi_C(x_C)$ ; /* Update the maxclique potential with the clique potential. */
8 return The finished junction tree potentials

```

Note that in Line 5 of Algorithm 13, we are finding any maxclique C' of G^t that covers the clique C of G — any such maxclique will work (since multiplication is commutative) as long as we do not do this twice.

As we saw before, if we are interested in marginals over cliques in the original graph, one run of elimination is sufficient to produce a triangulated graph that is optimal from the perspective of doing all runs of elimination. Therefore, we do not need to spend time triangulating the graph multiple times and can operate on the junction tree representation of the graph for all queries. We basically therefore need to go from a configuration of the junction tree potentials as given in in Equation 6.70 and initialized by Algorithm 13, and end up in a configuration where the junction tree potentials are marginals, as in Equation 6.63. That is, we want $\psi_{C'}(x_{C'}) = p(x_{C'})$ for each $C' \in \mathcal{C}'$ and $\psi_S(x_S) = p(x_S)$ for each $S \in \mathcal{S}$.

First, why might is the junction tree not in this final configuration from the start? There are several reasons why they don't start out as such.

1. After introduction of evidence, some clique potentials are unchanged, namely those with $C' \cap E = \emptyset$, and will not be able to produce $p(x_{C'} | \bar{x}_E)$ but this is really what we wish to obtain.
2. Not all the evidence \bar{x}_E is present at (probably most or all all) of the clique potentials. That is, $E \not\subseteq C'$ for any C' .

3. The junction tree's clique potentials might be initialized by conditional probabilities from directed graph, so we might have, for example, the initial configuration being something like $\psi_{DEF} = p(F|D, E)$, which is not a marginal over DEF .
4. The junction tree clique potentials might be specified arbitrarily at first as a collection of unnormalized factors multiplied together. Consider, for example, MRFs using log-linear models, where none of the factors are normalized, and where the global normalization constant $1/Z$ is either not present or is contained in one of the factors. In such a case, none of the potentials are marginals. In either case, clique potentials wouldn't be marginals.

Recall again that if we started from a Bayesian network, we first moralized and then triangulated the graph, and initialized the potential functions to each conditional probability table as appropriate. If any evidence exists, we utilize delta functions as explained in the chapter on evidence, so we can think of the evidence as being part of the existing clique potential functions of C .

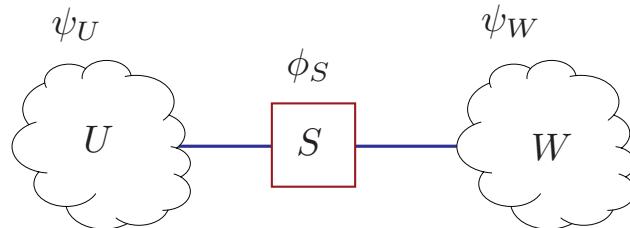
Now, if our final configuration is such that the junction tree potentials equal the marginals, these potentials must agree at least on the nodes that they have in common with each other. To agree means to be consistent with, which means that they give the same sub-marginals over the variables that are in the intersection of the corresponding two maxcliques. That is, given maxclique C'_1 and C'_2 of C , with $S = C'_1 \cap C'_2$, we must have that

$$\sum_{x_{C'_1 \setminus S}} \psi_{C'_1}(x_{C'_1}) = \sum_{x_{C'_2 \setminus S}} \psi_{C'_2}(x_{C'_2}) \quad (6.72)$$

It should be clear that this, so far, is only a necessary (but not yet a sufficient) condition for the junction tree potentials to equal the marginals. In other words, if junction tree potentials were already equal the marginals, we would certainly have this consistency as

$$\sum_{x_{C'_1 \setminus S}} \psi_{C'_1}(x_{C'_1}) = \sum_{x_{C'_1 \setminus S}} p(x_{C'_1}) = \sum_{x_{C'_2 \setminus S}} p(x_{C'_2}) = \sum_{x_{C'_2 \setminus S}} \psi_{C'_2}(x_{C'_2}) \quad (6.73)$$

Lets now assume that we have two maxcliques V and W with separator $S = V \cap W$, and potential functions ψ_V , ψ_W , and ϕ_S , arranged in a junction tree as in the following figure:



As is common, we use a shorthand notation $\phi_S^* = \sum_{V \setminus S} \psi_V$ to represent a new potential over the separator S obtained from ψ_V where all but S has been marginalized away. We make sure this is clear, we have that

$$\psi_V = \psi_V(x_V) \quad \text{represents a clique potential on } x_V$$

and

$$\phi_S = \phi_S(x_S) \quad \text{represents a separator potential on } x_S$$

Also, we have that $V = S \cup (V \setminus S)$, so V contains a part containing S and an “innovation” $V \setminus S$ since $S \subset V$. Thus,

$$\sum_{V \setminus S} \psi_V = \sum_{x_{V \setminus S}} \psi_V(x_V) = \sum_{x_{V \setminus S}} \psi_V(x_{V \setminus S}, x_S) = \phi_S^*(x_S)$$

which is a function only of x_S .

We will also utilize *table multiplication* notation. That is, we will see things like:

$$\psi_W^* = \frac{\phi_S^*}{\phi_S} \psi_W \quad (6.74)$$

What does this mean? Again, we expand out: Let $W_S = W \setminus S$, so that $W = S \cup W_S$. then

$$\psi_W = \psi_W(x_W) = \psi_W(x_S, x_{W_S}) , \phi_S = \phi_S(x_S) \quad (6.75)$$

and

$$\psi_W^* = \psi_W^*(x_W) = \psi_W^*(x_S, x_{W_S}) , \phi_S^* = \phi_S^*(x_S) \quad (6.76)$$

so to expand everything out, we get

$$\psi_W^* = \psi_W^*(x_S, x_{W_S}) = \frac{\phi_S^*(x_S)}{\phi_S(x_S)} \psi_W(x_S, x_{W_S}) \quad (6.77)$$

Suppose, now, that the junction tree potentials start out as being inconsistent. That is, we have

$$\sum_{V \setminus S} \psi_V \neq \sum_{W \setminus S} \psi_W \quad \text{and} \quad \phi_S = 1 \quad (6.78)$$

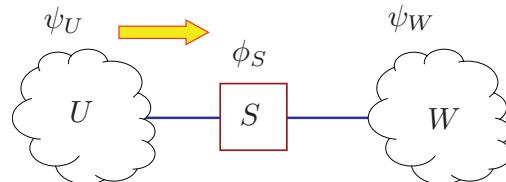
but we still have that $p(x_H, \bar{x}_E) = \psi_V \psi_W / \phi_S$.

The key operation that we will perform is message passing between cliques via the separators. Multiple messages means exchange of info between clique potentials, which should yield “consistency”. We have a set of operations:

- **Marginalize V :**

$$\phi_S^* = \sum_{V \setminus S} \psi_V \quad (6.79)$$

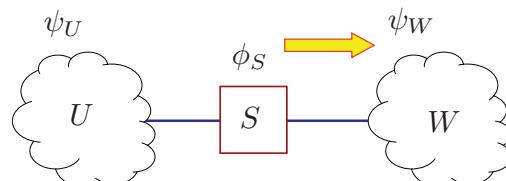
which leads to a new separator potential ϕ_S^* and can be seen as a partial message, as shown in the following figure



- **Rescale W :**

$$\psi_W^* = \frac{\phi_S^*}{\phi_S} \psi_W \quad (6.80)$$

This produces a new potential on W based on the updated separator potential at S . This can also be seen as a partial message.



After these operations, the new joint $p(x_H, \bar{x}_E)$ has not changed. If we define $\psi_V^* = \psi_V$ for convenience, we get:

$$\frac{\psi_V^* \psi_W^*}{\phi_S^*} = \frac{\psi_V \psi_W \phi_S^*}{\phi_S \phi_S^*} = \frac{\psi_V \psi_W}{\phi_S} \quad (6.81)$$

While the joint distribution over these potentials has not changed, we have not yet achieved consistency since

$$\sum_{V \setminus S} \psi_V^* = \sum_{V \setminus S} \psi_V = \phi_S^* \neq \sum_{W \setminus S} \psi_W^* = \frac{\phi_S^*}{\phi_S} \sum_{W \setminus S} \psi_W \quad (6.82)$$

which follows because

$$\phi_S \neq \sum_{W \setminus S} \psi_W \quad (6.83)$$

But we do at least have one marginal at ψ_W^* . This is because we started with:

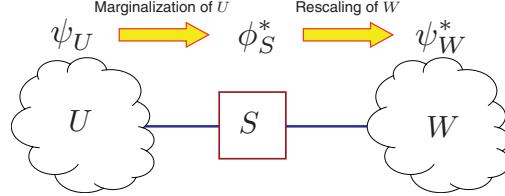
$$p(x_H, \bar{x}_E) = \frac{\psi_V \psi_W}{\phi_S} \quad (6.84)$$

and

$$\psi_W^* = \frac{\phi_S^*}{\phi_S} \psi_W = \psi_W \sum_{V \setminus S} \psi_V = \sum_{x_{V \setminus S}} p(x_H, \bar{x}_E) = p(x_{H \cap W}, \bar{x}_E) \quad (6.85)$$

is one of the marginals that we desire.

In general, we can see this as a message passing procedure, passing a message from maxclique V through S and to W as shown next:



What if we were to do the same set of operations in reverse, i.e., send a message from W back to V using the new state of the potential functions. I.e., we first

- **Marginalize W :**

$$\phi_S^{**} = \sum_{W \setminus S} \psi_W^* \quad (6.86)$$

resulting in still another separator potential, and

- **Rescale V :**

$$\psi_V^{**} = \frac{\phi_S^{**}}{\phi_S^*} \psi_V^* \quad (6.87)$$

resulting in a new potential on V .

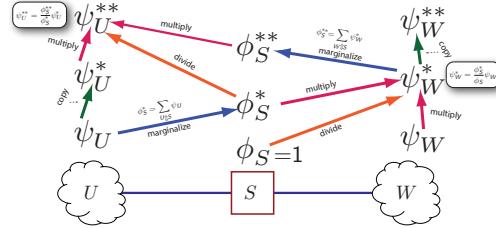
After these operations, the new joint $p(x_H, \bar{x}_E)$ has again not changed. If we define $\psi_W^{**} = \psi_W^*$ for convenience, we get:

$$\frac{\psi_V^{**} \psi_W^{**}}{\phi_S^{**}} = \frac{\psi_V \phi_S^{**} \psi_W \phi_S^*}{\phi_S^{**} \phi_S^* \phi_S \phi_S^*} = \frac{\psi_V \psi_W}{\phi_S} \quad (6.88)$$

Perhaps more importantly, after this second “backwards” message pass, consistency is now achieved. In particular, ψ_V^{**} and ψ_W^{**} are now consistent since:

$$\sum_{V \setminus S} \psi_V^{**} = \sum_{V \setminus S} \frac{\phi_S^{**}}{\phi_S^*} \psi_V^* = \frac{\phi_S^{**}}{\phi_S^*} \sum_{V \setminus S} \psi_V^* = \frac{\phi_S^{**}}{\phi_S^*} \phi_S^* = \phi_S^{**} = \sum_{W \setminus S} \psi_W^{**} \quad (6.89)$$

We have therefore performed a forward and then a backwards message passing scheme on the clique in the graph as described in the following figure, where the arrows show the dependency graph based on the definitions of the messages:



We moreover have the other marginal we want at ψ_V^{**} since:

$$\begin{aligned}\psi_V^{**} &= \frac{\phi_S^{**}}{\phi_S^*} \psi_V = \psi_V \frac{\sum_{W \setminus S} \psi_W^*}{\sum_{V \setminus S} \psi_V} = \psi_V \frac{\sum_{W \setminus S} \frac{\phi_S^*}{\phi_S} \psi_W}{\sum_{V \setminus S} \psi_V} \\ &= \psi_V \frac{\sum_{W \setminus S} \psi_W \sum_{V \setminus S} \psi_V}{\sum_{V \setminus S} \psi_V} = \psi_V \sum_{W \setminus S} \psi_W = \sum_{W \setminus S} p(x_H, \bar{x}_E) \\ &= p(x_{H \cap V}, \bar{x}_E)\end{aligned}$$

So, now we have a way both to get the marginals we want and thus achieve local consistency, at least when we have only two cliques in the junction tree.

We wish next to address the question of how can we ensure that this is still correct when we have multiple overlapping cliques, i.e., a tree of cliques? In other words, how do we make sure that any local consistency we have achieved is not ruined by later message passing steps that involve the maxcliques that are locally consistent? For example, once we send message $V \rightarrow W$ and then $W \rightarrow V$, we know W and V are consistent. If we next send messages $W \rightarrow D_1$ and $D_1 \rightarrow W$, then W & D_1 are consistent, but V & W are no longer necessarily consistent.

Lets hypothesize that the same message passing scheme that we discovered during simple inference on 1-trees will work here in the junction tree as well. We repeat it here in the context of junction trees mainly for convenience:

Definition 114 (Message passing protocol). *A clique can send a message to a neighboring cluster in a JT only after it has received messages from all of its other neighbors.*

Note that we have defined this here in terms of clusters.

The proof that MPP works here as well is also similar, but we give the complete proof in this context:

Theorem: The message passing protocol renders the clusters in a cluster tree locally consistent between all pairs of connected cliques in the tree.

Note that we have defined this in terms of clusters.

Suppose W has received a message from all other neighbors, and is sending a message to V . There are two possible cases:

Case A: V already sent a message to W before, so V already received message from all other neighbors, & message renders the two consistent since neither receives any more messages.

Case B: V has not yet sent a message to W , so W sends to V & waits. Later, V will have received message from all other neighbors & will send message back to W , but this will contain appropriate update from W .

We have not used r.i.p. in this theorem.

This part requires r.i.p.

Junction tree satisfies r.i.p. so the above theorem also holds for JTs.

6.10 From JT back to normal tree

Another idea: start with the regular tree inference stuff, but add some variables that are attached only to the edges. I.e., if $(i, j) \in E(T)$, and $\psi_{i,j}(x_i, x_j)$ is the potential, we might have some other variable x_k where $k \notin V(T)$ and where we have another factor $\psi_{i,j \rightarrow k}(x_i, x_j, x_k)$ where x_k is a (possibly deterministic) function of x_i and x_j . We can keep adding bundles of variables in this way, all variables of which are directly or indirectly functions of the original edge variables x_i and x_j .

The idea is that the 1-tree case, the nodes of the 1-tree correspond to the separators in the junction tree, and the cliques in the junction tree correspond to the edges. But in a JT, there might be nodes in a given cluster C_j that do not live in any of the neighboring separators. I.e., we might have that $C_{j, \neg S} \triangleq C_j \setminus \bigcup_{S \in \mathcal{S}(C)} S = \emptyset$ where $\mathcal{S}(C)$ is the set of separators adjacent to C in the cluster tree. So in this case, these variables are similar to the extra bundle of variables in the tree defined above.

Key trick: turn any junction tree into one that has the following property. Each clique has variables corresponding to the union of all of its neighboring separators. If there is any clique that has more than the union of its neighboring separators, then hang a new clique off of it with a separator, then prune off that extra stuff, hand a new clique and separator off of that clique, and then we'll satisfy property.

I.e., lets say that $C \setminus \{S_1, S_2, S_3\} \neq \emptyset$. Then we form $C' = S_1 \cup S_2 \cup S_3$ to replace C , and then form another new clique C'' that contains it and then ...

Given that we can embed any graph into a junction tree with sufficiently large clique, perhaps it is not surprising that we can then re-embed the JT back into a normal tree, but with variables having domain sizes commensurate with the states spaces of each of the cliques in the JT, and with additional factors.

Cliques in a junction tree correspond to edges in a tree, and separators in a junction tree correspond to nodes in a tree. In a standard tree, when we consider the “degree” of a node, it is often the case that the degree of a node is more than 2. In a junction tree, a we have set up, the separator is such that it corresponds to two cliques. Note that a given separator however might be used multiple times (consider a decomposition tree where $d(S) > 2$). In such case, we use $d(S) - 1$ identically constructed separators to form the junction tree, but this need not be the case. A JT would be well formed if we only used one separator in a JT, and that separator is connected to $d(S)$ cliques, however large $d(S)$ is. In such case, this is more like what a standard tree is like. On the other hand, we always modify a regular tree to look more like a standard junction tree, and this was done in Figure 6.15 — the separator x_3 has a separate data structure for each pair of cliques (edges) that it is involved in.

So, let us now construct such a junction tree, where there is only 1 unique separator used rather than $d(S)$ but where the degree of that separator is given by $d(S)$. this is shown in Figure XXX.

Our goal is to reconstruct a regular tree.

Start with simple example, how we can take two random variables, say x_1, x_2 with $x_i \in D_{X_i}$ and create a new random variable $y \in D_Y$ with $|D_Y| = |D_{X_1}| |D_{X_2}|$. With such a domain, we can form a one-to-one mapping between pairs of values x_1, x_2 and y , call this mapping $(x_1, x_2) \leftrightarrow y$, and then form distribution:

$$p(y) \triangleq p(x_1, x_2) \text{ for } (x_1, x_2) \leftrightarrow y \quad (6.90)$$

so that we still have $\sum_y p(y) = 1$. One possible mapping might be to do

$$y \leftarrow |D_{X_2}| x_1 + x_2 \quad (6.91)$$

and

$$x_1 \leftarrow \lfloor y / |D_{X_2}| \rfloor \quad (6.92)$$

$$x_2 \leftarrow y \bmod |D_{X_2}| \quad (6.93)$$

We can use the names as mapping functions. I.e., $y = y(x_1, x_2)$ corresponds to $y \leftarrow |\mathcal{D}_{X_2}|x_1 + x_2$, and vector-valued function $(x_1, x_2) = x_{1,2}(y)$ corresponds to the other direction.

We can generalize this for any vector of variables, i.e., if we have $W \subseteq V$, we can form a y with $|\mathcal{D}_Y| = \prod_{w \in W} |\mathcal{D}_Y|$, and produce the mapping $(x_1, x_2, \dots, x_{|W|}) \leftrightarrow y$.

Any Therefore, we see that some graphs can be viewed as trees once they have been embedded into these higher order variables.

Consider a 4-cycle with extra edge between x_3 and x_2 . create new super nodes $y_1 = (x_1, x_2, x_3)$ and $y_2 = (x_2, x_3, x_4)$. each of the y nodes are vector nodes, so that $y_1 \in \mathcal{D}_{X_1} \times \mathcal{D}_{X_2} \mathcal{D}_{X_3}$

The basic idea is to do one of these mappings for every separator. Given triangulated graph, for each separator s , construct a integer valued variable $y_s \leftrightarrow x_s$. So we have that $p(y_s) = p(x_s(y_s))$ thinking of x_s as a mapping function. Between two separators s_1, s_2 lies a clique c , where $s_1 \subseteq c$ and $s_2 \subseteq c$. It might also be the case that $s_1 \cup s_2 \subset c$ meaning that the clique has some unique variables that are not in any of its separators. In any event, we have functions $p(x_{s_1})$ $p(x_{s_2})$ and $p(x_c)$. For every clique c we associate an edge in the tree graph.

and consider the following distribution

talk about y nodes, with x at first being free, but then joint distribution of y can be defined, along with constraint making sure that corresponding x nodes are identical.

summing out y_1 really means summing out x_1, x_2, x_3

since the nodes are identical, we can just say that they are the same, and define a probability distribution over y nodes, as follows

$$p(y_1, y_2) = \quad (6.94)$$

we see that there is are two y nodes, and interaction terms between these two y nodes, so that there is a 2-node tree on which the y nodes factor.

Do this for a more general structure, a k tree, along with constraint functions that keep the different variables the same (corresponding to RIP, but don't say RIP yet). i.e., consider a 3-tree mesh kind of thing, then convert each clique into y variable, and each separator into a constraint on subset of the y vectors.

There is some key connection between trees and triangulated graphs it appears.

Chordal graphs are analogous to trees. The minimal separators of a chordal graph are like the non-leaf node (i.e., internal) single nodes in a tree.

Could add tree stuff here first? - decomposable - triangulated - junction tree exists - no-fill-in elimination order exists

triangulated graphs and trees have lots in common. hypertree

6.11 Bayesian networks and chordal graphs

remind reader the discussion of generality and specificity regarding BNs and MRFs and moralization to get from BN to covering MRF family.

Mention again that moralization allows us to find a clique that can represent any factor in the original BN, yielding the same exact distribution.

Show that if G is triangulated than we can add directions to the edges to form a DAG (BN) that when moralized gives G , and that we can find a BN that gives exactly the same family as G .

I.e., proof that the BN and MRF families intersect on the class of models given by triangulated graphs.

6.12 complexity of exact inference

We say that trying to find the best way to do inference is NP-complete.

Even so, doing inference itself is NP-complete. I.e. there are graphs such that doing inference is identical to 3-SAT. Show reduction.

6.13 other tricks

When interested down to original factor potentials (in factor graph or BN case), there might be > 1 clique that contains those factors. We choose the smallest one to marginalize down into. I.e., if we have $A_s \subset A_1$ and $A_s \subset A_2$ but $|A_2| < |A_1|$, then we might want to compute $\sum_{x_{A_2} \setminus x_{A_1}} \psi_{A_2}(x_{A_2})$ rather than $\sum_{x_{A_2} \setminus x_{A_1}} \psi_{A_2}(x_{A_2})$. On the other hand, if another potential was needed of the form $A_b \subset A_1$ but with A_b having a symmetric diff with A_2 and $A_b < A_2$, then it might be better to first compute A_b from A_1 and then compute A_s from A_b . This all depends on the underlying factors and the queries that are desired.

How to choose the root clique - can choose the one with the biggest cardinality. In general, that can help when there are zeros in the distribution, if the message is implemented in the right way.

Describe how to sort the messages when there is an option.

Should look at lepar98 paper and its references for some other JT-like data structures.

6.14 Other applications of triangulated graphs

Should we do this section here?

phylogenetic trees

relational data base schemes

Gaussian elimination

Perfect graphs

6.15 message passing on rings

Commutative semi-ring, generalize to max-product, and beyond.

Also k -best ring as example, define operations for.

6.16 space-time complexity trade offs

Perhaps this should be its own chapter.

Chapter 7

Overview of Approximate Inference Methods

Need to write.

Part III

Overview of Dynamic Graphical Models

Chapter 8

Dynamic Graphical Models

Many real-world signals are temporal processes, including speech, language, biological signals, economic and financial indicators, weather patterns, human activity patterns, brain waves, and so on. We wish to study and represent the way a natural process evolves over time and how it may correspond to an (also co-evolving) sequence of hidden events. Any graphical model meant to represent such processes must thus take time (or some sequential dimension such as length or position) into account.

Dynamic graphical models (DGMs) are graphs that represent the temporal evolution of the statistical properties of a temporal signal. Like all graphical models, there are many different types of DGM, and some are better at explicating certain phenomena than others. The fundamental property shared by all DGMs is the following: along one dimension, a DGM can be expanded to any arbitrary (and in some cases unbounded) length. The dimension along which a DGM can be expanded depends on the application: for example, it is *time* in speech recognition or financial modeling or any time-series application; it is *linguistic unit* (such as sentence, word, or sub-word, and so on, all of which are related to time) in natural language processing; or position or space in bioinformatics (e.g., amino acids in protein modeling, or base/nucleotides in DNA).

In all DGMs, there is a *template* and a set of *rules* for expansion. The pair (template,rules) along with non-negative integer T can be used to expand the model to length T . Alternatively, there are *online* applications where information is streaming into an application — in this case, for each new chunk of data that comes in, an additional expansion of the template is created and appended to what already exists. In most online applications, moreover, one need not keep the entire history of the expansion, and in fact one can keep only the expansion at $t - 1$ when one expands at time t . Hence, the sequence model can be arbitrarily long and there is no fixed T .

Throughout this document, we will use variables t as a point along the dimension of expansion, and T as the length of a sequence (when it exists). We will also use the word “time” to refer to a unit in the dimension of expansion, but “time” might really refer to “position” or “length” or whatever dimension is appropriate for your application.

Before going into a detailed discussion about dynamic graphical models (which we do in §8.10 and §8.11), in the next section we first review various forms of template models and template expansion methods. Next we start expanding our list of examples of dynamic graphical models starting with Markov chains §8.3, the many faces of hidden Markov models (HMMs) in §8.4, conditional random fields in §8.8, hierarchical HMMs in §8.9.2, dynamic Bayesian networks in §8.10, and end with dynamic graphical models in their most general form in §8.11. We also touch upon structured prediction in §8.13.

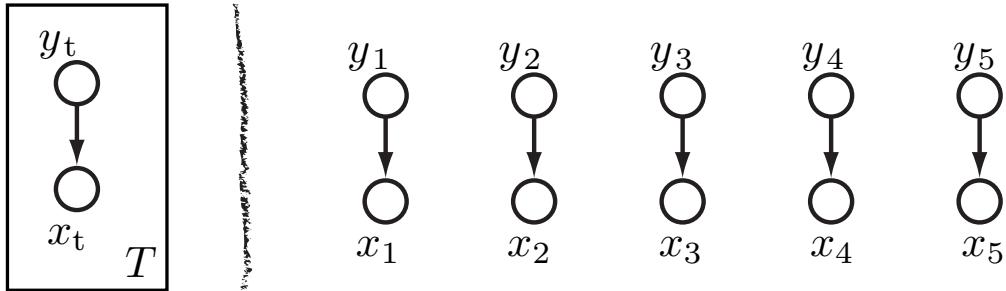


Figure 8.1: A template model corresponding to a sequence of independent mixtures. Left shows “plate” notation where T indicates degree of expansion. Right shows the plate instantiated with $T = 5$

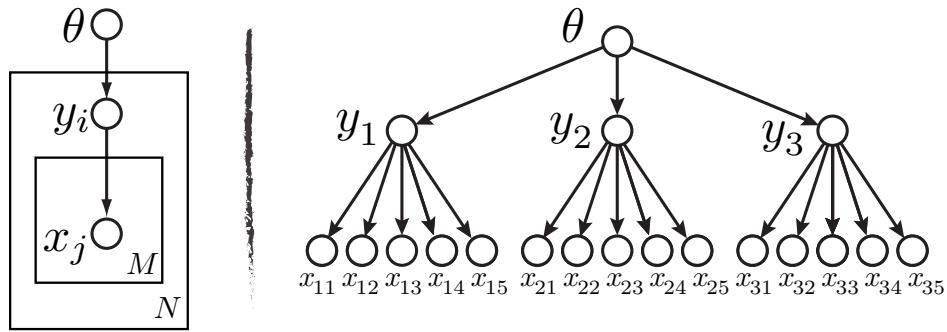


Figure 8.2: Left: An example of a plate template with two levels in a hierarchy. The top-most level is a variable θ which, since it is out of a plate, is not expanded at all. The next variable y_i is in a plate that is expanded M times. The inner-most variable x_j is in both plates, and thus is expanded MN times. At each expansion, the variables share parents from their outer level, thus y_i for all i has θ as a parent , and any x_j has a corresponding y_i as a parent. Right: An expansion of the template with $M = 3$ and $N = 5$. Since the x_i variables are expanded in two dimensions (both M and N), they now have two indices.

8.1 Template Graphical Models

A dynamic graphical model (DGM) is inherently a *template model*. In general, a template graphical model corresponds to a collection of graphs each of which may be expanded from some template and based on some rules for expansion. The template is static and usually corresponds to a piece of a graph and possibly some other annotations. The rules for expansion say how a template may be instantiated to form a particular (expanded) model instance. The family of distributions corresponding to the template is the union of all probability distributions that obey the rules for any one of these expansions.

Before going further, we characterize more precisely what we mean by a template model — we do this by explaining a variety of different ways that a template may be expanded, starting from the simplest non-expandable static model that we saw in §2. Terminologically, we say that a template may be expanded or instantiated, and use the terms interchangeably. A graphical model that is derived from a template is called an instantiated model.

- In a static graphical model, the graph G is static and directly characterizes the family $\mathcal{F}(G, \mathcal{M})$. The properties of G directly determine which distributions p are or are not members of the family. As mentioned above, G itself can determine the computational cost of performing inference (or at least an upper bound on this computation cost), and operations based on G alone can often help to produce fast exact or approximate inference algorithms as shown in Figure 6.1.

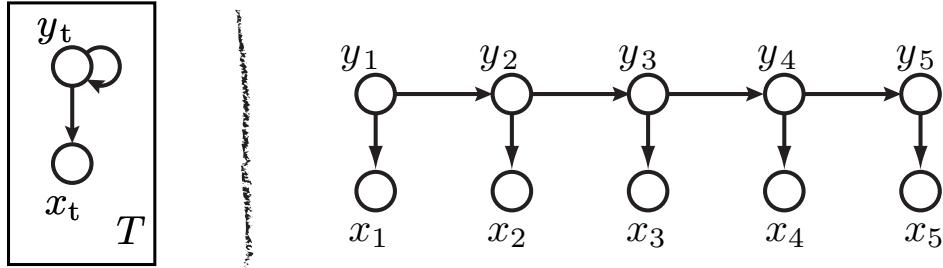


Figure 8.3: Left: An example of a plate template similar to Figure 8.1 except that here, there is a self-loop in the template. Normally self-loops are not allowed in Bayesian networks. In a plate template, the self loop indicates that there is to be an edge from a variable to its successor in any expansion. All such edges correspond to persistent edges (edges from a variable to itself in the next time frame). In fact, in this case, the template is able to generate the class of HMMs.

- Plate models: plate model [218] were developed as a convenience to represent the fact that a portion of a graph can be expanded any number of times, say M . The plate consists of a graph stub surrounded by a rectangle, and an integer that indicates the number of times that the stub should be repeated.

If there are no edges crossing the plate, then the graph is simply expanded M times corresponding to M set of random variable groups, where the groups are mutually independent of each other but dependencies exist within each group identical to the plate template. An example is shown in Figure 8.1.

If there are edges crossing the plate (between variables outside and inside of the plate), however, then this allows for coupling to occur between stub instantiations. For example, if there is one variable external to the plate that is a parent of some within-stub variable, after expansion this might be seen as a random variable corresponding to a Bayesian parameter prior. The plate expansion, moreover, can be repeated hierarchically so as to be able to quickly express a wide variety of interesting models, where the total expansion corresponds to the Cartesian product of the expansion factors. For example, one can easily express hierarchical Bayesian models using plate notation. Figure 8.2 shows an example where in fact there is a hierarchy of 2 nested plate models.

In a plate model, the stub itself might possess self-loops, something normally disallowed in a Bayesian network. This corresponds to edges between the same variable in successive instantiations of the stub after the plate expansion into a totally ordered sequence of instantiations — these are akin to *persistent* edges in a dynamic graphical model, as we will see below. In fact, an HMM can be described using a plate model under this mechanism, as shown in Figure 8.3.

- Dynamic graphical model: A DGM also consists of a template and a set of rules for expansion along one dimension, such as time or position. Time is discrete, and we say that a DGM has been expanded to T frames or slices. We will use these terms interchangeably. For example, the well-known hidden Markov model is an example of a DGM (cf. §8.4).

A key aspect of the DGM is that its template and expansion rules are typically sufficient to infer computational properties of any model after expansion, regardless of the amount of expansion, and this can be done based only on the template. This means that, like static models (and as depicted in Figure 6.1), it should be possible to perform offline computation to produce an inference algorithm for any probability model in the family, regardless of the expansion. Indeed, for GMTK, this is shown in

Note that plate models share this property. However, unlike in the plate case, the DGM expansion rules no longer need to require dependencies between only corresponding variables in successively

expanded chunks (non-persistent edges are possible). In other words we can specify dependencies between, say X_t and Y_{t+1} . There is of course even more flexibility than this. Dynamic Bayesian networks (DBNs) are an example of this and are described in §8.10. DGMs are more fully described in §8.11.

- General template models: The notion of a template and a set of rules for expansion can be generalized in a variety of ways. For example, probabilistic relational models [158], multi-dynamic Bayesian networks [145], and Markov logic [360] are three examples of classes of models where the family of probability distributions corresponding to the set of expanded models can be incredibly diverse and flexible. This diversity, however, often means that it is difficult if not impossible to infer computational properties of any possible expansion based only on the template. At best, there is not as much “offline” processing that can be performed based only on the template, and we need instead to produce inference procedures for each template expansion. On the other hand, usually such inference procedures can take advantage of the fact that the model did originally come from an expanded template (see for example the lifted inference procedures discussed in [64]).

In this document, we do not consider these models further. An open research question, however, is what is the limit of type of model where computational properties of any expansion can be inferred only from the template: DGMs have this property, as do grid-based Markov random fields, but it is not yet known if there are other classes with this property as well.

8.2 Stochastic Processes, Discrete-time Markov Chains, and Correlation

A discrete-time stochastic process is a collection $\{X_t\}$ for $t \in 1 : T$ of random variables ordered by the discrete index t that often corresponds to time or one-dimensional position. In general, the distribution for each of the variables X_t can be arbitrary and different for each t . There may also be arbitrary conditional independence relationships between different subsets of variables of the process — this means that such a set of random variables lies within the family corresponding to a graphical model that may have edges between an arbitrary set of vertices.

Certain types of stochastic processes are common because of their analytical and computational simplicity. One example follows:

Definition 115. Independent and Identically Distributed (i.i.d.) *The stochastic process is said to be i.i.d. [97, 334, 126] if the following condition holds:*

$$p(X_t = x_t, X_{t+1} = x_{t+1}, \dots, X_{t+h} = x_{t+h}) = \prod_{i=0}^h p(X = x_{t+i}) \quad (8.1)$$

for all t , for all $h \geq 0$, for all $x_{t:t+h}$, and for some distribution $p(\cdot)$ that is independent of the index t .

An i.i.d. process therefore comprises an ordered collection of independent random variables each one having exactly the same distribution. A graphical model of an i.i.d. process contains no edges at all and can easily be described by a plate model with a single variable within a box.

An i.i.d. process satisfies what is known as exchangeability — the sequence of random variables is exchangeable when any reordering of the variables does not change the probability. That is, if σ is a permutation of the indices, then for any set of values $x_{t:t+h}$, we have

$$p(X_t = x_t, X_{t+1} = x_{t+1}, \dots, X_{t+h} = x_{t+h}) = p(X_{\sigma(t)} = x_t, X_{\sigma(t+1)} = x_{t+1}, \dots, X_{\sigma(t+h)} = x_{t+h}) \quad (8.2)$$

If the statistical properties of variables within a time-window of a stochastic process do not evolve over time, the process is said to be stationary.

Definition 116. *Stationary Stochastic Process* The stochastic process $\{X_t : t \geq 1\}$ is said to be (strongly) stationary [179] if the two collections of random variables

$$\{X_{t_1}, X_{t_2}, \dots, X_{t_n}\}$$

and

$$\{X_{t_1+h}, X_{t_2+h}, \dots, X_{t_n+h}\}$$

have the same joint probability distributions for all n and h .

In the continuous case, stationarity means that $F_{X_{t_1:n}}(a) = F_{X_{t_1:n+h}}(a)$ for all a where $F(\cdot)$ is the cumulative distribution and a is a valid vector-valued constant of length n .¹ In the discrete case, stationarity is equivalent to the condition

$$P(X_{t_1} = x_1, X_{t_2} = x_2, \dots, X_{t_n} = x_n) \quad (8.3)$$

$$= P(X_{t_1+h} = x_1, X_{t_2+h} = x_2, \dots, X_{t_n+h} = x_n) \quad (8.4)$$

for all t_1, t_2, \dots, t_n , for all $n > 0$, for all $h > 0$, and for all x_i . Every i.i.d. process is stationary.

The covariance between two arbitrary random vectors X and Y is defined as:

$$\text{cov}(X, Y) \quad (8.5)$$

$$= E[(X - EX)(Y - EY)'] = E(XY') - E(X)E(Y)' \quad (8.6)$$

where Y' is the transpose of vector Y . It is said that X and Y are uncorrelated if $\text{cov}(X, Y) = \vec{0}$ (equivalently, if $E(XY') = E(X)E(Y)'$) where $\vec{0}$ is the zero matrix. If X and Y are independent, then they are uncorrelated, but not vice versa unless they are jointly Gaussian [179].

8.3 Markov Chains

Perhaps the simplest dynamic graphical model, next to an i.i.d. sequence, is a (first order) Markov chain. Like any graphical model, a Markov chain is a set of random variables that obeys a set of factorization constraints. In this case, those constraints are identical to a set of conditional independent requirements. A Markov chain can be expanded to any length, but the set of requirements grows with the length of the chain. Here's the definition.

Definition 117 (Markov chain). *A collection of discrete-valued random variables $\{Y_t : t \geq 1\}$ forms an n^{th} -order Markov chain if*

$$P(Y_t = y_t | Y_{t-1} = y_{t-1}, Y_{t-2} = y_{t-2}, \dots, Y_1 = y_1) = \quad (8.7)$$

$$P(Y_t = y_t | Y_{t-1} = y_{t-1}, Y_{t-2} = y_{t-2}, \dots, Y_{t-n} = y_{t-n}) \quad (8.8)$$

for all $t \geq 1$, and all y_1, y_2, \dots, y_t .

In other words, given the previous n random variables, the current variable is conditionally independent of every variable earlier than the previous n . Note that this definition does not imply that a variable is independent of future variable — on the contrary, a variable might be quite dependent on future variables. This definition only states that a variable is independent of the distant-past given the recent past. Bayesian networks of n^{th} -order Markov chains for various n are depicted in Figure 8.4. A zero'th order Markov chain is i.i.d..

¹We assume that scalars can add to vectors in our notation, so $t_{1:n} + h \equiv (t_1 + h) : (t_n + h)$.

One often views the event $\{Q_t = i\}$ as if the chain is “in state i at time t ” and the event $\{Q_t = i, Q_{t+1} = j\}$ as a transition from state i to state j starting at time t . This notion arises from viewing a Markov chain as a finite-state automaton (FSA) [197] with probabilistic state transitions, a topic we will touch on in §8.4.4. In this case, the number of states corresponds to the domain size of the state random variable. In general, a Markov chain may have unboundedly many states, and some recent models can effectively deal with such unbounded numbers of states by imposing a Dirichlet process prior distribution on the states (or some other non-parametric Bayesian prior) during the learning process. In this section, however, we will only work with a finite number of states, so the random variables have finite cardinality domain sizes.

With a 1st order Markov chains ($n = 1$), each of the factors only involves Y_{t-1} . This means that we can write the joint distribution over T variables as follows:

$$p(y_1, y_2, \dots, y_T) = p(y_1) \prod_{t=2}^T p(y_t | y_1, y_2, \dots, y_{t-1}) \quad (8.9)$$

$$= p(y_1) \prod_{t=2}^T p(y_t | y_{t-1}) \quad (8.10)$$

Each of the factors in this latter case involve only two successive in time variables. We say that y_t *directly depends* on y_{t-1} . This does not mean that y_t is independent of y_{t+1} or other future variables. In fact, there are no independent assumptions made in this model — by this, we mean that there are no two variables that are independent of each other. That is, in a Markov chain, $Y_t \perp\!\!\!\perp Y_\tau$ is **not** necessarily true for *any* t and τ .

There are many *conditional* independence properties, however. In this model, the future and past are independent given the present. This general property (“the future and past are independent given the present”) is in fact true for any DGM as we will see, but the definition of future, past, and present depends very much on the model. In a first order Markov chain, the present is the current state Y_t and the past (future) is all states before (after) Y_t . This means that $Y_{A_b} \perp\!\!\!\perp Y_{A_a} | Y_t$ for any t and set of time indices A_b and A_a where for all $\tau \in A_b, \tau < t$ and for all $\tau \in A_a, \tau > t$. Thus, in a 1st order Markov chain, the “present” is fairly local.

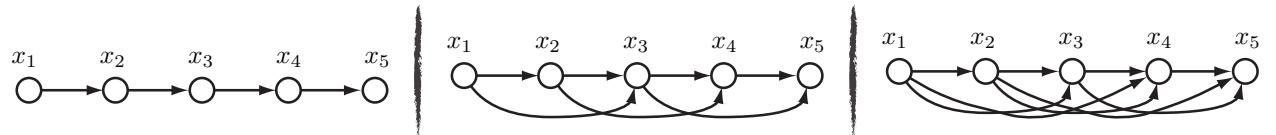


Figure 8.4: Bayesian network views of Markov chains of various orders. Left: a first-order Markov chain. Center: a second-order Markov chain. Right: a third-order Markov chain.

8.3.0.1 Higher order Markov chains/conversion of n^{th} order to first order

Higher order Markov chains also have the past and future independent given the present property. These models, however, have longer term notions of what the “present” is. Figure 8.4 shows a first order Markov chain (left), a second order (middle), and a third order (right). In a 2nd order Markov chain, the “present” is a pair of successive states Y_t, Y_{t+1} , and in a 3rd order Markov chain, the present is a triple of successive states Y_t, Y_{t+1}, Y_{t+2} . Obviously, in an n^{th} -order Markov chain, the present is the set n of states $\{Y_t, Y_{t+1}, \dots, Y_{t+n-1}\}$. In each case, the “present” renders the past and future independent.

An n^{th} -order Markov chain may be converted into an equivalent first-order Markov chain by forming a new 1st order Markov chain who’s “present” representation is the same as the n^{th} -order Markov chain’s “present” representation. That is, we can perform the following transformation:

$$Y'_t \triangleq \{Y_t, Y_{t-1}, \dots, Y_{t-n}\}$$

where $(Y_t)_t$ is an n^{th} -order Markov chain. Thus, Y'_t is a vector valued random variable that exists at time t , and for this to be a Markov chain, we need to make sure that the independence relationships between Y'_t and Y'_{τ} are followed. Indeed, Y'_t is a first-order Markov chain because

$$\begin{aligned} P(Y'_t = y'_t | Y'_{t-1} = y'_{t-1}, Y'_{t-2} = y'_{t-2}, \dots, Y'_1 = y'_1) \\ = P(Y_{t-n:t} = y_{t-n:t} | Y_{1:t-1} = y_{1:t-1}) \\ = P(Y_{t-n:t} = y_{t-n:t} | Y_{t-n-1:t-1} = y_{t-n-1:t-1}) \\ = P(Y'_t = y'_t | Y'_{t-1} = y'_{t-1}) \end{aligned}$$

Thus, $Y'_{t_b} \perp\!\!\!\perp Y'_{t_a} | Y'_t$ for any $t_b < t < t_a$.

Hence, given a large enough state space, first-order Markov chain may represent any n^{th} -order Markov chain. Note, however, that while Y' is a first order Markov chain, the state space of Y' is much bigger than that of Y . Indeed, if $Y \in \mathcal{D}_Y$, then $Y'_t \in \mathcal{D}_{\{Y_t, Y_{t-1}, \dots, Y_{t-n}\}}$ and $|\mathcal{D}_{Y'_t}| = |\mathcal{D}_Y|^n$. Hence, the state space in a converted lower-to-higher order Markov chain is exponentially bigger than in the first order case.

Also note that we can use a single integer to represent the (discrete valued) vector Y'_t so we can collapse any vector representation of state variables down to single integers if we wish. The resulting first-order Markov chain, however, has a number of states that is exponentially large in n . And using a single integer for Y' loses part of the original “structure” in the chain (cf. §8.13).

We note that in the context of DGMs, where the state space is structured, the representation of a higher-order by a first order can be improved in some cases by taking advantage of this structure. This issue is discussed in the context of general DGMs in §13.3.4.

8.3.0.2 State transition probabilities

In general, the statistical evolution of a 1st-order Markov chain is determined by the state transition probabilities $a_{ij}(t) \triangleq P(Y_t = j | Y_{t-1} = i)$. Note that this is a function both of the states at successive time steps and of the current time t . More often than not, however, there is no dependence on t , meaning that $a_{ij}(t) = a_{ij}(\tau)$ for all t, τ . When this is the case, the Markov chain is called *time-homogeneous* (or just *homogeneous*).

The transition probabilities of a time-homogeneous Markov chain are determined by a transition matrix A where $a_{ij} \triangleq (A)_{ij}$. The rows of A form potentially different probability mass functions over the states of the chain. For this reason, A is also called a stochastic transition matrix (or just a transition matrix).

Because of the homogeneity property, it is possible to use model with a fixed and finite number of parameters to represent an arbitrary unbounded length sequence. That is, the model for:

$$p(y_1, y_2, \dots, y_T) = p(y_1) \prod_{t=2}^T p(y_t | y_{t-1}) \quad (8.11)$$

requires only $\mathcal{O}(r^2)$ parameters, regardless of how big T is, where $r = |\mathcal{D}_{Y_t}| = |\mathcal{D}_{Y_\tau}| \forall t, \tau$. This is an important property of dynamic graphical models in general, as we will see in §8.11.

Because the transition probabilities in homogeneous Markov chains stay fixed for all time, they can be represented by one fixed transition matrix A where $p(Y_t = j | Y_{t-1} = i) = a_{ij} \triangleq (A)_{ij}$.

It is common to show a 1st-order Markov chain via its transition matrix as a graph $G = (\mathcal{Q}, \mathcal{A})$ of nodes \mathcal{Q} and arcs \mathcal{A} . The graph consists of a set of nodes corresponding to the states of the Markov chain, and the arcs depict the allowable transitions in the HMM’s Markov chain. Each node corresponds to one of the states in \mathcal{D}_Q , where an arc going from node i to node j indicates that $a_{ij} > 0$, and the lack of such an arc

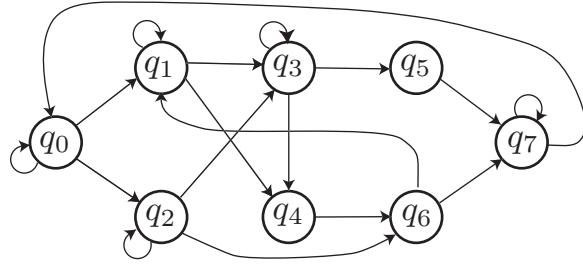


Figure 8.5: Markov chain transition matrix shown as a graph of nodes and arcs. There are eight nodes $\mathcal{Q} = \{q_1, q_2, \dots, q_8\}$. Only the possible (i.e., non-zero probability) Markov chain state transitions/arcs are shown.

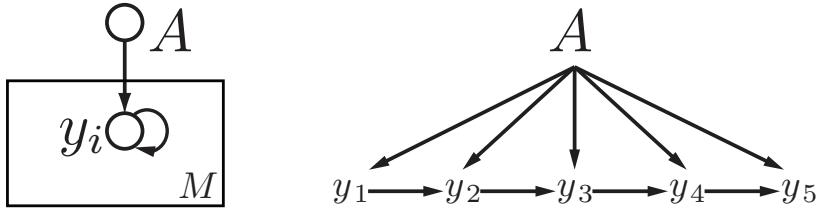


Figure 8.7: A time-homogeneous Markov chain viewed in a Bayesian fashion. Here, the values of the transition matrix $p(y_t|y_{t-1}, \Theta_{y_t|y_{t-1}})$ is governed by the random variable $\Theta_{y_t|y_{t-1}}$. If it is the case that $\Theta_{y_t|y_{t-1}}$ is a constant random variable, then this corresponds to a standard time-homogeneous Markov chain. A Bayesian prior on the parameters, however, would lead to a more general Markov chain. For example, if $\Theta_{y_t|y_{t-1}}$ is time-dependent but otherwise constant, then we would represent a time-inhomogeneous Markov chain. If $\Theta_{y_t|y_{t-1}}$ is random, we would have a random Markov chain.

indicates that $a_{ij} = 0$. The transition matrix associated with Figure 8.5 is as follows:

$$A = \begin{pmatrix} a_{00} & a_{01} & a_{02} & 0 & 0 & 0 & 0 & 0 \\ 0 & a_{11} & 0 & a_{13} & a_{14} & 0 & 0 & 0 \\ 0 & 0 & a_{22} & a_{23} & 0 & 0 & a_{26} & 0 \\ 0 & 0 & 0 & a_{33} & a_{34} & a_{35} & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & a_{46} & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & a_{57} \\ 0 & a_{61} & 0 & 0 & 0 & 0 & 0 & a_{67} \\ a_{70} & 0 & 0 & 0 & 0 & 0 & 0 & a_{77} \end{pmatrix}$$

where it is assumed that the explicitly mentioned a_{ij} are non-zero. In this view, an HMM is seen as an stochastic finite state automaton (or SFSA) [294]. We discuss FSAs more in § 8.4.4.

A state of a Markov chain may be categorized into one of three distinct categories [179]. A state i is said to be *transient* if, after visiting the state, it is possible for it never to be visited again, i.e.:

$$p(Q_n = i \text{ for some } n > t | Q_t = i) < 1.$$

A state i is said to be *null-recurrent* if it is not transient but the expected return time is infinite (i.e., $E[\min\{n > t : Q_n = i\} | Q_t = i] = \infty$). Finally, a state is *positive-recurrent* if it is not transient and the expected return time to that state is finite. For a Markov chain with a finite number of states, a state can only be either transient or positive-recurrent. In Figure 8.6, q_0 is transient while states q_1, q_2 , and q_3 are positive-recurrent.

A time-homogeneous Markov chain can be represented using a Bayesian model, as shown in Figure 8.7.

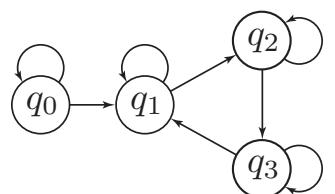


Figure 8.6: State types.

8.3.0.3 Markov chains: stationarity vs. time-homogeneity

Like any stochastic process, an individual Markov chain might or might not be a stationary process. The stationarity condition of a Markov chain, however, depends on 1) whether or not the Markov chain transition matrix has (or “admits”) a stationary distribution, and 2) if the current distribution over states is one of those stationary distributions.

If Q_t is a time-homogeneous stationary first-order Markov chain then:

$$P(Q_{t_1} = q_1, Q_{t_2} = q_2, \dots, Q_{t_n} = q_n) \quad (8.12)$$

$$= P(Q_{t_1+h} = q_1, Q_{t_2+h} = q_2, \dots, Q_{t_n+h} = q_n) \quad (8.13)$$

for all t_i , h , n , and q_i . Using the first order Markov property, the above can be written as:

$$P(Q_{t_n} = q_n | Q_{t_{n-1}} = q_{n-1}) \quad (8.14)$$

$$P(Q_{t_{n-1}} = q_{n-1} | Q_{t_{n-2}} = q_{n-2}) \quad (8.15)$$

$$\dots P(Q_{t_2} = q_2 | Q_{t_1} = q_1) P(Q_{t_1} = q_1) \quad (8.16)$$

$$= P(Q_{t_n+h} = q_n | Q_{t_{n-1}+h} = q_{n-1}) \quad (8.17)$$

$$P(Q_{t_{n-1}+h} = q_{n-1} | Q_{t_{n-2}+h} = q_{n-2}) \quad (8.18)$$

$$\dots P(Q_{t_2+h} = q_2 | Q_{t_1+h} = q_1) P(Q_{t_1+h} = q_1) \quad (8.19)$$

Therefore, a time-homogeneous Markov chain is stationary only when $P(Q_{t_1} = q) = P(Q_{t_1+h} = q) = P(Q_t = q)$ for all $q \in D_Q$ (clearly, this is an if and only if condition). This is called a stationary distribution of the Markov chain and will be designated by ξ with $\xi_i = P(Q_t = i)$.²

According to the definition of the transition matrix, a stationary distribution has the property that $\xi A = \xi$ implying that ξ must be a left eigenvector of the transition matrix A . For example, let $p_1 = [.5, .5]$ be the current distribution over a 2-state Markov chain (using matlab notation). Let $A_1 = [.3, .7; .7, .3]$ be the transition matrix. The Markov chain is stationary since $p_1 A_1 = p_1$. If the current distribution is $p_2 = [.4, .6]$, however, then $p_2 A_1 \neq p_2$, so the chain is no longer stationary.

In general, there can be more than one stationary distribution for a given Markov chain (as there can be more than one eigenvector of a matrix). The condition of stationarity for the chain, however, depends on if the chain “admits” a stationary distribution, and if it does, whether the current marginal distribution over the states is one of the stationary distributions. If a chain does admit a stationary distribution ξ , then $\xi_j = 0$ for all j that are transient and null-recurrent [179]; i.e., a stationary distribution has positive probability only for positive-recurrent states (states that are assuredly re-visited).

The time-homogeneous property of a Markov chain is distinct from the stationarity property. Stationarity, however, does imply time-homogeneity. To see this, note that if the process is stationary then $P(Q_t = i, Q_{t-1} = j) = P(Q_{t-1} = i, Q_{t-2} = j)$ and $P(Q_t = i) = P(Q_{t-1} = i)$. Therefore, $a_{ij}(t) = P(Q_t = i, Q_{t-1} = j)/P(Q_{t-1} = j) = P(Q_{t-1} = i, Q_{t-2} = j)/P(Q_{t-2} = j) = a_{ij}(t-1)$, so by induction $a_{ij}(t) = a_{ij}(t+\tau)$ for all τ , and the chain is time-homogeneous.

On the other hand, time-homogeneity does not require stationarity as there can be many non-stationary distributions under a homogeneous Markov chains. A time-homogeneous Markov chain, however, must always admit at least one a stationary distribution.

Note that an inhomogeneous Markov chain might seem like it has a stationary distribution, and in fact the marginal distribution of states at time t (or $p(Q_t = q)$) might not change over time for inhomogeneous Markov chains. For example, let $A_t = [.3, .7; .7, .3]$ when t is even and $A_t = [.4, .6; .6, .4]$ when t is odd. Then the Markov chain is inhomogeneous but if the current state distribution is $p = [.5, .5]$, then $pA_t = p$

²Note that in the speech recognition literature, the symbol π is often used to indicate the initial (i.e., at time $t = 1$) state distribution which might or might not be a stationary distribution of the Markov chain.

for both even and odd values of t . Note that this is not a stationary distribution. When t is even, we have that $p(Q_t = 0, Q_{t+1} = 1) = 0.5 \times 0.3$ but when t is odd, $p(Q_t = 0, Q_{t+1} = 1) = 0.5 \times 0.4$, so the chain does not exhibit a stationary distribution according to the definition. In general, the aforementioned criterion for stationary ($\xi A = \xi$) requires a homogeneous chain. The condition $\xi A_t = \xi$ alone does not guarantee stationarity.

In general, it is important to realize that stationarity and homogeneity of a Markov chain (or any DGM) are distinct properties. If the chain is stationary, then it is homogeneous. If it is homogeneous, then might or might not be stationary.

8.3.0.4 Chapman-Kolmogorov equations

The idea of “probability flow” may help to determine if a first-order Markov chain is in a stationary distribution. Stationarity, or $\xi A = \xi$, implies that for all i

$$\xi_i = \sum_j \xi_j a_{ji}$$

or equivalently,

$$\xi_i(1 - a_{ii}) = \sum_{j \neq i} \xi_j a_{ji}$$

which is the same as

$$\sum_{j \neq i} \xi_i a_{ij} = \sum_{j \neq i} \xi_j a_{ji}$$

The left side of this equation can be interpreted as the probability flow out of state i and the right side can be interpreted as the flow into state i . A stationary distribution requires that the inflow and outflow cancel each other out for every state.

In a first-order time-homogeneous Markov chain, with a_{ij} the transition probability from state i to state j , let a_{ij}^k be the probability of transitioning from state i to state j in exactly k steps. That is,

$$a_{ij}^k \triangleq p(Q_{t+k} = j | Q_t = i) \quad (8.20)$$

which holds $\forall t$ since the chain is homogeneous. Then we first note that

$$a_{ij}^2 = p(Q_{t+2} = j | Q_t = i) \quad (8.21)$$

$$= \sum_{\ell} p(Q_{t+2} = j, Q_{t+1} = \ell | Q_t = i) \quad (8.22)$$

$$= \sum_{\ell} p(Q_{t+2} = j | Q_{t+1} = \ell) p(Q_{t+1} = \ell | Q_t = i) \quad (8.23)$$

$$= \sum_{\ell} a_{i\ell} a_{\ell j} \quad (8.24)$$

and more generally, we have that

$$a_{ij}^k = \sum_{\ell} a_{i\ell}^m a_{\ell j}^n \quad (8.25)$$

where $m, n \geq 0$ are both such that $k = m+n$. This is known as the Chapman-Kolmogorov equation for first-order time-homogeneous Markov chains, and can be generalized to n^{th} -order and/or time-inhomogeneous chains.

There are many good books that describe Markov chains and their mathematical properties. My favorite one is [180].

8.4 The Many Faces of Hidden Markov Models

Like all graphical models, an Hidden Markov Models (HMMs) is a (uncountably infinite sized) family of models rather than just one model. Hidden Markov models, in fact, are the most widely used of all classes of dynamic Graphical model, and therefore we spend a bit of extra effort describing them in detail. HMMs are used for countless speech recognition, natural language processing, and bioinformatics applications. HMMs are also misconstrued in equally countless research papers [54]. GMTK can easily train, represent, and perform inference on HMMs as we will see. Starting with an HMM is often an easy way to get started both with GMTK and with your model — once you have a working HMM, it is easy to add additional variables thereby extending it into a DBN.

HMMs were originally developed as a mathematical curiosity (see §0.8.3), and in fact they were first conceived of as early as Shannon's famous and influential 1948 paper [393] that happened, also, to introduce to the world the field of information theory.³ This was followed by papers such as [183, 59, 171] where they were called “functions of finite Markov chains” (where the observation distributions were deterministic), and where learning the transition matrix A was termed the “identifiability problem.” In [21, 22], the more general fully-stochastic HMMs were defined and termed “stochastic functions of finite Markov chains.” A thorough account of the history of HMMs is given in [138]. HMMs were significantly advanced by the speech recognition communities in the late 1960s through the 1980s [265, 10, 264, 225, 349, 348, 345, 99, 42, 30]. HMMs were also developed and used for many applications in computational biology [253]. These days, it is well known that an HMM is just one particular instance of a DGM, as first noted in [398].

We will introduce HMMs by viewing them in a number of different ways. In §8.4.1, we'll view them as random balls drawn from random urns. In §8.4.2, we'll first view them as (deterministic or random) functions of a Markov chain. In §8.4.3, we'll view them as simple (but exponentially large) mixture distributions. In §8.4.4, we'll view them as stochastic finite state automata (SFSA). In §??, we'll see them as weighted finite state transducers. In §8.4.5, we'll view them as a trellis (or lattice). In §8.4.8, we'll see them as smoothed (or regularized) unary potential functions. In §8.4.9, we'll see them as dynamic graphical models (both Dynamic Bayesian networks and as Dynamic undirected graphical models).

8.4.1 HMM as random balls drawn from random urns

In the widely read and cited paper [348], an HMM is introduced as a collection of urns each containing a different proportion of colored balls (see Figure 8.8). Sampling (generating data) from an HMM occurs by choosing a new urn based on only the previously chosen urn, and then choosing with replacement a ball from this new urn. The sequence of urn choices are not made public (and are said to be “hidden”) but the ball choices are known (and are said to be “observed”). Along this line of reasoning, an HMM can be defined in such a generative way, where one first generates a sequence of hidden (urn) choices, y_1, y_2, \dots, y_T , and then generates a sequence of observed (ball) choices x_1, x_2, \dots, x_T where the probability of choosing a particular ball a time t is determined entirely by the urn y_t that happened to have been chosen at that time, i.e., $\Pr(X_t = \text{a color} | Y_t = \text{an urn})$.

Note that given a length T , we can equivalently generate from this process by either of the following:

- Randomly choose a sequence of urns y_1, y_2, \dots, y_T based on the urn transition probabilities, and once done, for each urn choose a ball based on $\Pr(x_t | y_t)$.

³In fact in Shannon's 1948 paper [393], Section 4 Figure 3 gives an example of a 1-state HMM, Figure 4 gives an example of a 3-state HMM where the observations are a deterministic function of the state, and Figure 5 gives a full HMM with both stochastic transitions and observations where the stochastic observations are expressed by there being more than one possible transition (edge) between two states. More discussion of this is given in §8.4.4.6.

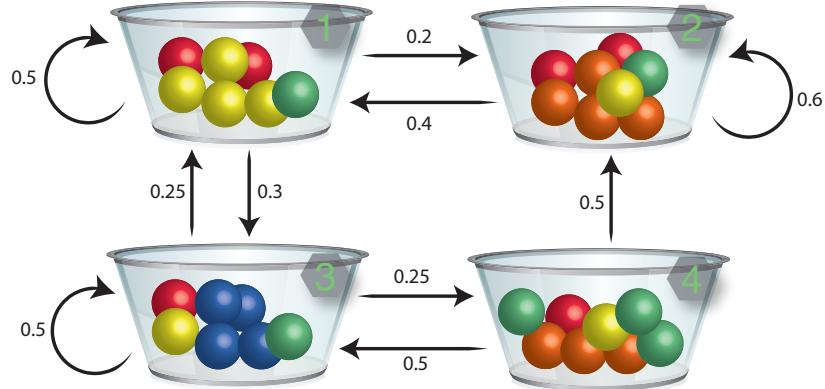


Figure 8.8: A 4-state HMM viewed as balls in urns. Each HMM state corresponds to an urn and the directed edges between urns give the state transition matrix, where missing edges correspond to zeros in the matrix. In each state, there is a distribution on colored balls. For example, in state one, we have $\Pr(\text{color} = \text{yellow}|\text{state} = 1) = 4/7$ while $\Pr(\text{color} = \text{red}|\text{state} = 1) = 2/7$, $\Pr(\text{color} = \text{green}|\text{state} = 1) = 1/7$, and $\Pr(\text{color} = \text{blue}|\text{state} = 1) = 0$. That is, in each state (urn), the probability of drawing a color is equal to the number of balls of that color in that urn divided by the total number of balls.

- Interleave the following: For $t = 1 \dots T$, choose an urn y_t at time t randomly based on the urn already chosen at time $t - 1$, and then choose a ball based on the distribution $p(x_t|y_t)$.

At the first time step, there is an initial distribution $\Pr(Y_1 = j)$ which gives a distribution on which urn to choose. It is not a conditional distribution which thus allows the process to begin.

We note that the urn/ball description of an HMM is simply a physical generative random process that happens to precisely match the assumptions made by an HMM. If one, for example, were to modify the ball picking process so that at each time step, one say chooses from urn y_t only if a draw from the urn at time step T (with $t < T$) results in a non-blue ball, and otherwise chooses randomly from some other urn, then this would no longer be precisely represented with an HMM. Of course, there are an unlimited number of ways of modifying the generative process so that it is no longer exactly an HMM. As we will see in subsequent sections, sometimes it is still possible to reduce the new process back into an HMM.

8.4.2 HMM as Random function of a Markov Chain

Consider a first-order Markov chain (§ 8.3). Normally, in such a chain, the variables are observable, meaning that we can directly see or use the values of the chain, and given a set of values $\bar{y} = \bar{y}_{1:T}$, we can directly produce a probability score $p(\bar{y}_1|\bar{y}_0)p(\bar{y}_2|\bar{y}_1)\dots p(\bar{y}_T|\bar{y}_{T-1})$. What if instead we are not able to directly observe the Markov chain, but we know it exists? This means that the Markov model is *hidden* from view, but is presumed to exist (i.e., it is a “known unknown”).

Let $y = y_{1:T}$ be a length T 1st-order time-homogeneous Markov chain (for the time being, all Markov chains will be first order unless otherwise indicated). We define another sequence $x = x_{1:T}$ to be related to y by the sequence mapping $x = f(y)$ where $f : D_Y^T \rightarrow D_X^T$ is a function (which might or might not itself be random) that maps from a sequence of Y random variable values to a sequence of X random variable values. Then we have that the sequence x is a function of the Markov chain y .

To make it more clear that f can be a random function, we make this randomness explicit. That is, suppose further that we have a set of functions $\{f_1, f_2, \dots, f_s\}$, each of which can map from the entire y sequence to the entire x sequence. I.e., each f_i is such that $f_i : D_Y^T \rightarrow D_X^T$ and each f_i is now a deterministic function mapping a Y sequence to an X sequence. Also, assume there is a random variable $p(S = s)$ that selects one of the functions. We can then define a *random sequence* by taking the Markov

chain y and producing a random sequence X via the random function, that is $X = f_S(y)$. Here, $X = f_S(y)$ is a random variable since it is a function of random variable S as y is fixed.

In fact, this process can be made “doubly random”, by taking both the Markov chain as the random variable itself Y (i.e., a random sequence) and also the random variable S , and then forming $X = f_S(Y)$. Here X is doubly random since it is a function of two random variables S and Y . We can then view X is a probabilistic function of a Markov chain. The random generation process would be to choose a random sample $y \sim p(Y = y)$, then choose a random $s \sim p(S = s)$, and finally form $\bar{x} = f_s(y)$ for those samples s, y . We can thus define a random process in this way: we observe \bar{x} which presumably, based on our model, has been generated by choosing one f_i randomly based on some random assignment to Y . The *identifiability problem* is to identify s and y for a given X , a problem that has become known as “Viterbi decoding.”

There are many ways of producing such mapping $\{f_i\}_i$. For example, all of x might be directly influenced by all of y . To get an HMM, significant restrictions are placed on the class of functions that map from y to x . That is, HMMs do not allow every x_t within $x = (x_1, x_2, \dots, x_T)$ to be a joint function of simultaneously every y_t within $y = (y_1, y_2, \dots, y_T)$. With an HMM, in fact, each x_t is allowed to be “directly dependent” only on y_t and not directly dependent on any other variable. A better way to say this is that in an HMM, for all t , the variable x_t is conditionally independent of all the other variables given y_t .

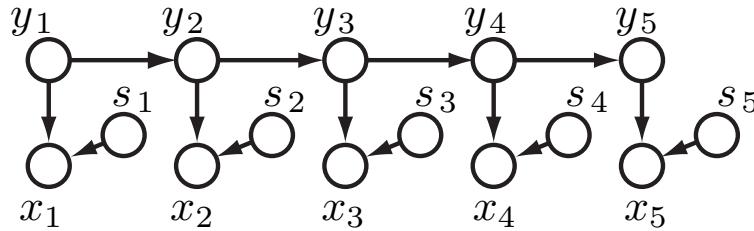


Figure 8.9: An HMM with explicit mention of the random variable S_t that selects the random function $f_{S_t}(y_t)$ at each time step that maps from state y_t to observation x_t .

Suppose we have a set of $|D_S| * T$ functions $f_{i,t}$, where $f_{i,t} : D_X \rightarrow D_Y$. That is, for every i , each function $f_{i,t}$ maps a single scalar (state) random variable Y_t to a single random variable X_t . Suppose also we have T independent random variables S_t . Then given a Markov chain Y , we can define each element of the sequence X as a function of the corresponding element in the sequence Y . That is, for all t , we produce $X_t = f_{S_t}(Y_t)$. The generative process is as follows: we choose a sequence $y = (y_1, \dots, y_T)$, then choose a sequence of independent variables $s = (s_1, s_2, \dots, s_T)$, and then finally choose each x_t as $x_t = f_{s_t}(y_t)$. This corresponds to Figure 8.9, where the graph explicitly represents the variables $\{S_t : t\}$. Here, we can think of x_t as a deterministic function of both y_t and s_t , and all of the randomness relating x and y is captured by the variables s .

Revisiting the balls and urn example from §8.4.1, we see that the sequence $y = (y_1, \dots, y_T)$ is the choice of T urns based on a Markov chain. Then, s_t is used to choose which ball is chosen from each urn. That is, suppose that $y_t = 4$ (we chose the fourth urn at time t). In such case, if $s_t = 2$ this might correspond to $f_{2,t}(4) = \text{yellow}$. Hence, x_t is a draw from the urn corresponding to urn y_t and the ball within that urn, determined by s_t .

Yet another way of viewing this is as follows. Let Y_1, Y_2, \dots, Y_n be a stationary Markov chain. Let X_1, X_2, \dots, X_N be a random function of this Markov chain. I.e.,

$$X_t = \left\{ \begin{array}{ll} f_1(Y_t) & \text{with probability } p_1(Y_t) \\ f_2(Y_t) & \text{with probability } p_2(Y_t) \\ \vdots & \\ f_m(Y_t) & \text{with probability } p_m(Y_t) \end{array} \right\} = f_{S(Y_t)}(Y_t) \quad (8.26)$$

where $S \in \{1, 2, \dots, s\}$ itself is a random variable with $p(S = i) = p_i$. Note that S might itself also be dependent on Y_t . One can see the double stochasticity in that $f_S(Y_t)$ involves two random variables, S and Y_t in producing the resultant random variable X_t .

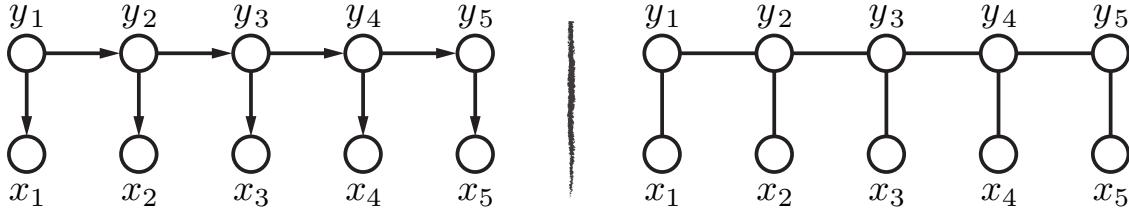


Figure 8.10: A HMM can be viewed as either a Bayesian network (left) or as a Markov random field (right). They both express the same factorization constraints, so in this case both the BN and the MRF indicate the same family of models. Another way to see this is that the moralization process applied to the left model adds no additional edges since it has no V-structures. Moreover, once moralized the model is already triangulated, decomposable, and identical to the right model. Hence, the two graphical models correspond to the same family of distributions.

8.4.3 HMMs as exponential (in T) sized mixture distribution

We can also view HMMs as an exponentially large mixture distribution. That is, the random variable X is distributed according to a mixture of a set of distributions.

A mixture distribution over X can be described simply as

$$p(x) = \sum_c p(x, c) = \sum_c p(x|c)p(c) \quad (8.27)$$

where c is a mixture variable, so that $p(c)$ is the mixture weight for mixture component $p(x|c)$. The Bayesian network for this mixture is shown in Figure 8.11. In an inclusionary sense, Figure 8.11 therefore is accurate for an HMM — if X is vector valued and C is vector valued, then an HMM does not violate the properties stated by this figure. In other words, when x and c are vector valued with length T , then the graph's family “covers” HMMs. On the other hand, very little violates the properties of this figure and moreover the figure does not at all characterize the properties of HMMs that makes them interesting (and computationally tractable). The figure's specificity is quite poor, and the peculiarities of HMMs that make them both useful and computationally efficient in practice are not at all expressed by this graph.

Ignoring this problem for the moment, let's proceed and look at multiple mixtures at the same time. Suppose we have a set of random variable pairs $(X_1, Y_1), (X_2, Y_2), \dots, (X_T, Y_T)$ each of which is an individual independent mixture distribution, with mixture variable Y_t . Also, assume that all of the Y_t variables are (marginally) independent, meaning that $Y_t \perp\!\!\!\perp Y_\tau$ for $t \neq \tau$ (this is not a required assumption in an HMM, but it does not violate any HMM assumptions, so it would still be considered an HMM, albeit a degenerate



Figure 8.11: A simple mixture distribution. The joint distribution $p(x, c)$ is such that $p(x) = \sum_c p(x|c)p(c)$ where $p(c)$ is the weight for mixture c and $p(x|c)$ is the c^{th} mixture component.

one). With such assumptions, we get the following collection of mixtures:

$$p(x, y) = p(x_1, \dots, x_T, y_1, \dots, y_T) \quad (8.28)$$

$$= \prod_{t=1}^T p(x_t, y_t) \quad (8.29)$$

$$= \prod_{t=1}^T p(x_t|y_t)p(y_t) \quad (8.30)$$

In fact, Figure 8.1 shows exactly this in plate notation, where the set of random variables is repeated T times. On the right of the figure shows the expanded case when $T = 5$. Since in this collection of mixtures we have that $Y_i \perp\!\!\!\perp Y_j$ for all $i \neq j$, the mixtures are exchangeable (the temporal order does not matter to the score as any permutation would be scored identically).

To see how HMMs can be viewed as an exponentially large mixture (in T) using a vector-valued mixture variable, consider:

$$p(x) = \sum_y p(x, y) \quad (8.31)$$

$$= \sum_{y_1, y_2, \dots, y_T} p(x_1, x_2, \dots, x_T, y_1, y_2, \dots, y_T) \quad (8.32)$$

$$= \sum_{y_1, y_2, \dots, y_T} p(x_1, x_2, \dots, x_T|y_1, y_2, \dots, y_T)p(y_1, y_2, \dots, y_T) \quad (8.33)$$

$$= \sum_{y_1, y_2, \dots, y_T} p(x_1, x_2, \dots, x_T|y_1, y_2, \dots, y_T)c_{y_1, y_2, \dots, y_T} \quad (8.34)$$

where c_{y_1, y_2, \dots, y_T} is a mixture weight. Thus, we can view $p(x)$ as having a mixture distribution over vectors, with an exponential number of mixture weights $p(y_1, y_2, \dots, y_T)$ and also mixture components $p(x_1, x_2, \dots, x_T|y_1, y_2, \dots, y_T)$ — e.g., if $|D_{Y_t}| = r$ for all t , then there would be r^T mixture weights and mixture components.

Hence, it appears that this model (in its full generality) would require exponential computation. To get this into an HMM form, we make further assumptions. First, lets assume that y forms a first-order Markov chain (lets call this assumption I), so that

$$p(y_1, y_2, \dots, y_T) = \prod_t p(y_t|y_{t-1}). \quad (8.35)$$

Second, lets assume that X_t is conditionally independent of all else given Y_t (calling this assumption II), giving

$$p(x_1, x_2, \dots, x_T|y_1, y_2, \dots, y_T) = \prod_{t=1}^T p(x_t|x_1, \dots, x_{t-1}, y_1, y_2, \dots, y_T) \quad (8.36)$$

$$= \prod_{t=1}^T p(x_t|y_t) \quad (8.37)$$

and now Equation 8.34 can be simplified to

$$p(x) = \sum_{y_1, y_2, \dots, y_T} \prod_t p(y_t|y_{t-1})p(x_t|y_t), \quad (8.38)$$

which is the classic expression for an HMM. Now there are still an exponential number of terms in the sum, but because assumption I and II made above, performing this sum can be done using dynamic programming without an exponential number of steps. It follows from the distributive law $a(b + c) = ab + ac$. Using this property, we may simplify the HMM as follows (we assume that x is constant in the computation):

$$p(x) = \sum_{y_1, y_2, \dots, y_T} \prod_t p(y_t | y_{t-1}) p(x_t | y_t) \quad (8.39)$$

$$= \sum_{y_1, y_2, \dots, y_{T-1}} \prod_{t=1}^{T-1} p(y_t | y_{t-1}) p(x_t | y_t) \left(\sum_{y_T} p(y_T | y_{T-1}) p(x_T | y_T) \right) \quad (8.40)$$

$$= \sum_{y_1, y_2, \dots, y_{T-1}} \prod_{t=1}^{T-1} p(y_t | y_{t-1}) p(x_t | y_t) \phi(y_{T-1}) \quad (8.41)$$

$$= \sum_{y_1, y_2, \dots, y_{T-2}} \prod_{t=1}^{T-2} p(y_t | y_{t-1}) p(x_t | y_t) \left(\sum_{y_{T-1}} p(y_{T-1} | y_{T-2}) p(x_{T-1} | y_{T-1}) \phi(y_{T-1}) \right) \quad (8.42)$$

$$= \sum_{y_1, y_2, \dots, y_{T-2}} \prod_{t=1}^{T-2} p(y_t | y_{t-1}) p(x_t | y_t) \phi(y_{T-2}) \quad (8.43)$$

$$= \dots \quad (8.44)$$

$$= \sum_{y_1, y_2} \prod_{t=1}^2 p(y_t | y_{t-1}) p(x_t | y_t) \phi(y_2) \quad (8.45)$$

$$= \sum_{y_1} p(y_1) p(x_1 | y_1) \left(\sum_{y_2} p(y_2 | y_1) p(x_2 | y_2) \phi(y_2) \right) \quad (8.46)$$

$$= \sum_{y_1} p(y_1) p(x_1 | y_1) \phi(y_1) = p(x) \quad (8.47)$$

In each step, we are sending in the highest value T summation into the factorization as far to the right as possible, performing the summation, and then renaming the result of that summation with a new function ϕ that involves only the hidden variable remaining to be summed over. Each time we do this, the computation is only r^2 and since we do this T times, we see that we can compute the exponential mixture model score in $O(Tr^2)$ time rather than $O(r^T)$ time. Indeed, the above calculation is often called the “beta” recursion for HMMs, a recursive calculation that often uses a variable named β . If we started summing first from the lowest value of t rather than the highest, we would get the standard “alpha” recursion equation for HMM. More generally, this is just an instance of the elimination algorithm (§2.2.1) run on an HMM (also see [398]). It is also an instance of the dynamic programming algorithm (cf. §8.4.6.1).

8.4.4 HMMs as Stochastic Finite State Automata

The field of Finite state automata (FSA) is a big area, so we are not able to properly cover them in this subsection. The interested reader could *start* by considering [226, 198, 129] and the many references contained therein. Also, a nice overview is given in [440]. We do give a bit of background, however, so as to aid in our understanding of HMMs and ultimately dynamic graphical models.

An FSA consists of a machine that generates a set of strings, called a language. The machine consists of a finite number of states, where there is one start state and at least one termination state, and a set of allowable transitions between states. Each transition moreover has a substring label that is “consumed” whenever one transitions between the two states incident to the transition. The set of all sequences of substrings that may

validly be consumed by the FSA corresponds to the FSA's language. Given a sequence of symbols, the task of deciding if the sequence lies within the FSA's language or not is called parsing and a given set of states that allows a sequence to be accepted by the FSA is called a parse.

There are a number of different types of automata including deterministic and stochastic — we start first with the deterministic.

8.4.4.1 Deterministic FSAs

Definition 118 (Deterministic Finite State Automata). A *deterministic finite state automata* consists of the following:

1. A finite set of states, we denote \mathcal{Q} as these states. We label the states as $\{q_0, q_1, \dots, q_n\} = \mathcal{Q}$ where $n = |\mathcal{Q}| < \infty$ is the number of states.
2. A finite set of input symbols, \mathcal{S}
3. A transition function $\delta : \mathcal{Q} \times \mathcal{S} \rightarrow \mathcal{Q}$ that maps from a pair consisting of a state and an input symbol and returns a new state. The set of possible transitions is a subset of the Cartesian product of the states with itself $\mathcal{Q} \times \mathcal{Q}$. We define these set of possible transitions as $\mathcal{A} \triangleq \{(q, r) : q, r \in \mathcal{Q} \text{ and } \exists s \in \mathcal{S} \text{ s.t. } \delta(q, s) = r\}$. Hence, we see that $\mathcal{A} \subseteq \mathcal{Q} \times \mathcal{Q}$. Moreover, for each arc $a \in \mathcal{A}$, there is an associate string $s \in \mathcal{S}$. We can think of \mathcal{S} as a function from transitions to strings so that for $(q, r) = a \in \mathcal{A}$, $\mathcal{S}(a) = s$ where $\delta(q, s) = r$.

Note also that for each state $q \in \mathcal{Q}$ and each $s \in \mathcal{S}$ there is at most following state $\delta(q, s)$ — this is what makes the FSA deterministic.

4. A pre-designated start state $q_0 \in \mathcal{Q}$, where the FSA always begins.
5. A set of termination or final states $\mathcal{F} \subseteq \mathcal{Q}$.

Hence, we can think of an FSA as the tuple $DFSA = (\mathcal{Q}, \mathcal{S}, \delta, q_0, \mathcal{F})$.

It is perhaps easiest to depict small FSAs using a graph where the nodes of the graph correspond to states and the arcs correspond to transitions (recall our terminology in §0.7). It is critical to ones understanding that a graph for an FSA is **not** a graphical model. An FSA graph does not encode a set of random variables and their factorization properties, as do graphical models (cf. §2.2.1). Graphical representations of FSAs are entirely different than graphical encodings of factorization properties that define families of probability distributions (I've seen many a first-year graduate student confuse these two).

A graphical representation of an FSA has a set of nodes corresponding to the states \mathcal{Q} , and a set of arcs corresponding to the set of possible transitions \mathcal{A} . Hence, we can define a graph $\mathcal{G} = (\mathcal{Q}, \mathcal{A})$ that faithfully depicts an FSA. Lets do an example.

Our first example will be quite simple, and will have $n = |\mathcal{Q}| = 3$ states and a binary string alphabet, so $\mathcal{S} = \{0, 1\}$. The FSA will be one that consumes all binary strings that contain a substring 01 somewhere within it ([198]) and is shown in Figure 8.12. Upon starting at q_0 the FSA can generate either a 1 or a 0, and if it generates a 1 it stays in state q_0 (and could stay there for an unbounded amount of time). When and if it generates a 0 it goes to q_1 where if it keeps generating a 0 it stays in q_1 or goes to q_2 if it generates a 1. Hence, the only way to reach q_2 which is the termination state is if, along the way, it generates a 01 substring which is either the 0 from the arc (q_0, q_1) or from the arc (q_1, q_1) and the one from (q_1, q_2) . Once in q_2 the machine

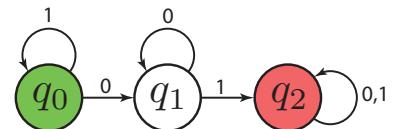


Figure 8.12: Simple 3-state deterministic FSA. The start state is shaded green and the final end state is shaded red so $\mathcal{F} = \{q_2\}$.

may terminate, or may generate any binary string. Hence, upon seeing a binary string $b = (b_0 b_1 \dots b_T)$ with $b_i \in \{0, 1\}$, the string b might either be consumed (parsed) by the FSA or not.

Note that we can also view the FSA as a producer (or generator) of strings as well as a consumer of strings. That is, if at each state we choose arbitrarily one of the strings that lead it to the next state, then the FSA produces such a string. For example, in Figure 8.12, once we reach q_1 we can choose, arbitrarily, to generate either a 0 or a 1, and depending on what we choose will determine which next state we go to (q_1 or q_2 respectively).

Of course, deterministic FSA's can be much more complicated than this. Here's another example, again shamelessly stolen from [198]. Here, we want to produce produce a deterministic FSA that can consume (or produce) strings from the following language of binary strings:

$$\mathcal{L} = \{b : b \text{ has both an even number of 0's and an even number of 1's}\} \quad (8.48)$$

That is, if $b \in \mathcal{L}$ then somehow, the FSA must reject the string if it has either an odd number of 0's or 1's. This is shown in Figure 8.13. The start state q_0 is also the termination state, so we shade this node both green and red to indicate this. State q_0 means that the string so far consumed/produced (which might just be the empty string) is in \mathcal{L} . State q_1 means that have an odd number of 1's, state q_3 means we have an odd number of both 1's and 0's, and state q_2 means we have an odd number of 0's.

Of course, the set of strings \mathcal{S} need not be binary. We might use an FSA to produce the set of strings in a given language, and that language might be a programming language such as C or C++ (in which case \mathcal{S} is the set of possible tokens in the language), or even a human language (where \mathcal{S} would be the set of words or types in the language). In this latter case, we could use an FSA to attempt to produce strings in a real language, or pronunciations of a given word.

8.4.4.2 Non-deterministic FSAs

A non-deterministic FSA (NFSAs) is different from a deterministic FSA in that there might be a set of states that might follow a given state for a given input symbol. That is, the state transition function δ outputs a set of states rather than a single state. Formally, the definition follows:

Definition 119 (Non-deterministic Finite State Automata). A *non-deterministic finite state automata* consists of the following:

1. A finite set of states, we denote \mathcal{Q} as these states. We label the states as $\{q_0, q_1, \dots, q_n\} = \mathcal{Q}$ where $n = |\mathcal{Q}| < \infty$ is the number of states.
2. A finite set of input symbols, \mathcal{S}
3. A transition function $\delta : \mathcal{Q} \times \mathcal{S} \rightarrow 2^{\mathcal{Q}}$ that maps from a pair consisting of a state and an input symbol and returns a new set of possible states. The set of possible transitions is a subset of the Cartesian product of the states with itself $\mathcal{Q} \times \mathcal{Q}$. We define these set of possible transitions as $\mathcal{A} \triangleq \{(q, r) : q, r \in \mathcal{Q} \text{ and } \exists s \in \mathcal{S} \text{ s.t. } \delta(q, s) = r\}$. Hence, we see that $\mathcal{A} \subseteq \mathcal{Q} \times \mathcal{Q}$. Moreover, for each arc $a \in \mathcal{A}$, there is an associate string $s \in \mathcal{S}$. We can think of \mathcal{S} as a function from transitions to strings so that for $(q, r) = a \in \mathcal{A}$, $\mathcal{S}(a) = s$ where $\delta(q, s) = r$.

Note also that for each state $q \in \mathcal{Q}$ and each $s \in \mathcal{S}$ the transition function $\delta(q, s) \subseteq \mathcal{Q}$ is a state — this is what makes the FSA non-deterministic.

4. A pre-designated start state $q_0 \in \mathcal{Q}$, where the FSA always begins.

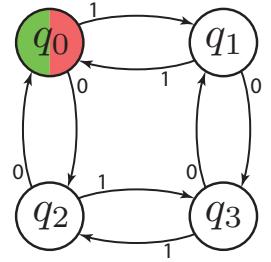


Figure 8.13



Figure 8.14: Left: a DFSA for the sheep language. At each state, any symbol from \mathcal{S} takes you to only one possible next state. Right: an Equivalent NFSA for the sheep language. Here, when in q_2 the symbol “a” $\in \mathcal{S}$ means one could be in one of two next states. I.e., $\delta(q_2, “a”) = \{q_2, q_3\}$. Taken from [226]

5. A set of termination or final states $\mathcal{F} \subseteq \mathcal{Q}$.

Hence, we can think of an non-deterministic FSA also as the tuple $\text{NFSA} = (\mathcal{Q}, \mathcal{S}, \delta, q_0, \mathcal{F})$.

The state transition function in an NFSA being a set means that upon seeing a symbol, the NFSA is allowed to go next to any of the states within the set. It can choose so arbitrarily. When seen as a generator (consumer), the language strings that may be generated (consumed) by the NFSA consists of all strings that can be generated (consumed) by any arbitrary choice of next state within the set returned by the transition function.

Now with this extra flexibility, one might think that a NFSA has more power (meaning the set of language strings is larger) than a DFSA, but it turns out not to be the case. In fact, any finite-state NFSA can be converted into an equivalent DFSA possibly with more states. Often, the number of states is the same, but in the worst of cases, a given language can be achieved with an n state NFSA but only with a 2^n state DFSA [198].

Why would we want a non-deterministic FSA over a deterministic FSA? There are several reasons. First, a language can sometimes be more concisely represented using a NFSA than an FSA, as implied above. In some cases, this can lead to more efficient representation of a language. Secondly, since NFSAs can be more concise, they can seem more natural for many applications and can actually be easier to understand (and design). Third, NFSAs immediately generalize to stochastic FSAs (as we see below).

Give example of DFSA for the pronunciations of the word “and.”

8.4.4.3 Epsilon Arcs

FSAs (either deterministic or non-deterministic) can contain what are called epsilon arcs (or ε -arcs or epsilon moves). These arcs are given a special label (unsurprisingly ε) and what this means is that when the transition between the states incident to the arc occurs, no symbol from the symbol set is consumed (or produced in the production interpretation of FSAs). Another way to view this is to extend the symbols set with a special symbol ϵ to $\mathcal{S} \leftarrow \mathcal{S} \cup \{\epsilon\}$, and then we can interpret ϵ however we like (e.g., as being out of the language). Epsilon arcs can make it quite a bit easier to specify a given language using FSAs since it is possible to directly jump from any one state to another without needing to produce a symbol. On the other hand, epsilon arcs do not increase the flexibility of FSAs — one can take an FSA (either deterministic or non-deterministic) and transform it to one that does not use epsilon arcs, albeit with some potential loss of naturalness and some potential increase in complexity of the model.

Note that in some sense, a deterministic FSA with epsilon arcs can be seen as a non-deterministic one. The reason is, if we have arrived at a state with incident outgoing arcs that are labeled with both non-epsilon arcs and epsilon-arcs, then one can view this as being able to decide non-deterministically if one should next generate one of the non-epsilon symbols along the outgoing arc, or if one instead should jump to one of the states at the destination side of the incident epsilon arcs and proceed from there. The language generated by the FSA, in this case, includes strings generated via both cases. Hence, epsilon arcs may add considerable flexibility to a deterministic FSA.

8.4.4.4 Transducers FSAs

So far, we've seen FSAs as either being consumers of strings of a given language or producers of strings of a given language. We can also view them as both, meaning that they consume strings of one language and produce strings of another language. This is called a transducer.⁴ With a transducer, we have both an input and an output string alphabet. We give here the corresponding definition (in the non-deterministic case).

Definition 120 (Non-deterministic Finite State Machine/Transducer). *A non-deterministic finite state transducer consists of the following:*

1. A finite set of states \mathcal{Q} .
2. A finite set of input symbols \mathcal{S} (the input alphabet from which input strings are constructed).
3. A finite set of output symbols \mathcal{O} (the output alphabet from which output strings are constructed).
4. A transition function $\delta : \mathcal{Q} \times \mathcal{S} \rightarrow 2^{\mathcal{O}}$ (or $\delta : \mathcal{Q} \times \mathcal{S} \rightarrow \mathcal{Q}$ in the deterministic case).
5. An output function $b : \mathcal{Q} \times \mathcal{S} \rightarrow \mathcal{O}$.
6. A pre-designated start state $q_0 \in \mathcal{Q}$, where the FSA always begins.
7. A set of termination or final states $\mathcal{F} \subseteq \mathcal{Q}$.

The definition of the set of arcs \mathcal{A} is identical to Definition 119.

The only difference between Definition 119 and Definition 120 is that the transducer has a set of output symbols and an output function so that for each arc $a \in \mathcal{A}$ there is a single associated output symbol. That is, each output symbol $o \in \mathcal{O}$ is produced based on the current state $q \in \mathcal{Q}$ and the current input symbol $s \in \mathcal{S}$ via the function $o(q, s)$. We can think of this is that all arcs in the transducer have two labels, that of the input symbol that induces the set of possible output arcs of a given state, and also the corresponding output symbol generated when that arc is taken (Figure 8.15 shows a transducer that transforms from the language of all binary strings containing a substring 01 somewhere within it to the language of all binary strings containing a string 10 somewhere within, in fact this transducer just inverts the 0's and 1's but it only accepts strings from the input language rather than all binary strings). Moreover, we might as well at this point note that it is possible to have more than one designated start states — this comes with no gain or loss of generality.

A FST can be seen as mapping from strings in one language (the input language) to strings in another language (the output language) and we can study the types of functions that may be expressed by such machines. Indeed, finite state transducers are very powerful and have been used very successfully in a number of ways in the field of speech recognition and statistical machine translation [308, 307, 306] where the goal is to transform one language into another — in the case of speech recognition, we are translating from a sequence of acoustic feature vectors to a sequence of words, and in the case of machine translation we are transforming from strings of words in one language to strings of words in another language.

Note that a FST is a strict generalization of an FSA. In fact, one can recover an FSA from an FST by ignoring (or removing) the output associated with the FST thus achieving an FSA that “consumes” the input symbols.

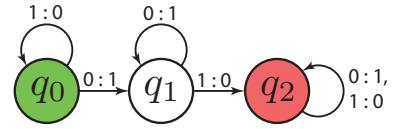


Figure 8.15

⁴finite state transducers were also discussed in Shannon's 1948 paper [393].

8.4.4.5 Weighted or Stochastic FSAs

A non-deterministic FSA/FST is such that $\delta(q, s) \subseteq \mathcal{Q}$ meaning that in state q and when considering string s , the set of next states $\delta(q, s)$ can be more than one. The choice of which is arbitrary — when seen as an input language acceptor, then any next state which ultimately is able to accept the complete language string may be chosen. These decisions being arbitrary is what makes the machine non-deterministic.

It is possible, however, to express preferences on each of these next states. The preferences can be specified using either a non-negative weight, or equivalently a probability. When we consider the transition function as a graph $G = (\mathcal{Q}, \mathcal{A})$ with nodes \mathcal{Q} and arcs \mathcal{A} , then for any $q, r \in \mathcal{Q}$ such that $(q, r) \in \mathcal{A}$ we can associate a non-negative weight $w : \mathcal{A} \rightarrow \mathbb{R}_+$. The weights might correspond to a “cost” or a “distance” associated with taking that particular arc, and when an FSA/FST is at a given state and considering a next state, it might choose it based on the one with the least cost. Alternatively, these weights might be normalized to produce conditional probabilities. One such normalization might be to provide a probability function on each arc $a = (q, r)$ as follows:

$$\Pr(a) = \Pr(r|q) = \frac{e^{-w(q,r)}}{\sum_{q' : (q,q') \in \mathcal{A}} e^{-w(q,q')}} \quad (8.49)$$

With this, we immediately have a probabilistic interpretation of an FSA, something often called a stochastic finite state automata a (stochastic finite state transducer) or just SFSA (SFST).

With an SFSA, then any string in the language has an associated probability, and when we sum together the probabilities of all language strings we get unity. Hence, the SFSA can be seen as a probability distribution on strings, but one where a concise representation is used to express this distribution (the alternative would be to list the potentially exponentially large set of strings in the language and associate each with a probability, say, in a large table). Similarly, an SFST has an associated probability with each transformation from one string to the other. We can see this as a conditional distribution $\Pr(\text{output string}|\text{input string})$ where for each input string we have a (concisely and algorithmically specified) distribution over all possible output strings. For this reason, SFSTs are particularly useful for the speech and language processing tasks mentioned above.

Note that weighted or stochastic FSAs can be generalized even further by allowing the weights to be drawn from any set that is also a (commutative) semiring.

Definition 121. A commutative semiring is a set \mathbf{K} with two binary operators \oplus and \otimes having three axioms, for all $a, b, c \in \mathbf{K}$.

S1: \oplus is commutative $(a \oplus b) = (b \oplus a)$ and associative $(a \oplus b) \oplus c = a \oplus (b \oplus c)$, and \exists a unique additive identity called $\bar{0} \in \mathbf{K}$ such that $k \oplus \bar{0} = k$ for all $k \in \mathbf{K}$. I.e., (\mathbf{K}, \oplus) is a commutative monoid.

S2: \otimes is also associative, commutative, and \exists a unique multiplicative identity called $\bar{1} \in \mathbf{K}$ s.t. $k \otimes \bar{1} = k$ for all $k \in \mathbf{K}$ $((\mathbf{K}, \otimes)$ is also a commutative monoid).

S3: the distributive law holds: $(a \otimes b) \oplus (a \otimes c) = a \otimes (b \oplus c)$ for all $a, b, c \in \mathbf{K}$.

There are in fact many possible semi-rings — Table 8.1 lists some of them.

Interestingly, here an additive inverse (a value $k^- \in \mathbf{K}$ such that $k \oplus k^- = \bar{0}$) need not exist. If additive inverse exists, then we get a commutative ring (the “semi” in “semiring” is since we need not have additive inverse). Note that in algebra texts, a ring often does not require multiplicative identity, but we (and do many in the speech, language, and graphical model communities) assume it exists here.

Also note, the above definition does not mention that the additive identity $\bar{0}$ is also multiplicatively closed — that is, that $\bar{0} \cdot k = \bar{0}$. This follows from above properties since $k \otimes k = k \otimes (k \oplus \bar{0}) = k \otimes k + k \otimes \bar{0}$ so that $k \otimes \bar{0}$ must also be an additive identity. Since it is presumed unique, this means that $k \otimes \bar{0} = \bar{0}$.

With this we can make the next definition.

	K	“(⊕, 0)”	“(⊗, 1)”	short name
1	A	(+, 0)	(·, 1)	semiring
2	$A[x]$	(+, 0)	(·, 1)	polynomial
3	$A[x, y, \dots]$	(+, 0)	(·, 1)	polynomial
4	$[0, \infty)$	(+, 0)	(·, 1)	sum-product
5	$(0, \infty]$	(min, ∞)	(·, 1)	min-product
6	$[0, \infty)$	(max, 0)	(·, 1)	max-product
7	$[0, \infty)$	(kmax, 0)	(·, 1)	k -max-product
8	$[-\infty, \infty]$	(min, ∞)	(+, 0)	min-sum/tropical
9	$[-\infty, \infty)$	(max, −∞)	(+, 0)	max-sum
10	$[-\infty, \infty]$	(LOG-ADD, +∞)	(·, 0)	log
11	{0, 1}	(OR, 0)	(AND, 1)	Boolean
12	2^S	(\cup , \emptyset)	(\cap , S')	Set
13	Λ	(\vee , 0)	(\wedge , 1)	Lattice
14	Λ	(\wedge , 1)	(\vee , 0)	Lattice

Table 8.1: There are many semi-rings, here are a few of them. Here, A denotes arbitrary commutative semiring, S is arbitrary finite set, Λ is an arbitrary distributive lattice. The LOG-ADD operator for operands a and b is $-\log(\exp(-a) + \exp(-b))$. We discuss the k -max-product case in §.

Definition 122 (semiring weighted transducer (after [305])). A semiring weighted transducer over a semiring $(K, \oplus, \otimes, \bar{0}, \bar{1})$ is an 8-tuple $(S, O, Q, I, F, A, \lambda, \rho)$ where S is a finite input alphabet, O is a finite output alphabet, Q is a finite set of states, $I \subseteq Q$ is a set of initial states, $F \subseteq Q$ is a set of final states, A is a finite multi-set of arcs/transitions (more on this below), $\lambda : I \rightarrow S$ is an initial weight function with weights according to the semiring K , $\rho : F \rightarrow S$ is a final weight function. The arcs $a \in A$ are themselves elements of the 5-tuple of the form $Q \times (S \cup \{\varepsilon\}) \times (O \cup \{\varepsilon\}) \times K \times Q$. Each arc $(q, r) = a \in A$ (between nodes q and r) is labeled with: 1) an input symbol drawn from $S \cup \{\varepsilon\}$; 2) an output symbol drawn from $(O \cup \{\varepsilon\})$; and 3) a weight drawn from the semiring symbol set K .

Definition 122 is quite general. In the simplest case specific, the semiring can be seen to be either a non-negative weight, or a probability but many other semirings also exist and can be quite useful (which is why people like semirings), e.g., the n -best semiring. Definition 122 also generalizes the state transition function $\delta : Q \times S \rightarrow 2^Q$ and the output function $b : Q \times S \rightarrow O$ we saw in earlier definitions to the arc multiset A .

Note that there is no gain in generality by allowing two arcs that are *exactly* the same since in such case the weights may be added using \oplus and all such arcs replaced with a single arc with the “sum” of weights. On the other hand, it is quite important to allow more than one arc between the same two states. For example, between states q and r we might have an arc with different input and/or output symbols.

There are many other important topics regarding FSAs that we will not be discussing here. This includes minimization, determinization, and composition. The reader should indeed look at the references [226, 198, 440, 308, 307, 306].

8.4.4.6 Mealy and Moore FSAs

One distinction that we will make here, since it has consequences when understanding the relationship between FSAs and graphical models, is the difference between Mealy FSAs and Moore FSAs.

The FSAs that we so far have been discussing in are really Mealy FSAs.⁵ What characterizes them is

⁵in 1955, G.H. Mealy first introduced them although it can be argued that Claude Shannon also introduced them in 1948 [393].

that strings in the language are associated with arcs in the machine, or equivalently the transitions between states. When a transition between two states is taken, then a string is consumed (or produced). Recall above, that we can view a non-deterministic FSA also as the tuple $\text{NFSA} = (\mathcal{Q}, \mathcal{S}, \delta, q_0, \mathcal{F})$. This definition is of the Mealy FSA form since the transition function $\delta : \mathcal{Q} \times \mathcal{S} \rightarrow 2^{\mathcal{Q}}$ yields a set of arcs \mathcal{A} each of which has an associated $s \in \mathcal{S}$.

An alternative form of FSA is what is known as a Moore machine.⁶ In a Moore machine, a string is associated with each state of the machine (equivalently, node in the corresponding transition graph) rather than each arc. That is, a Moore machine generates (or consumes) a symbol $s \in S$ upon each arrival at a state rather than each transition between states.

We can more formally define a Moore FSA as follows:

Definition 123 (Moore Non-deterministic Finite State Automata). *A Moore Non-deterministic finite state automata consists of the following:*

1. A finite set of states \mathcal{Q} .
2. A finite set of input symbols \mathcal{S} .
3. A finite set of output symbols \mathcal{O} .
4. A transition function $\delta : \mathcal{Q} \times \mathcal{S} \rightarrow 2^{\mathcal{Q}}$.
5. An output function $b : \mathcal{Q} \rightarrow \mathcal{O}$.
6. A pre-designated start state $q_0 \in \mathcal{Q}$, where the FSA always begins.
7. A set of termination or final states $\mathcal{F} \subseteq \mathcal{Q}$.

The only difference between a Mealy machine (Definition 120) and a More machine (Definition 123) is that the Moore machine's output function is based only on the current state rather than on both the current state and the current input symbol.

As stated, a Moore machine has only one possible output for each state — hence, we can think of the output being a function only of the current state.

A Mealy machine has only one possible output for each state-input pair — we can think of this as follows: given a current state, the input determines both the next state and the current output. Equivalently, given a current state, and a next state (which is determined by the input), there is a corresponding output. Hence, we can think of a Mealy machine as one where the output is a function of the transition between two states.

Mealy and Moore “machines” were originally developed to characterize computational devices via combinatorial logic and clocked sequential circuits — after all, every computer (even an iPad) can be seen as a FSA, albeit with quite a large number of states and rich input/output alphabets. This view is shown in Figure 8.16.

A Moore machine of course also has a stochastic variant where each possible next state has an associated probability. That is, for each of the $r \in \delta(q, s)$ we might associate a weight $w(q, r)$, probability $\Pr(r|q)$, or element of a semiring.

Mealy and Moore machine are the same in that neither of them is able to express a language (or a transduction) that the other is unable to express. Hence, a Mealy machine can be transformed into a Moore machine, and vice versa (in either the deterministic, non-deterministic, stochastic, or weighted cases). This transformation can be done in polynomial time and in fact can be done without more than a quadratic (in n) number of additional states. Hence, they are mathematically equivalent.

⁶in 1956 F. Moore studied them, and this was apparently independent of Mealy's work at roughly the same time

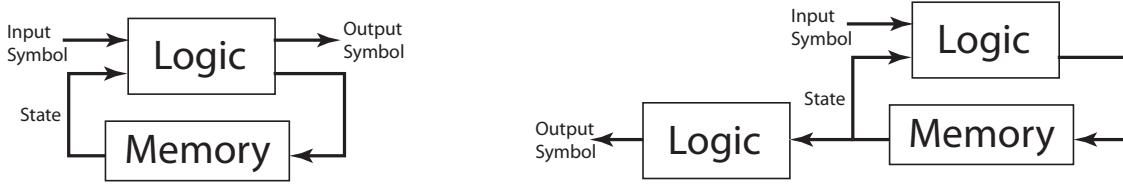


Figure 8.16: Left: a Mealy FST seen as typically drawn when viewed as a computational device. Right: a Moore FST drawn similarly. With this depiction, it perhaps makes more sense to consider only the deterministic case.

But just because they are mathematically equivalent does not mean that the choice between using one vs. the other in practice is without consequence. Indeed, some languages can be more efficient Mealy machines.

Consider, as an example, a machine whose job it is to accept all ASCII strings that have matching parentheses but where parentheses may be nested at most one deep (i.e., once we have seen a left-parenthesis, we cannot see another left-parenthesis until after we have seen a matching right-parenthesis). Some length 10 example strings in and out of the language are given in Figure 8.17.

Lets first consider doing this with a Moore machine, and we start by doing this with $|\Omega| = 4$ states. State 1 corresponds to a background state (meaning we are outside of a pair of parentheses) and we stay there while there are none. State 2 corresponds to the emission of a “(“ character. State 3 corresponds to being inside a match, and we are waiting for a “)” character. And lastly, state 4 corresponds to the emission of a “)” character.

With this state characterization, we can generate the language using the following state transition graph between the four states: Note that normally the computational complexity associated with a Markov chain grows as $O(n^2)$ where n is the number of states. In this case, however, due to the large number of missing transitions, the complexity is not as bad as 4^2 . We note that state 1 goes to 1 or 2; state 2 goes to 3 always; state 3 goes to 3 or 4; and state 4 goes to 1 always. Normally, the complexity is 4×4 (for each state, 4 next possibilities) and this comes from the alpha computation

$$\alpha_t(q_t) = o_t(q_t) \sum_{q_{t-1}} \alpha_{t-1}(q_{t-1}) p(q_t | q_{t-1}) \quad (8.50)$$

where for each of the four q_t values, we do a sum over four terms.

8.4.4.7 Push Strategy for HMM Inference

The above α recursion can be inefficient when there are many zeros in the array at each time step. The issue is that the sum over r might be summing over zeros (sometimes mostly). This “pull” form of message passing is ignorant of any sparsity that might exist at the previous time step.

An alternate form is what could be called a “push” strategy, where for each current possible state we push out only to the next states with non-zero possible next probability. That is, rather than using “pull” previous states to the current, we push current messages to next. The strategy can best be described algorithmically:

In-Language Strings	Out-of-Language Strings
(.)	(())
... ()	(. (.)
... . (. . . .) .	(. (. . .) . . .)
(. . . .) () . .	((((((((((
() . . () . . ()	() ())) () ()

Figure 8.17: FSA Depth one parenthesis Matching

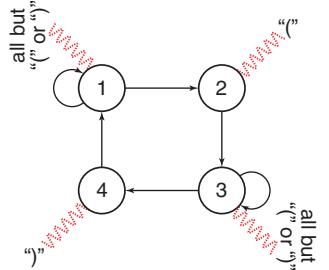


Figure 8.18: Moore machine parenthesis matching state transition diagram.

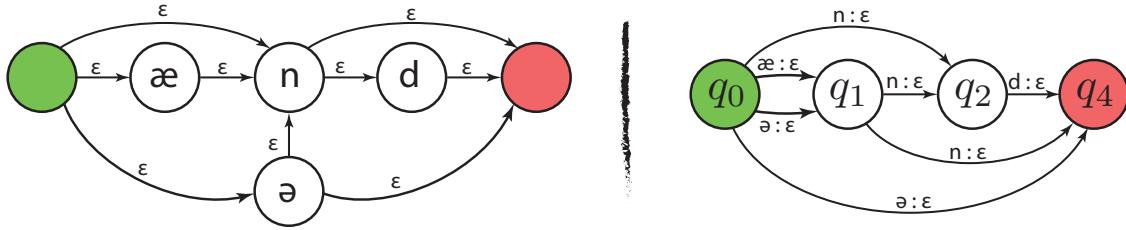


Figure 8.20: Pronunciation model for the word “and” — Left: Moore machine, Right: Mealy machine. The output label on each arcs is shown as epsilon since these machines are simply acceptors of all the pronunciations of “and.”

```

1 for  $r = 1 \dots N$  do
2   if  $\alpha_{t-1}(r) > 0$  then
3     for  $q \in \{q : p(q|r) > 0\}$  do
4        $\alpha_t(q) \leftarrow \alpha_t(q) + p(q|r)\alpha_{t-1}(r)$ 
5 for  $q = 1 \dots N$  do
6    $\alpha_t(q) \leftarrow \alpha_t(q)p(\bar{x}_t|q)$ 

```

With the push form of implementation (where we push only to non-zero values), then the computation shrinks from 16 in Figure 8.18 down to $2 + 1 + 2 + 1 = 6$ operations per time step.

Another possible approach is to consider the merging of states. For example, it might be possible to consider the case when we get to a state with only one outgoing, we can collapse that into the effect of the current state. For example, perhaps states 1 and 4 could be considered together (as one state) and states 2 and 3 could be considered together. But the problem is that with a Moore machine, the number of types of observed behavior has to correspond to the number of states, since the observation is a function of the state. Even though in some sense this language has four behaviors (i.e., anything ending with a “(“ character, anything ending with a “)” character, and the two corresponding characters) the Moore machine is unable to achieve this language with fewer than 4 states.

With a Mealy machine, where state transitions have emissions, things are different. In fact, one can generate this language with only two states, and where transitions between states correspond to the transition between behaviors. Figure 8.19 shows a Mealy machine associated with this language. Since there are two states, and since behaviors correspond to transitions between states, we are able to implement the four behaviors with only two states with 4 transitions. The complexity of this is then n^2 which in this case is only 4.

Hence, in general the Mealy and Moore FSAs while being equally powerful in terms of the languages that they may generate are not equally powerful in terms of how many states they need to implement a given language. Moreover, transforming between a Mealy and a Moore machine might change the meaning of the states. For example, Figure 8.20 shows a Mealy and a Moore machine for generating a set of possible pronunciations for the word “and.” Again, here, we see that the Mealy machine can do it with fewer states.

In general, which machines (Mealy or Moore) are more useful? It really depends. It has been the case that mealy style machines seem to be quite useful for describing operations such as composition [129]. On the other hand, Moore machines are quite simple. Indeed, the ball and urn HMM examples of §8.4.1 first described by Lawrence Rabiner, and the random function of a Markov chain example of § 8.4.2 certainly have the flavor of Moore HMMs.

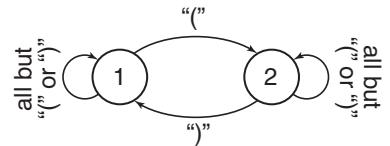


Figure 8.19: Mealy machine parenthesis matching state transitions

Indeed, SFSA and SFSTs (under a probability semiring, which we will henceforth assume) seem to share quite a bit in common with HMMs. First, there is a finite set of states \mathcal{Q} and a probability distribution between states $\Pr(r|q)$. Secondly, there are outputs associated with states (in the Moore case). HMMs, like Mealy machines, have also traditionally been defined, by Fred Jelinek, so that there is an output associated with each transition [205, 206].

One difference, however, is that in an HMM we have a distribution over outputs rather than one output for each state, as this can be seen as $p(x|q)$ in the Moore machine case and $p(x|q \rightarrow r)$ in the Mealy machine case⁷. When we consider the stochastic variants of FSA, however, we see that this difference vanishes.

In the case of a Moore machine, suppose that we wanted to have stochastic outputs at each state rather than deterministic outputs. That is, we should have that for a state $q \in \mathcal{Q}$ we have that $b(q)$ is a distribution. Let's say that we want to have it that $b(q)(o) \propto p(o|q)$. One simple way to

do this is to expand the state space. That is, rather than $|\mathcal{Q}|$ states, we will now have $|\mathcal{Q}| \times |\mathcal{O}|$ states \mathcal{Q}' . The original states will be called $s \in \mathcal{S}$ and the set of new states will be referenced by the pair $(s, o) \in \mathcal{S} \times \mathcal{O}$. Each state $s \in \mathcal{Q}$ then becomes $|\mathcal{O}|$ states. If we are currently in state s , whenever we wish to make a transition to original state r and then emit observation $o \in \mathcal{O}$ in state r , we will instead make a transition to state (r, o) and do this with probability $p(r|s)p(o|r)$. That is, given a pair of states $(r, o) \in \mathcal{Q}'$ and $(r', o') \in \mathcal{Q}'$, we have a transition probability between these pairs of new states $(r, o) \rightarrow (r', o')$ as follows:

$$\Pr((r', o')|(r, o)) = p(r'|r)p(o'|r') \quad (8.51)$$

Note that

$$\sum_{(r', o')} \Pr((r', o')|(r, o)) = \sum_{r', o'} p(o'|r')p(r'|r) = \sum_{r'} p(r'|r) = 1 \quad (8.52)$$

as required for a valid stochastic state transition matrix in an SFSA. An example of this transformation is given Figure 8.21.

A Mealy-like machine with stochastic outputs can also be transformed in a similar way to a machine where only the transitions are random, and at each transition between two states only a single symbol is generated. In this case, the resulting machine is quite likely to have more than one transition between any two states. That is, it is no longer possible to represent the arcs as (q, r) where $q, r \in \mathcal{Q}$. Instead, an arc is also labeled with a symbol from \mathcal{S} . This means that the graphs corresponding to the FSA are not a simple

⁷It should be clear that in this section, we are using \mathcal{Q} as the set of HMM states rather than \mathcal{Y} , and we are using lower case q or r for a state rather than lower case y . After this section, we will go back to using $Y = y$ for states of an HMM. While HMM literature historically used q for a state, the machine learning literature has more recently more often used y .

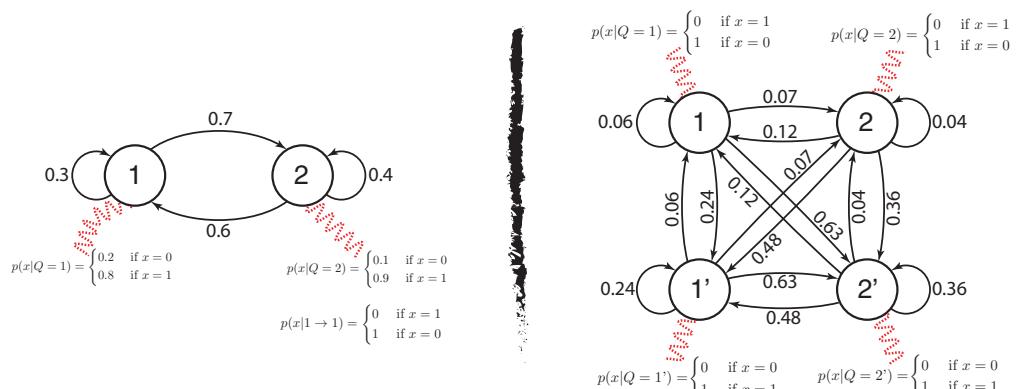


Figure 8.21: This figure shows the transformation to a deterministic output Moore machine. On the left shows the original Stochastic Moore machine, with output distributions indicated by wavy dotted red lines. On the right, we see the transformed state machine where each output distribution is deterministic. The “randomness” of the original output distributions has been factored into the edges.

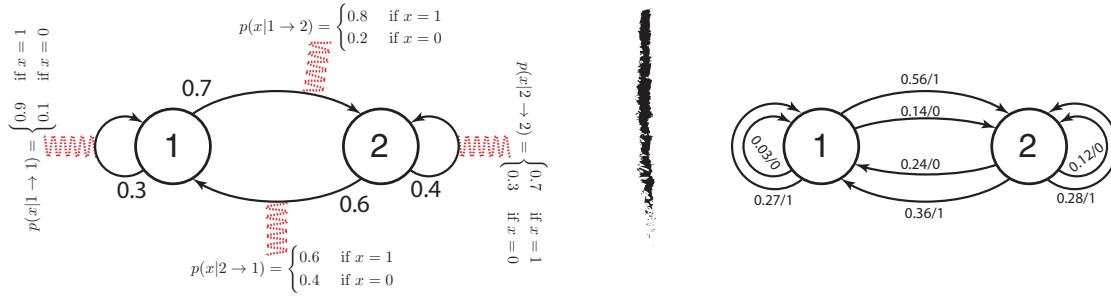


Figure 8.22: Left: Mealy machine where each arc has a stochastic output distribution. Right: An equivalent Mealy machine where each arc has a deterministic distribution and hence is shown where each arc is annotated with a label along with the transition probability of the form p/ℓ where p is the probability and ℓ is the label.

graph and have parallel edges, meaning there can be more than one edge (in a given direction) between two states. An example is shown in Figure 8.22. If one also labels arcs with the corresponding probability (and also allows for both input and output symbols to be labeled along the arc) then we immediately get weighted transducer of Definition 122.

Thus, any stochastic finite-state and finite-output alphabet HMM can be transformed into a stochastic FSA with deterministic outputs and hence an HMM can be seen as a SFSA, and this of course also includes the weighted finite-state transducer (where we only need to ignore the output symbols of the transducer to get a traditional HMM (more on this in §??).

In fact, this is the reason why we say that Shannon's 1948 paper [393] first defined HMMs. In that paper's Figure 5, there is an HMM shown with both stochastic transitions and observations but where the stochastic observations are expressed by there being more than one possible weighted transition (arc) between two states (just like Figure 8.22-right).

8.4.4.7.1 Rabiner vs. Jelinek HMMs: From the above, we see that there are two types of HMMs, one where the observations are associated with states and another where the observations are associated with state transitions, and that these types are equivalent in that they can be transformed into one another. We note that Fred Jelinek first defined HMMs where the output distributions were a function of the arcs, while Lawrence Rabiner first defined HMMs where the output distributions were a function of the current state. Hence, rather than calling them Mealy vs. Moore HMMs, we will call them, respectively, *Jelinek HMMs* vs. *Rabiner HMMs*.

One last point we should discuss here and that is epsilon arcs. In a Jelinek HMM, it is easy to have epsilon arcs, namely we are able to jump from one state to the next without needing to emit any symbols. In a Rabiner HMM, arcs have no emissions so this means that the observation distributions must also have epsilon symbols. One simple way of doing that is to augment the alphabet: rather than draw from an alphabet \mathcal{O} , we draw from $\mathcal{O} \cup \{\varepsilon\}$ as we did in Definition 122.

Henceforth, we will treat SFAs and HMMs as being the same. When we start discussing dynamic Bayesian networks, however, we will revisit the topic of weighted finite-state transducers as there are some interesting connections to make there (see §).

8.4.5 HMMs as a Trellis or a “Lattice”

The word “lattice” comes from lattice theory where there is a set of objects along with a “join” and a “meet” operator that, if certain properties hold, create a mathematical lattice. Lattices were studied by Birkhoff

[56], and one of the ways to graphically depict lattices is via a Hasse diagram. The term “lattice” has also been adopted in the speech recognition community as a structure to efficiently represent a very large number of subsequences in a Markov chain. Note that a speech recognition lattice is quite different than a Birkhoff lattice, although the Hasse diagrams do have some semblance to the lattice diagrams often used in speech.

In this document, we tend to use the term “trellis” rather than lattice since it is a bit less mathematically overloaded. The word “mesh” is also possible, but mesh sometimes implies a regular or symmetric grid, and the trellis structures we discuss need not have any particular form of symmetry. For historical reasons, however, we will use the term “trellis” and “lattice” interchangeably. On the other hand, a lattice will come up again when we discuss: 1) the lattices that are used in GMTK’s Lattice CPT (where lattice refers to things like word lattice or phone lattice and hence is used in the way described above, see §); and 2) the lattice of backoff paths in a conditional probability table implemented via a generalized backoff strategy (where lattice is used in the traditional Birkhoff sense, see §)

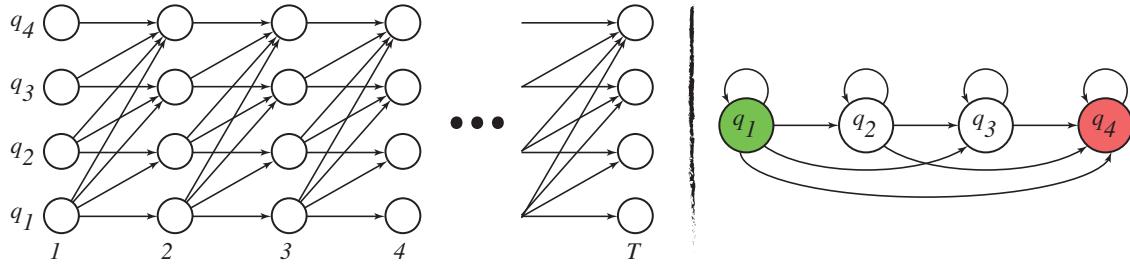


Figure 8.23: Left: lattice or trellis view of a Hidden Markov Model’s state transitions. This figure shows a 4-state HMM, and its possible state transitions over T time steps. Right: The state transition graph of nodes and arcs corresponding to the trellis on the left. Note that since the HMM’s emissions are not shown, this also describes the a 1st order Markov chain (cf. §8.3).

A general trellis representation of an HMM, or a state transition matrix, is a two-dimensional grid of *points* where the horizontal axis corresponds to discrete time and the vertical axis corresponds to discrete state (see Figure 8.23 and Figure 8.24 for an annotated version). The points in the trellis are connected by links that indicate which points may follow one another. Hence, a trellis is really a graph $G = (\mathcal{P}, \mathcal{L})$ consisting of a set of points \mathcal{P} and a set of links $\mathcal{L} \subseteq \mathcal{P} \times \mathcal{P}$. That is, the trellis’s links depicts the allowable transition structure of the underlying Markov chain in an HMM.

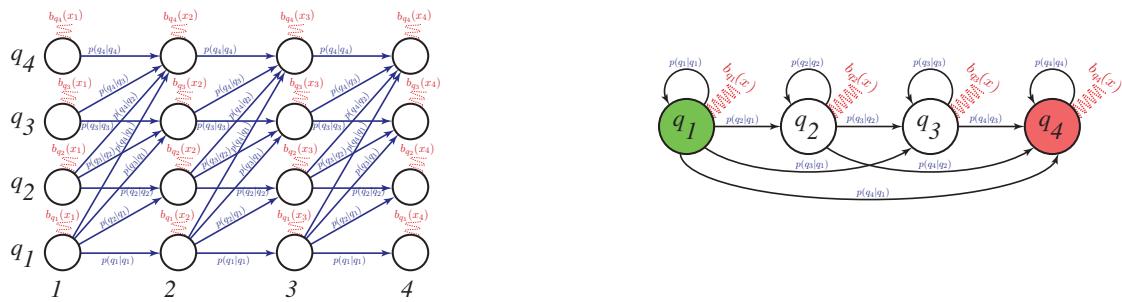


Figure 8.24: A view of Figure 8.23 but where the edges of both the trellis (on the left) and the HMM state transition diagram (on the right) have been labeled both with transition probabilities and emission/observation distributions. Note that what is being indicated is a time-homogeneous Markov chains since the state transition probabilities are shown as being the same for all t .

The trellis representation is the first to be explicit about absolute time, meaning that the width of the trellis corresponds to the window of time spanned by the representation. That is, the points \mathcal{P} are partitioned

into different time slices $\mathcal{P} = \mathcal{P}_0 \cup \mathcal{P}_1 \cup \dots \cup \mathcal{P}_T$ where each \mathcal{P}_t is at most isomorphic to the state space \mathcal{Q} (i.e., $|\mathcal{P}_t| \leq |\mathcal{Q}|$ for all t). We will see in fact that $\mathcal{P}_t \subseteq \mathcal{Q} \times \{t\}$, meaning that each entry in \mathcal{P}_t corresponds to a time-labeled state. Moreover, a given \mathcal{P}_t might be empty. This is different from the state transition graphs used to represent the states and allowable transitions (and which were used to represent state transitions in for FSAs in § 8.4.4).

The trellis view of an HMM's state space has some advantages and some disadvantages. On the one hand, the trellis only shows the state space and not the observation distributions. In the case of a Jelinek (resp. Rabiner) machine, it would be possible to augment each link (resp. point) in the trellis with an indicator of an observation distribution, similar to the way we did this on the left in Figure 8.22 (resp. Figure 8.21).

On the other hand, with a trellis it is quite easy to indicate time-inhomogeneous transition functions, where the allowable transitions between states will depend on the current time. Figures such as Figure 8.5 can only show a time-homogeneous Markov chain. Another way of saying this is that \mathcal{P}_t need not be isomorphic to \mathcal{P}_τ for $\tau \neq t$, nor need the edges in a trellis between two successive time steps. In a lattice, the pattern of zero/non-zero probabilities can hence depend on the current time, and this is quite common. The trellis therefore is often useful to display the HMM search space [206, 373] in a multi-pass speech recognition task, where a given word (or set of words) might be possible for only a limited time range. For example, a first recognition run might be based on a simple HMM model which produces a trellis showing only the likely word strings at various times. A second pass would use the trellis from the first pass to “rescore”, meaning the trellis restricts the search space to only those paths represented in the trellis. Hence, a second pass model can be much more complicated since its entire search space is not considered. Several of the trellis figures shown (e.g., Figure 8.28 and Figure 8.27) are time-inhomogeneous.

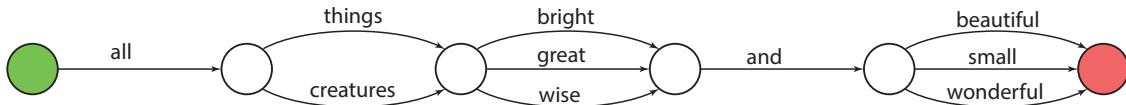


Figure 8.25: A lattice for generating sequences of words (18 sequences in total). Here, the time values associated with each of the points of the graph are not shown, nor are the probability scores associated with each edge.

Figure 8.25 shows a simple lattice that corresponds to a length 18 list of word sequences, where each sequence consists of 5 words. Note that displaying all lists would take considerably more space than the lattice. In fact, the lattice can generate an exponential number of sequences with a structure that is only linear in size.

Figure 8.26 shows an example of a peptide (subsegments of proteins) lattice. Here, the lattice corresponds to six peptides as indicated.

Figure 8.27 shows a lattice that is annotated: each point is annotated with the point identifier n_i and also with the time associated with the point; links are annotated with the probability score associated with the link as well as a label (which could correspond to any symbol from \mathcal{S} in the FSA terminology of § 8.4.4). Note that the lattices in Figures 8.25 and 8.26 and 8.27 really correspond only to a subset of the grid of points in Figure 8.23-left. For example, the lattice in Figure 8.27 only has one point at times $t = 0, 40, 60, 80, 90$ (i.e., $|\mathcal{P}_t| = 1$ for $t \in \{0, 40, 60, 80, 90\}$ and $|\mathcal{P}_t| = 0$ for other values of t). Hence a lattice is really a sparse representation of the space of states that might exist in an expansion of an HMM.

A slightly more complicated example of a lattice, one that we will revisit again in §, is given in Figure 8.28. As we will see, trellises and lattices can be directly used in GMTK without any pre-processing or prior transformation. GMTK in fact supports HTK lattices directly. This is described in §

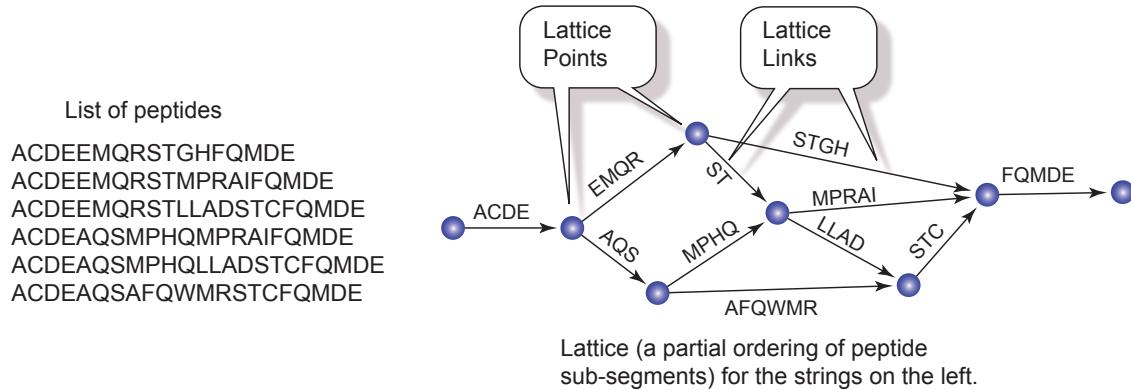


Figure 8.26: A lattice for generating a list of peptides (shown on the left) from a number of peptide segments (arc labels).

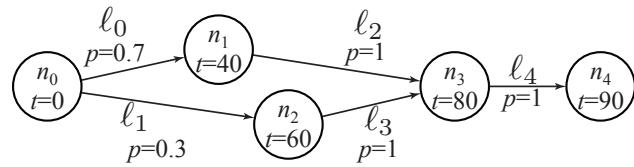


Figure 8.27: A lattice with annotated time points and links with both labels ℓ_i and probability scores.

8.4.6 Dynamic Time Warping (DTW)

Dynamic time warping (DTW) is a method that was used in early automatic speech recognition (ASR) systems. In the 1950s and early 1960s, the goal of ASR was to match a variable length acoustic speech utterance (a source) with a given template (a target). Within such systems, for each possible spoken utterance that could be recognized (such as a set of possible digits or words, each of which is spoken in isolation of each other), there existed a canonical “Platonic” instance of that object. For example, when recognizing the ten digits “oh”, “one”, “two”, …, “nine”, there would be a template instance of each of these digits. Each “perfect” instance might be represented as a temporal sequence of features of a given (perfect) length. The goal is to match an unknown utterance to each of the perfect templates and choose the one with the “best” match. An unknown spoken utterance, however, might not be the same length of the current template (or any template) and hence it is necessary to warp or align the objects to each other to determine a match. DTW addresses the problem of how to define “match.” Warping is done in time, and involves repetitions,

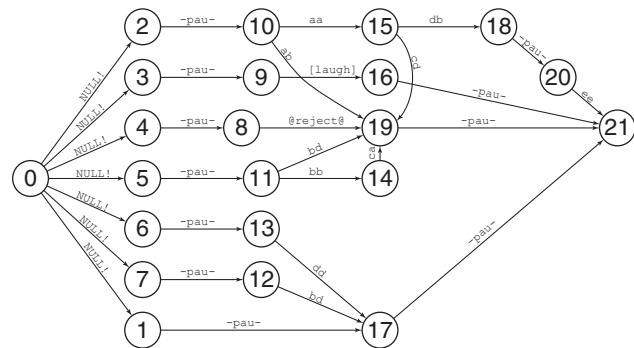


Figure 8.28: A small but real-world lattice with arc labels.

insertions, deletions, substitutions, and other distortions as well⁸.

While DTW is about “time” warping, and it was developed for speech recognition, we note that any dimension can be warped, such as position, length, and so on. Hence, while we may use the term time in this section to retain DTW as a moniker, “time” could mean any positional dimension. In fact, DTW is really just a special case of the dynamic programming paradigm of algorithms. Here, we study it, however, in the context in which it was defined for temporal sequences (and in particular speech recognition).

There are many ways to determine the closest match, and to do so we need a divergence function of some sort. This function takes the form $d(\text{unknown word}, \text{template}_i)$ for template i . It must possible to compute a numeric match score for objects of very different length, such as $d(\text{"Heeeeeellllloooooo"}, \text{"hello"})$ and hopefully it is the case that, e.g., $d(\text{"Heeeeeellllloooooo"}, \text{"hello"}) \gg d(\text{"Heeeeeellllloooooo"}, \text{"goodbye"})$.

To make this more concrete, assume that we have two sequences $x = (x_1, x_2, \dots, x_{T_x})$ and $y = (y_1, y_2, \dots, y_{T_y})$ of not necessarily the same length, so that we need not have that $T_x = T_y$ (in fact, it is quite likely that $T_x \neq T_y$). Each element of the sequence $x_i = (x_i(1), x_i(2), \dots, x_i(d))$ or $y_j = (y_j(1), y_j(2), \dots, y_j(d))$ consists of length d vectors of features. Assume that we have a “distortion” measure (which might or might not be a true distance [117]) between two d -vectors of the same length. For example, it might take the form

$$d(x_i, y_j) = \frac{1}{d} \sum_{k=1}^d (x_i(k) - y_j(k))^2 w_k \quad (8.53)$$

where w_i are non-negative weights and where $x_i(k)$ is the k^{th} element of x_i . Alternatively, we might generalize and use the Mahalanobis distance, as in:

$$d(x_i, y_j) = (x_i - y_j)^T A (x_i - y_j) \quad (8.54)$$

where A is a non-negative definite symmetric matrix. Indeed, there are many possibilities.

Now, typically $T_x \neq T_y$ so we can't use distances of the above form between sequences x and y since the total number of scalar elements in x and y are different. That is, since $T_x \neq T_y$, it would not make sense to define $d(x, y)$ as $\sum_{i=1}^{T_x} d(x_i, y_i)$.

Lets define a bit more notation. We define the index sets, and elements of the index sets, as follows:

$$i_x \in \{1, 2, \dots, T_x\} \quad (8.55)$$

$$i_y \in \{1, 2, \dots, T_y\} \quad (8.56)$$

We moreover use a shorthand to get at the distortion between individual elements.

$$d(i_x, i_y) \triangleq d(x_{i_x}, y_{i_y}) \quad (8.57)$$

That is, $d(i_x, i_y)$ can itself be Mahalanobis distance since it is the case that each of x_{i_x} and y_{i_y} are length- d vectors. In such case, $d(x, y)$ is really a distance between matrices, where one dimension of the matrix is the same and the other is different.

Given these definitions, one simple way of defining a distortion between x and y , $d(x, y)$, would be the following:

$$d(x, y) = \sum_{i_x=1}^{T_x} d(i_x, \text{round}(\frac{T_y}{T_x} i_x)) \quad (8.58)$$

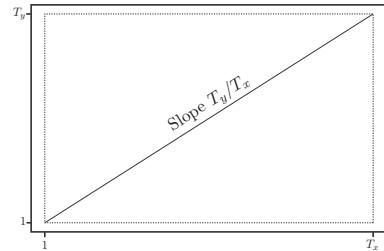


Figure 8.29: Linear time axis warping between source and target.

⁸Much of this section was modeled after the discussion of DTW in [347].

In other words, the “slope” $\frac{T_y}{T_x}$ linearly warps from the x time axis to the y time axis so that we can compute a distance between these two different length strings. This is shown in Figure 8.29.

Actually, Figure 8.29 is not a continuous mapping but a discrete mapping between points on a grid. That is, points on grid (i_x, i_y) corresponds to a distortion between the corresponding vectors at the corresponding times. A line corresponds to a path through i_x and round($T_y/T_x i_x$). We moreover have a vector of alignment points for each of x and y given by $(i_x(1), i_x(2), \dots, i_x(T))$ and $(i_y(1), i_y(2), \dots, i_y(T))$. The distance is then

$$d(x, y) = \sum_{t=1}^T d(i_x(t), i_y(t)). \quad (8.59)$$

For the case above, when $T_x \geq T_y$, we have

$$I_x \triangleq (i_x(1), i_x(2), \dots, i_x(T)) = (1, 2, \dots, T) \quad (8.60)$$

and

$$I_y \triangleq (i_y(1), i_y(2), \dots, i_y(T)) = (\text{round}(\frac{T_y}{T_x} 1), \text{round}(\frac{T_y}{T_x} 2), \dots, \text{round}(\frac{T_y}{T_x} T)) \quad (8.61)$$

where $T = \max(T_x, T_y)$. A symmetric case occurs if $T_x < T_y$.

Drawing this line is similar to drawing a smooth line on a pixelated computer screen (see Figure 8.30). There is an important problem with this warping though. Since all parts of one vector are warped uniformly to the other utterance, it can be the case that entirely wrong parts of the two vectors are mapped to each other, thereby yielding an artificially low value for $d(x, y)$. For example, consider matching the string “hhheeeeeelo” to “hello” (where we are using the orthographic spelling of words as indicators of its phonemic spelling in say real acoustic utterances). With these two instances and a linear warping, we see in Figure 8.30 we have “matches” between characters “h” and “e”, “e” and “l”, and “l” and “o”, which is highly suboptimal.

Jumping to the most general case, we can align x and y with specific alignment functions ϕ_x and ϕ_y which yield the indices I_x and I_y of alignment. That is, we have

$$I_x = (\phi_x(1), \phi_x(2), \dots, \phi_x(T)) \quad (8.62)$$

$$I_y = (\phi_y(1), \phi_y(2), \dots, \phi_y(T)) \quad (8.63)$$

where here $T \leq T_x T_y$. This then defines a global distance parameterized by the $\phi = (\phi_x, \phi_y, w^\phi)$ consisting of the two functions and a weight vector w^ϕ as follows:

$$d_\phi(x, y) = d_{\phi_x, \phi_y, w^\phi}(x, y) = \sum_{t=1}^T d(\phi_x(t), \phi_y(t)) w_t^\phi \quad (8.64)$$

where $d(\cdot, \cdot)$ is again a local distance function and w_t^ϕ is a weight at alignment point t that is dependent on ϕ .

Then, we can construct an overall distance by minimizing over all functions $\phi = (\phi_x, \phi_y)$ as in:

$$d(x, y) = \min_{\phi} d_\phi(x, y) \quad (8.65)$$

Note that when y is a template speech utterance and x is an unknown utterance, it makes sense to define $d(x, y)$ using the min operator. For example, if we used max instead, then even the same utterance could have a very large distance. An alternative might be to take the average (or some form of aggregation function) rather than the min. For example, we could take an expected value of the form

$$d(x, y) = E_{\Phi} d_{\Phi}(x, y) = \sum_{\phi} d_{\phi}(x, y) \Pr(\phi) \quad (8.66)$$

where $\Pr(\phi)$ is the probability of warping options ϕ that must be obtained from somewhere (e.g., by hand, or learnt from data). The min function is standard, however, and when we morph from DTW to HMMs, we will see will have a nice probabilistic interpretation both for the min and for how to produce probabilities for each ϕ warping.

Now, as given, there are many many possible paths that could be taken, in fact all possible functions ϕ_x and ϕ_y that create an alignment, and the length of the alignment is allowed even be greater than the lengths T_x and T_y . Each element $\phi_x(k)$ may take on any of T_x possible values, so the number of ϕ_x functions is T_x^T . The total number of functions, then, is $(T_x T_y)^T$ and minimizing over all ϕ is prohibitively expensive for large T_x, T_y . It seems natural that some flexibility should be dropped in order to gain tractability (and reasonableness).

Moreover, many functions do not make sense for a given application. For example, consider the alignment paths given in Figure 8.31. If it is the case that the objects x and y do involve some sort of total or partial order (for example, the acoustic feature stream of a speech utterance and a sequence of words that might be spoken), then it would be reasonable to assume that the alignment should be mostly in the increasing direction of this order, rather than the zig-zag pattern shown in the figure. Consider words: “you” vs. “we”, which phonetically are reversals of each other — without constraints, these two words could be perfectly aligned.

On the other hand, for certain applications like statistical machine translation (where x might be a string in one language and y a string in another), a more flexible alignment is probably required. Also, we might want to place constraints on where the alignments should begin and end, even in the machine translation case. In general, there clearly should be some constraints placed on the ϕ functions.

One simple set of would be endpoint constraints, i.e., of the form:

$$\phi_x(1) = 1, \quad \phi_x(T) = T_x, \quad \phi_y(1) = 1, \quad \phi_y(T) = T_y. \quad (8.67)$$

The constraints ensure that we start the alignment at the beginning of the strings and end the alignment at the end of the string and already would exclude the alignment given in Figure 8.31.

Another possible set of constraints would be to insist upon monotonicity. I.e., the alignment is not allowed to go backwards in time, and these would take the form:

$$\phi_x(t+1) \geq \phi_x(t) \quad \text{and} \quad \phi_y(t+1) \geq \phi_y(t) \quad \forall t \quad (8.68)$$

We might also want to have local “continuity” constraints, meaning that we never want the alignment to jump by more than one step. These could be expressed as:

$$\forall t, \quad \phi_x(t+1) - \phi_x(t) \leq 1 \quad \text{and} \quad \phi_y(t+1) - \phi_y(t) \leq 1 \quad (8.69)$$

Since the constraints here and in Equation (8.68) apply for all t , there is no positional dependence in the grid on the constraints, meaning the constraints apply uniformly over the grid regardless of the grid position.

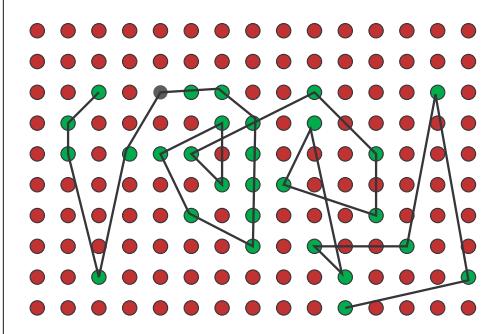


Figure 8.31: A probably absurd alignment.

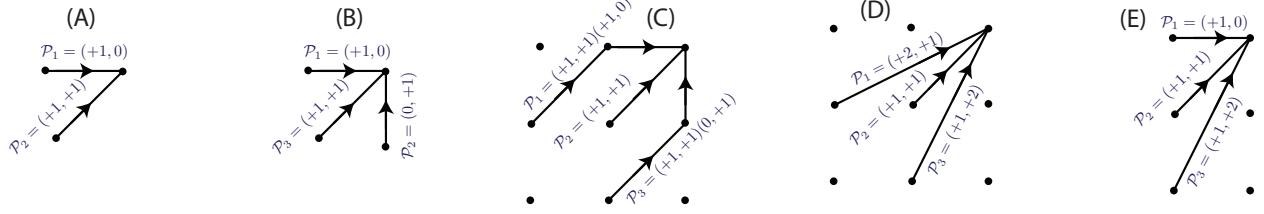


Figure 8.32: Five patterns each of which is created by merging three paths. Assume that vertical axis represents the exemplar y and the horizontal axis represents the unknown sequence x . Then (A) allows only matches, substitutions, or deletions; (B) allows matches, substitutions, insertions, and deletions (cf. §8.4.6.1.1 for a discussion on insertions and deletions); (C) allows a match, deletion, deletion right after a match, and an insertion right after a match; (D) allows skips; and (E) allows a combination of certain skips, deletions, or matches/substitutions.

We'll see in §8.4.7 that this corresponds to the time-inhomogeneous assumption in a Markov chain (cf. §8.3).

Now we could continue in this fashion, expressing constraints by equations, but perhaps a more effective strategy is to define a template pattern of allowable transitions and to impose that the local trajectory of any alignment must respect the local constraints specified by the template. This also will begin to tie into §8.4.4. The templates, moreover, are specified by a set of allowable *relative* paths that are unioned together to define the template. A path consists of a sequence of transition deltas, of the form $\mathcal{P} = ((\delta_x(1), \delta_y(1)), (\delta_x(2), \delta_y(2)), \dots, (\delta_x(k), \delta_y(k)))$. For example, simple one step paths would be $\mathcal{P}_1 = (+1, 0)$, $\mathcal{P}_2 = (0, +1)$, $\mathcal{P}_3 = (+1, +1)$. Merging these three paths leads to the template $\mathcal{P}_{1\cup 2\cup 3} = \mathcal{P}_1 \cup \mathcal{P}_2 \cup \mathcal{P}_3$. Paths are directed, and the way paths are unioned together is by joining them at final the head of the path.

Path merging is shown in Figure 8.32, and since many templates are possible, and a few simple additional ones are also shown in Figure 8.32. One might ask, in the figure, what is the difference between case (C) and case (D)? Lets number the local grid with (i, j) starting at $(1, 1)$. In case (D), we jump from $(1, 2)$ to $(3, 3)$ while in case (C), in going from $(1, 2)$ to $(3, 3)$ we must go through $(2, 3)$. Hence in case (C) we must also incur the local distortion of cell $(2, 3)$ in making the jump from $(1, 2)$ to $(3, 3)$ while in case (D) we do not incur this local distortion.

Global path constraints are constraints that are imposed directly on absolute locations in the grid. For example, we might just say that location $(10, 15)$ is impossible and no alignment may go through it. Global constraints are sometimes implied by the local constraints. For example, consider the insertion only pattern in Figure 8.32-(A), then in the complete grid, not all points are possible to reach. If we were to overlay Figure 8.32-(A) onto a grid, we would get the pattern shown in Figure 8.33.

More generally, this form of global constraint is based on an aggregation of information from all possible patterns. The constraint in Figure 8.33 is really a form of slope constraint, but this can be generalized over all patterns based on a global minimum and global maximum slope. These combined with the start/end constraints of Equation (8.67) make for some interesting global constraints. Let the ℓ 'th path

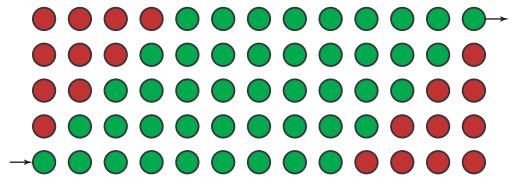


Figure 8.33: Global constraints from local constraint. Only the green points in the grid are reachable due to the local constraint from Figure 8.32-(A) applied everywhere. Arrows indicate local constraints of Equation (8.67).

of a pattern be referred to as:

$$\mathcal{P}^\ell = \left((\delta_x^\ell(1), \delta_y^\ell(1)), (\delta_x^\ell(2), \delta_y^\ell(2)), \dots, (\delta_x^\ell(k_\ell), \delta_y^\ell(k_\ell)) \right) \quad (8.70)$$

Then we can define a global max m^+ and min m^- slope as follows:

$$m^+ = \max_\ell \frac{\sum_{t=1}^{k_\ell} \delta_y^\ell(1)}{\sum_{t=1}^{k_\ell} \delta_x^\ell(1)} \quad m^- = \min_\ell \frac{\sum_{t=1}^{k_\ell} \delta_y^\ell(1)}{\sum_{t=1}^{k_\ell} \delta_x^\ell(1)} \quad (8.71)$$

Note for patterns that are diagonally symmetric, we have that $m^+m^- = 1$. As an example, for the pattern in Figure 8.33-(D), we have $m^+ = 2 = 1/m^-$.

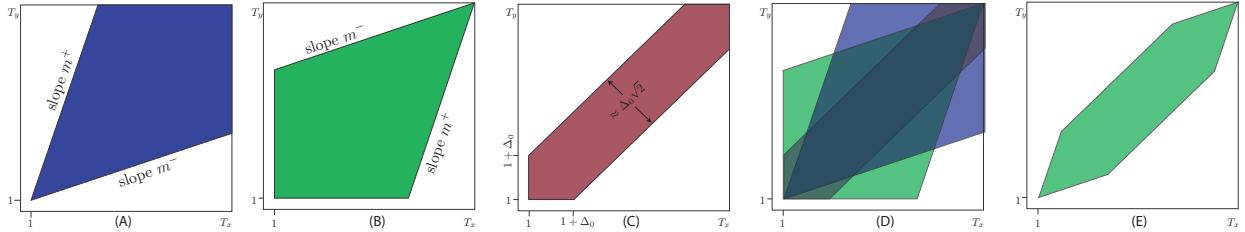


Figure 8.34: (A) shaded region indicates possible points that could be active based on global constraints implied by the max/min slopes slopes and the constraint of starting at $(1, 1)$. (B) slope constraints along with constraint of ending at (T_x, T_y) . (C) shaded region corresponds to limited asynchrony constraint. (D) overlap of the three previous constraints. (E) shaded region shows possible points when intersecting all previous constraints.

These slopes, along with the start/end constraints of Equation (8.67), define global constraints in several ways: the points that are reachable from $(1, 1)$ put a constraint on the relationship between $\phi_x(k)$ and $\phi_y(k)$ as follows:

$$1 + m^-(\phi_x(k) - 1) \leq \phi_y(k) \leq 1 + m^+(\phi_x(k) - 1). \quad (8.72)$$

This is shown in Figure 8.34-A. The points that may reach (T_x, T_y) can be defined as

$$T_y + m^+(\phi_x(k) - T_x) \leq \phi_y(k) \leq T_y + m^-(\phi_x(k) - T_x). \quad (8.73)$$

and are shown in Figure 8.34-B. To this, we can add global synchrony constraints of the form

$$|\phi_x(k) - \phi_y(k)| \leq \Delta_0 \quad (8.74)$$

and which are shown in Figure 8.34-C. Taking all of these constraints together yields the intersection of the possible points, and we get the global constraint shown in Figure 8.34-C, with the final set of reachable points shown in Figure 8.34-E.

We have not yet addressed the weight values w_t^ϕ in Equation (8.64). These are sometimes called slope weights since in some sense they can be seen as putting a weight or cost on the slope of the alignment made at the current point. In general, not all slopes need be treated equal, but how should w_t^ϕ be determined? One way is via the slope itself. For example, we could set either

$$w_t = \min(\phi_x(t) - \phi_x(t-1), \phi_y(t) - \phi_y(t-1), 1) \quad (8.75)$$

or

$$w_t = \max(\phi_x(t) - \phi_x(t-1), \phi_y(t) - \phi_y(t-1)) \quad (8.76)$$

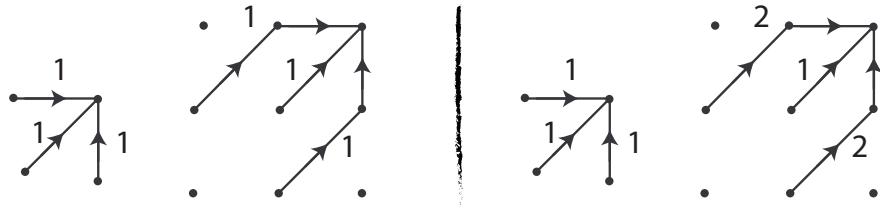


Figure 8.35: Left: two patterns with path weights according to Equation (8.75). Note that the weights in Equation (8.75) are on each segment on each path, but the weights shown in the figure correspond to the entire path for each pattern. Right: two patterns with path weights according to Equation (8.76).

and these are shown in Figure 8.35.

In general, constraints such as the above, the paths, the patterns, and the path weights all correspond to our heuristic beliefs about the reasonable differences between a template string y and the unknown string x . If there is too much flexibility, then it will be possible for different strings to be warped together and the distances between different objects will be too small. For example, consider the phonetic constituents of the two words “you” and “we” — if a reverse alignment was possible, we might find that $d(\text{"you"}, \text{"we"})$ could be quite small. On the other hand, if there is too little flexibility in the alignment functions, then different strings that are the same but that are highly warped w.r.t. each other will fail to be aligned. For example, consider again the “hhheeeeeelo” vs. “hello” example above — “hhheeeeeelo” is temporally warped but is otherwise identical to “hello”, so enough flexibility in the warping is needed to see this.

We note, this a standard problem in pattern classification which is not really perfectly solved by machine learning — that is, if a region for a given class is too general and broad, we will get false positives, and if it is too narrow and constraining, we will get false negatives. In general, the best strategy is to allow any uncertainty about any local decisions to propagate until a final global decision is being made. Having such a hierarchical representation of uncertainty can be important. With DTW, however, there really is not a good representation of uncertainty as there is with probability distributions.

In later sections of this text, we will see in the context of dynamic Bayesian networks that constructs very similar to local constraints can be expressed easily via virtual evidence (cf. §).

8.4.6.1 Dynamic Time Warping and Dynamic Programming

Even with a large number of constraints, computation of the distance in Equation (8.65) appears to be a daunting task, requiring computation exponential in T_x and T_y . The computation can be made tractable (and actually quite fast) via the use of dynamic programming. In fact, dynamic programming is critical in almost all graphical model inference tasks — the elimination algorithm described in §2.2.1 is in fact a dynamic programming algorithm. Moreover, the computation of $p(x)$ we already saw for HMMs used dynamic programming (cf. §8.4.3). Here we describe it in the context of DTW, which will be helpful when we generalize it to HMMs and dynamic graphical models.

Dynamic program is a class of algorithms where there is an inherent recursive structure. The recursive structure is such that an optimal solution to a problem at a given point is based on optimal solutions and other (sub-problem) points which are then aggregated in some way. Moreover, these sub-problems are reused over multiple points.

It is said that for dynamic programming to work, we need two things: *optimal substructure* and *overlapping subproblems*.

Optimal Substructure means the following: given an optimal solution to a problem, that solution contains optimal solutions to subproblems. For example, consider traveling from a building where you work to a point north of this building (say point A). Given optimal way to get from inside the building to point A, it must

include an optimal way to get out of the building heading north. If it didn't, then the solution to get to point A would not have been optimal.

To relate this to the grids we've been discussing, consider Figure 8.36. Suppose an optimal solution to point A in the grid (meaning have the optimal path to A starting at (1,1)) goes through C (using arc EC). If the path to C (via arc EC) was not optimal (and say instead the path to C via arc EC' was optimal), then neither would the path to A via arc EC be optimal since that path could be improved by taking the initial path to C via arc EC'. Therefore, if we have an optimal path to A (via EC) we must also have optimal path to C (via arc EC). When we compute the optimal path to A, we first compute optimal paths to all the things that could directly link to A (e.g., B,C, and D), and then choose the one (B,C, or D) that when connected to A is optimal. This is then done recursively.

Common sub-problems means that the optimal sub-problems are used many times over. For example, given optimal way to get from your office inside your building to point A, that includes a solution to get from your office to the door pointing north. This sub-problem (getting out of your building) could hence be useful for another optimal problem (e.g., leaving your building and going to a different point north of your building). The common sub-problem (of getting from your building to the door pointing north) is common to all problems involving going to any point north of your building.

With respect to grids, consider Figure 8.37. Suppose we have optimal solution to point A in the grid (so we have the optimal path to A). This means that we have computed and considered solutions to A via both B and C, and therefore must have computed optimal solutions to B and C. When we start computing the optimal solutions to D, we need not re-compute the optimal solutions to B and C since they have already been computed. This means, conceptually, if we were to solve dynamic programming using recursion, we memorize the solutions to the sub-problems as we go along in case they might be needed again. Memorizing such solutions is often called memoization.

So how do we solve DTW using dynamic programming via exploiting optimal substructure and overlapping subproblems? Lets define a partial computation up to time $T' \leq T$ as follows:

$$d_{T'}(i_x, i_y) \triangleq \min_{\phi: \phi_x(T') = i_x, \phi_y(T') = i_y} \sum_{t=1}^{T'} d(\phi_x(t), \phi_y(t)) w_t^\phi \quad (8.77)$$

with this, we can define a dynamic programming recursion as follows:

$$d_{T'}(i_x, i_y) = \min_{P \in \mathcal{P}(i_x, i_y)} \left[d_{T'-\ell(P)}(P_x(1), P_y(1)) + d((P_x(1), P_y(1)), (P_x(\ell(P)), P_y(\ell(P)))) \right] \quad (8.78)$$

where $\mathcal{P}(i_x, i_y)$ is the set of all valid paths that end at point (i_x, i_y) , $\ell(P)$ is the length of path P , $P_x(i)$ (resp. $P_y(i)$) is the x -coordinate (resp. y -coordinate) of the path at relative position i within the path (hence, when $P \in \mathcal{P}(i_x, i_y)$ we have $(P_x(\ell(P)), P_y(\ell(P))) = (i_x, i_y)$ and $(P_x(\ell(1)), P_y(\ell(1)))$ are the relative coordinates of the first position in the path that ends at (i_x, i_y)), and finally $d((P_x(1), P_y(1)), (P_x(\ell(P)), P_y(\ell(P))))$ is the accumulated distance from start to finish along path P . That is,

$$d((P_x(1), P_y(1)), (P_x(\ell(P)), P_y(\ell(P)))) = \sum_{t=1}^{\ell(P)} d(P_x(t), P_y(t)) w_t^P \quad (8.79)$$

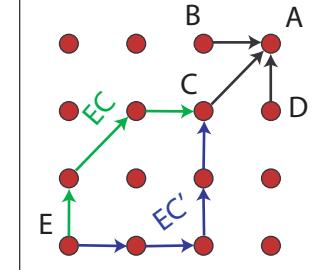


Figure 8.36: Optimal Sub-structure

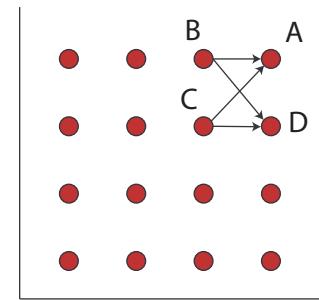


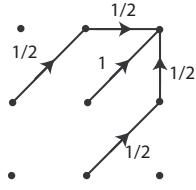
Figure 8.37: Common Sub-expressions

where w_t^P is the appropriate (slope) weight along path P at relative point t .

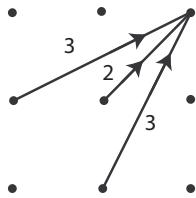
This might seem complicated in its general form, but based on the pattern, we only need to consider the allowable paths at the current point. Hence, the dynamic programming recursions take on fairly simple forms depending on each pattern and the path weights. A few simple ones are shown in Figure 8.38.



$$D(i_x, i_y) = \min \begin{cases} D(i_x - 1, i_y) + d(i_x, i_y), \\ D(i_x - 1, i_y - 1) + 2d(i_x, i_y), \\ D(i_x, i_y - 1) + d(i_x, i_y) \end{cases} \quad (8.80)$$



$$D(i_x, i_y) = \min \begin{cases} D(i_x - 2, i_y - 1) + \frac{1}{2}[d(i_x - 1, i_y) + d(i_x, i_y)], \\ D(i_x - 1, i_y - 1) + d(i_x, i_y), \\ D(i_x - 1, i_y - 2) + \frac{1}{2}[d(i_x, i_y - 1) + d(i_x, i_y)] \end{cases} \quad (8.81)$$



$$D(i_x, i_y) = \min \begin{cases} D(i_x - 2, i_y - 1) + 3d(i_x, i_y), \\ D(i_x - 1, i_y - 1) + 2d(i_x, i_y), \\ D(i_x - 1, i_y - 2) + 3d(i_x, i_y) \end{cases} \quad (8.82)$$

Figure 8.38: Three examples of pattern constraints (on the left) and the corresponding dynamic programming recursions to implement these constraints.

The recursion starts simply with

$$d_1(1, 1) = d(1, 1)w(1) \quad (8.83)$$

and for any i, j that might start the recursion $d_1(1, j) = d(1, j)$, and $d_1(i, 1) = d(i, 1)$. The recursion ends with; and ends with the final distance between the sequences x and y as:

$$d(x, y) = d_T(T_x, T_y) \quad (8.84)$$

The overall computation depends on the constraints, but for the constraints listed in Figure 8.38, the computation is only $O(T_x T_y)$ which is quite efficient.

Getting the optimal path requires backtracing from the optimum point. That is, at each point, we need to store the location of the previous point that yielded the optimal path up to the current point. That is, in addition to the step in Equation (8.85), we would also keep track of the indices that achieved the minimum at each step, as in for all (i_x, i_y) :

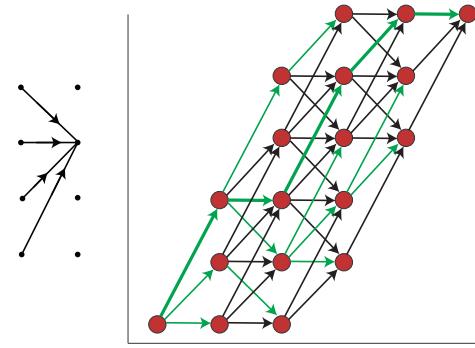
$$P^* \in \underset{P \in \mathcal{P}(i_x, i_y)}{\operatorname{argmin}} \left[d_{T' - \ell(P)}(P_x(1), P_y(1)) + d((P_x(1), P_y(1)), (P_x(\ell(P)), P_y(\ell(P)))) \right] \quad (8.85)$$

$$\psi_{T'}(i_x, i_y) = (P_x^*(1), P_y^*(1)) \quad (8.86)$$

The first step computes the optimal path P^* and the second step stores the initial indices of this optimal path. For the patterns above, the additional recursion might take the slightly more simple form of:

$$\psi(i_x, i_y) \in \underset{i'_x, i'_y}{\operatorname{argmin}} [D(i'_x, i'_y) + w(i'_x, i'_y)d(i'_x, i'_y)] \quad (8.87)$$

Figure 8.39: Dynamic programming backtracking in the context of DTW. The pattern associated with this grid (far right) is given by the near right figure. The figure here shows the complete expansion from start $(1, 1)$ to end (T_x, T_y) and where at each point, the best previous point is indicated by the green arc (where the set of possible immediately preceding points is given by the constraints in the pattern. Hence, each point has one unique previous best point. Once we reach (T_x, T_y) we can backtrack the series of unique previous best points to obtain the optimal alignment (indicated by the bold green path).



Once we have these values stored, we can backtrack starting at the end.

```

1  $(i_x^T, i_y^T) \leftarrow (T_x, T_y);$ 
2 for  $k = T$  down to 2 incrementing by  $-1$  do
3    $\sqcup (i_x^{k-1}, i_y^{k-1}) \leftarrow \psi_k(i_x^k, i_y^k)$ 
```

Using the pattern in Figure 8.32-(E), Figure 8.39 shows the the optimal previous points (green links) and global optimal path (bold green path).

8.4.6.1.1 String to String Alignment: DTW is often used for string-to-string alignment which is used to produce the edit distance or Levenshtein distance between the two strings. The algorithm is used quite a bit in computational biology as well as in text processing (e.g., spelling checking) and in fact some form of it is also used for automatic document layout for programs such as Microsoft word and latex. In the context of string-to-string alignment, there is some terminology that readers should be quite familiar with. Here x and y are strings, and x is known as the source string and y is known as the target string. The source string uses symbols from the source alphabet so $x_i \in \mathcal{X}$ and the target string uses symbols from the target alphabet $y_i \in \mathcal{Y}$. The goal is to transform (or really *align*) source string to the target string by a series of edit operations which correspond to DTW using the pattern from Figure 8.32-(A). There are four edit operations in total, each one corresponds to one of the paths in Figure 8.32-(A). In fact, we can “name” each of the edit operations with a pair $(s, t) \in (\mathcal{X} \cup \{\varepsilon\}) \times (\mathcal{Y} \cup \{\varepsilon\})$. Then the edit operations are as follows:

- Deletion: any (s, t) such that $s \neq \varepsilon$ and $t = \varepsilon$. This corresponds to path step $(\delta_x(t), \delta_y(t)) = (1, 0)$ and so is a vertical-only step in the grid. Since we are aligning the source to the target string, a deletion is relative to the source and means that we are deleting one element of the source string.
- Insertion: any (s, t) such that $s = \varepsilon$ and $t \neq \varepsilon$. This corresponds to path step $(\delta_x(t), \delta_y(t)) = (0, 1)$ and so is a horizontal-only step in the grid. Since we are aligning the source to the target string, an insertion is relative to the source and means that we are inserting one element into the source string from the target string.
- Substitutions: any (s, t) such that $s \neq \varepsilon$, $t \neq \varepsilon$, and $s \neq t$. This corresponds to path step $(\delta_x(t), \delta_y(t)) = (1, 1)$ and so is a diagonal step in the grid.
- Match: any (s, t) such that $s \neq \varepsilon$, $t \neq \varepsilon$, and $s = t$ (when the two alphabets allow this meaning that there is some symbol overlap). This corresponds to path step $(\delta_x(t), \delta_y(t)) = (1, 1)$ and so is also diagonal step in the grid.

Note that the role of source and target may be swapped in which case insertions become deletions and vice versa. In a given application, one must choose what is the source and what is the target in order to ground the otherwise ambiguous terms deletion and insertion.

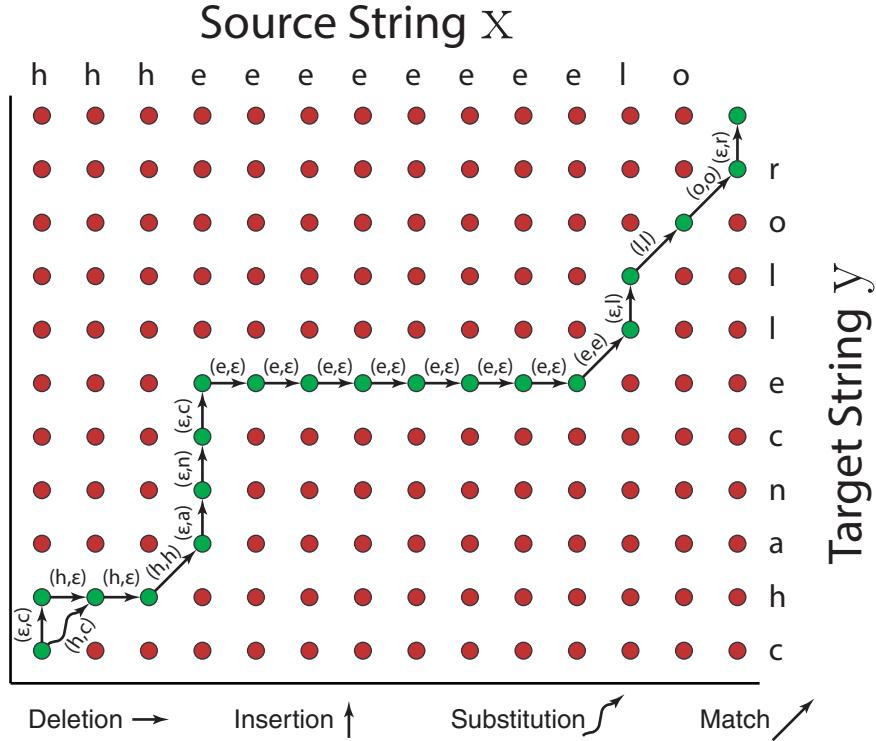


Figure 8.40: Example of two alignments between the two strings: source string $x = \text{"hhheeeeeelo"}$ and target string $y = \text{"chancellor"}$. In one alignment, there are deletions, insertions, and matches, but no substitutions. In the other alignment, there is one substitution (h, c) in place of the deletion (ε, c) and insertion (h, ε) of the other alignment, but otherwise the two alignments are the same. Whether a given path is optimal or not, and which of the two paths is less costly, depends on the relative costs of each of the sum of the edit costs along each alignment path. Note that the path in the figure must conform to the constraints given in the pattern Figure 8.32-(A). Using a different pattern would allow a different set of paths from start to end.

8.4.6.1.2 DTW Discussion: Dynamic time warping implemented with dynamic programming is a powerful algorithm with many advantages. It is easy to implement and with a well designed pattern, one can get reasonable results for many pattern alignment tasks such as small vocabulary speech recognition tasks in controlled acoustic environments. It can do an adequate job aligning templates with different rates of string progression.

On the other hand, DTW is quite limited relative to techniques available today. For example, in speech recognition, only one exemplar might exist per possible utterance. It is possible to use multiple templates per utterance, and use the score from the lowest template to determine the unknown utterance, but then one does not have the flexibility to represent utterances that are a combination of multiple templates. Ideally, we would like a template may have an “averaging” behaviour over multiple templates. Moreover, the DTW patterns and constraints are constructed heuristically and sometimes the choice can be somewhat arbitrary. The path weights, too, even if based on slope are somewhat arbitrary. An ideal method it should automatically learn the weights. A further limitation is extensibility. For example, if we wish the exemplars to be merged to create sequentially grouped exemplars. This is the case in continuous speech recognition, were real utterances consist of any number of words strung together to form a spoken phrase. It is not immediately obvious how to generalize DTW to do this since the number of templates needed in such case would be intractably large. Moreover, there is no precise mathematical system for dealing with uncertainty in DTW.

Ideally, we'd like a formal probabilistic scheme, and learning and optimization process within that scheme.

There are many discussions of DTW, and DP in the literature. Two good sources are [347] (from which much of this section is based) and also [182]. We note that DTW can be encoded with dynamic Bayesian networks, as described in [144] and as described in §. Also, DTW and dynamic programming in general can easily be extended to multi-string alignment, a problem of great importance in computational biology [134, 343] — in this case, the cost of doing the alignment grows exponentially with the number of strings needing to be jointly aligned, so in such case numerous approximations must be made.

In the next section, we'll see how hidden Markov models generalize (pairwise alignment) DTW and satisfy many of these desires. How one may encode multi-string alignment with dynamic Bayesian networks is made clear by reading [144].

8.4.7 Morphing from DTW to HMM

In this section, we will “morph” from DTW to HMMs and in doing so show how they are related. We assume that the Markov chain of the HMM is first order (cf. §8.3).

An HMM has a certain number $n = |\mathcal{Q}|$ of states. We rename the DTW exemplar template length $T_y \rightarrow N$ so that the exemplar has length equal to the number of states in the HMM. We'll take the Rabiner view of HMMs in this section (cf. §8.4.4.2) so that there is an emission at each state. We rename the exemplar elements y_1, y_2, \dots, y_N to be the HMM states q_1, q_2, \dots, q_N .⁹ Hence, using the terminology of string alignment introduced in §8.4.6.1, the *target* corresponds to the states, and we are aligning a *source* sequence of observations x to the target states.

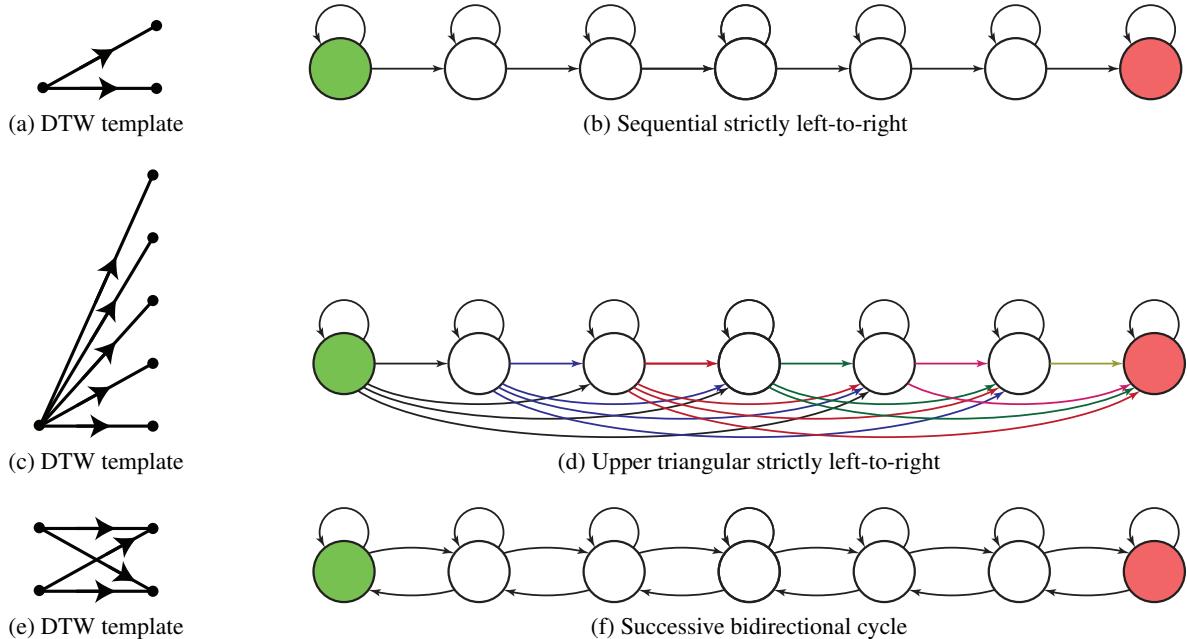


Figure 8.41: DTW templates (left) and their corresponding Markov chains transition diagrams (right).

In a Rabiner HMM, each state produces one emission. It is normally not the case that one can arrive at a state without producing an emission (although we will consider this case as well below) — doing so would constitute an insertion into the observation sequence from the state sequence. Hence, to avoid this

⁹In fact, it is common in the speech recognition community to call the HMM states $\{q_1, q_2, \dots, q_N\}$ while it is common in the machine learning community to call the HMM states $\{y_1, y_2, \dots, y_N\}$. We'll use both in this document.

we will disallow purely vertical path transitions (so no insertions are allowed). This means that we must always consume one element of x at each step, so paths such as \mathcal{P}_2 in the pattern in Figure 8.32-(B) and \mathcal{P}_3 in the pattern in Figure 8.32-(C) do not apply. This also means that x is never shorter than the shortest path through the set of states (which depends on the other constraints). Moreover, since the HMM's Markov chain is first order, the set of next allowable states must be determinable from current state, so paths such as \mathcal{P}_1 and (for an additional reason to the one above) \mathcal{P}_3 in the pattern in Figure 8.32-(C) also do not apply. For example, path \mathcal{P}_1 in Figure 8.32-(C) would say that an insertion is allowed only when followed by a match/substitution, a construct that could only be represented by a second-order Markov chain. Lastly, paths such as \mathcal{P}_1 in Figure 8.32-(D) are also not allowed since that would correspond to moving to a state that emitted two observations, but in a Rabiner HMM only one observation is emitted per state.

The restrictions in the previous paragraph can be relaxed if we use an augmented output alphabet. For example, if we allow epsilon observations (i.e., an alphabet $\mathcal{O} \cup \{\varepsilon\}$) then strictly vertical arrows (such as \mathcal{P}_2 in the pattern in Figure 8.32-(B)) would be allowed — that would correspond to emitting a ε . We could also augment the alphabet to be the Cartesian product $(\mathcal{O} \cup \{\varepsilon\}) \times (\mathcal{O} \cup \{\varepsilon\})$ (or alternatively, a subset of the disjoint union $(\mathcal{O} \cup \{\varepsilon\}) \uplus (\mathcal{O} \cup \{\varepsilon\})$) to allow constructs like \mathcal{P}_1 in Figure 8.32-(D), and higher-order Markov chains could deal with some of the other constructs (and recall that higher order HMMs can be converted to first order HMMs with more states §8.3). For simplicity, however, we assume that these processes have already been done so that the assumptions in the previous paragraph have no loss of generality.

In many applications, the states of an HMM have a natural ordering, and the transition matrix will obey this ordering by requiring that only strictly left-to-right transitions are possible. This occurs, for example, when modeling words in speech recognition applications where a word must consist of one of set of possible sequences through phonetic units. In speech, this occurs at many levels as well. I.e., there are sub-phones of a phone, phones of a word, words of a sentence, all of which is (mostly if not all) left to right. Transitions might be allowed only between successive states in the ordering (corresponding to a banded diagonal transition matrix) as shown in Figure 8.41b. This is appropriate, for example, when the HMM states represent one of the possible pronunciations of a given word in a speech utterance, and where to recognize (or match or generate) a particular pronunciation of a word one must visit each phone of the pronunciation. Alternatively, the model might also allow states to be skipped but still not violate the given order (an upper triangular transition matrix) as shown in Figure 8.41d. This might also correspond to a phone model of a word, but in this case multiple pronunciations are possible with different probabilities, and where phone deletions are possible (recall also Figure 8.20). In each of these cases, a transition is allowed from a current state to some latter state, and in the DTW transition grid, this corresponds to diagonal up-right links, as shown in Figures 8.41a and 8.41c. The right-going horizontal links in Figures 8.41a and 8.41c correspond in the HMM to staying at the same state for (at least) two successive time steps, and hence correspond to the self loops in Figures 8.41b and 8.41d.

HMMs states in general, on the other hand, need not follow any given strict sequential order. In many applications of HMMs where the goal is to model a periodic source, we might wish to allow cycles in the state transition graph, where each cycle corresponds to some periodic component in the source. In a DTW grid, this means that we might have downward right-going links, as shown in Figure 8.41e, whose corresponding state transition graph might be Figure 8.41f.

The path weights in DTW were encoded with $w^P(t) \in \mathbb{R}_+$ where P is a path and t is the relative position within this path. In general, the weight might depend on where the path is applied in the DTW grid, even meaning that if $w^P(t_0)$ were applied in two different locations in the grid, we might have different scores. In the DTW discussion above, however, the assumption has been that the weights are the same regardless of where the path is applied, so that if a path is applied in very different parts of the grid, the path weights would be the same. For example, in Figure 8.38, we see that the weights are used everywhere the path constraints are applied. Hence, it is possible to encode the path constraints using a form of transition matrix $W = [w_{ij}]_{ij}$ where $w_{ij} \geq 0$. This means that moving from state i to j costs w_{ij} regardless of where

in the grid such a transition occurs, and hence this is exactly analogous to a time-homogeneous assumption of § 8.3. Indeed, the horizontal axis of DTW corresponds to time.

Next, we consider the DTW costs encoded in the probabilistic framework. These costs encode preferences to stay in same state or move to next (or some other state). In speech, for example, vowels are typically much longer than constants, and states representing a vowel (much more so than say states representing a consonant) should express much more of a preference for staying put than moving on. Since transition costs in DTW are encoding using a cost matrix $W = [w_{ij}]_{ij}$, any transition between state i and j where $w_{ij} < \infty$ is shown in the state transition graph and any cost where $w_{ij} = \infty$ means there is no edge from i to j . A (first-order time-homogeneous) HMM requires stochastic transitions, and we can make this transformation using Equation (8.49). Hence, infinite cost gets mapped to zero probability.

Another mapping needs to be between the costs in DTW and an observation distribution in an HMM. The DTW cost is between any pair of element x_i and y_j and is computed using the function $d(x_i, y_j)$ as described in Equations (8.53) and (8.54). Note that if $d(x_i, y_j)$ is the Mahalanobis distance, then the transformation $\frac{1}{[2\pi A]^{1/2}} e^{\frac{1}{2}(x-y_j)^T A^{-1}(x-y_j)}$ results in a Gaussian density on x with mean y and covariance matrix A . That, if $p(x|y_q)$ is the observation distribution given state q , then $-\log p(x_i|y_q) + c = d(i, q)$ is the mapping between HMM observation probability and DTW cost. In this case, the “states” in the HMM select a Gaussian mean and covariance matrix, and the so DTW distance is seen as a weighted distance to the mean of a Gaussian. For example, at current frame, we have:

$$d(x, y_{q_i}) = -\log p(x|q_i) + c \quad (8.88)$$

or

$$p(x|q_i) = e^{-d(x, y_{q_i})+c} \quad (8.89)$$

So as the distance goes down, the probability goes up. Of course, this mapping can be used regardless of the form of the distance. For example, if $p(x_i|y_q)$ is a Gaussian mixture, or any normalized (so that $\int p(x|y_q)dx = 1$) score, then the mapping can be used as well to produce a distance.

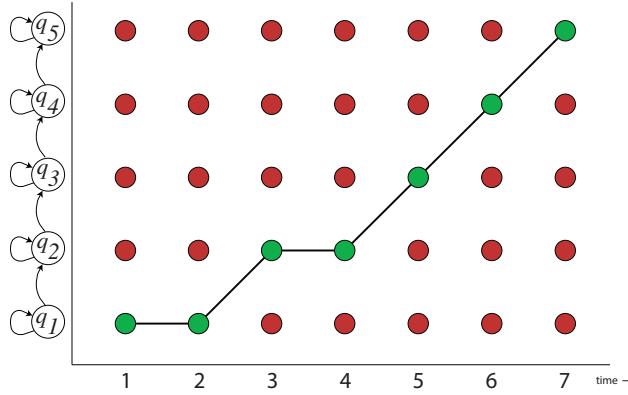


Figure 8.42: A path through a DTW grid is essentially the same as a path through a complete HMM trellis.

With the above two transformations, we can consider computing the “score” of a given path in a DTW vs. the score of a comparable HMM. Consider Figure 8.42. The first thing to note is that the DTW grid is essentially the same as the HMM trellis described in § 8.4.5. When viewed as an HMM, the probability of this path becomes:

$$\Pr(x_1, x_2, \dots, x_7, \text{path in Figure 8.42}) \quad (8.90)$$

$$= p(x_1|q_1)p(x_2|q_1)p(x_3|q_2)p(x_4|q_2)p(x_5|q_3)p(x_6|q_4)p(x_7|q_5)\pi_1 a_{11} a_{12} a_{22} a_{23} a_{34} a_{45} \quad (8.91)$$

where $\pi_i = \Pr(Q_1 = i)$ is the initial state probability.

The DTW score of the path in *Figure 8.42* is as follows:

$$d(x_1, q_1)w_{01} + d(x_2, q_1)w_{11} + d(x_3, q_2)w_{12} + d(x_4, q_2)w_{22} + d(x_5, q_3)w_{23} + d(x_5, q_4)w_{34} + d(x_7, q_5)w_{45} \quad (8.92)$$

where w_{01} is a preference for starting at state 1.

In order to compare the HMM and the DTW scores more directly, we use the following relationships between DTW score and HMM probability:

$$a_{ij} = \frac{e^{-w_{ij}}}{\sum_{j'} e^{-w_{ij'}}} \quad \text{and} \quad p(x|q_i) = e^{-d(x, y_{q_i})+c} \quad (8.93)$$

In the case of DTW, the score associated with this path is:

$$d_\phi(x, y) = \sum_{t=1}^T d(\phi_x(t), \phi_y(t))w_t^\phi \quad (8.94)$$

In the case of HMMs, the score (i.e., log probability) associated with this path is:

$$-\log \Pr(x_{1:T}, q_{1:T}) = -\sum_{t=1}^T d(x_t, y_{q_t}) + w_{q_t, q_{t+1}} + c \quad (8.95)$$

where c is a constant. Hence, we see that the scores are quite comparable in log space, in that the DTW distance is like the negative log probability of the HMM score. The key difference is that the path weights are incorporated multiplicatively in the case of DTW and additively in the case of the HMM's log score. So the two methods are similar but not identical. Which score is better will depend on the application. In DTW, more specifically, the weight has an interpretation as a slope penalty while in an HMM the “weight” has an interpretation as a state transition probability (HMMs use a stochastic matrix). Of course, HMMs also have a probabilistic interpretation that makes them perhaps more easily amenable to learning strategies (although when seen as purely numerical optimization, one could just as easily formulate optimization strategies that could find (or “learn”) the parameters for a DTW procedure as well).

We conclude with section by noting that we can easily obtain a DTW-like multiplicative factor, even in log space, within the log HMM transition scores when using a Jelinek HMM. This will be discussed in §??.

8.4.8 HMMs as smoothed (or regularized) unary potential functions

In many pattern classification tasks, an object (represented by an observed vector x) is classified to a vector or structured class y by a joint objective consisting of the sum of two terms. The first term is based on a decomposition of x into its constituent parts, and the term itself is separable in terms of these parts. This means that this first term itself consists of a sum of sub-terms. Each sub-term moreover can be seen as a joint score between a portion of x and some local decision. The second term is a smoothing prior, that does several things: 1) ensures that there is a preference for collections of local decisions that are jointly good in some way, and 2) precludes certain sets of local decisions from happening (sort of a hard constraint).

Perhaps the simplest example is that of image segmentation, where x is an image and y is a binary vector of the same size and dimensions as x . Each pixel of x needs to be given a binary label, and this is often done with a function based on just the pixel and perhaps the nearby pixels. If x_t is a pixel of x , then $\text{unary}(x_t, y_t)$ is a unary potential between the features that describe pixel x_t in an image and y_t is the corresponding label.

A simple but poor strategy would be to independently classify each pixel using only the unary potential functions, i.e.,

$$\sum_t \text{unary}(x_t, y_t) \quad (8.96)$$

Notice that the unaries are fully separable from each other (equivalently are fully decomposed, or independent) as they are represented as a sum. This means that making a decision about one unary (say that $y_t = 1$) does not at all influence the score in making a decision about some other unary (say that $y_\tau = 0$ vs $y_\tau = 1$ for $\tau \neq t$). Using only on the unaries for an image segmentation usually works quite poorly since: 1) there may be noise in x ; 2) the unaries might not be perfectly estimated; and 3) there might be constraints in the labeling of the pixels that the unaries alone might not necessarily follow.

In order to rectify these problems, a smoothing prior is used in addition to the unaries. The smoothing prior offers scores for sets of pixel labels alone, and prefers vectors y that have a low score. For example, in image processing, the smoothing prior might consist of a grid over pixels where pixels are connected to their immediately neighbors, and there can be some preference that says there is a preference for neighboring pixels to take on the same value.

If we mix the smoothing prior with the unaries additively, we get what is known as the energy function:

$$E(x, y) = \sum_t \text{unary}(x_t, y_t) + \lambda \text{smoothing prior}(y_{1:n}) \quad (8.97)$$

where λ is a tradeoff parameter. Setting $\lambda \leftarrow 0$ means use only the unary potentials, while setting λ large asks for a lot of smoothing. The energy function is usually used in a joint probability model of the form:

$$p(x, y) = \frac{1}{Z} \exp(-E(x, y)) \quad (8.98)$$

where Z is a normalizing constant known as the partition function. With this, we might instantiate the distribution with an unknown image \bar{x} and ask for $\text{argmax}_{y \in \{0,1\}^n} p(\bar{x}, y)$ as a candidate image segmentation.

It turns out that the above process is exactly analogous to HMMs for sequential/temporal models. The HMM's Markov chain can be seen as a smoothing prior over assignments to the hidden variables in an HMM. It acts as a form of "temporal coherency" that discourages inferring incoherent sequence that might otherwise be preferred by the unary potentials alone. It can also be seen as providing a soft score over the "grammar" of the HMM, where more grammatical assignments are given higher scores. The Markov chain expresses an assumption about how nature is smooth, and scores $y_{1:n}$ strings higher that are smoother according to the functional form of the prior. The prior might be an attempt to represent nature, meaning what is likely in the real world. For example, if decisions were made only based on the unary potentials, the decisions would be as follows:

$$\forall t, \quad y_t^* = \underset{y_t \in D_Y}{\text{argmax}} \text{unary}(x_t, y_t) \quad (8.99)$$

While these decisions would be optimal from the point of of the unary potentials, they would likely not respect the smoothness properties of the signal.

Consider an example from speech recognition. If, for example, D_Y corresponded to words and D_X corresponded to acoustics, then the unaries might make $y_t^* = \text{banana}$ and $y_{t+1}^* = \text{paramecium}$ but the smoothness priors (which is a language model in this case) might want to make it such that "paramecium" is not a word that may follow "banana". In the case of part-of-speech tagging, where D_X consists of words, and D_Y consists of tags of the words (such as "verb", "noun", and other higher-level units), then deciding the part of speech for the word "bank" based on the unary along would produce "noun" but if the word "bank" occurs in the context "the plane will bank before it landed", then the word "bank" is a verb and so

context counts. While it could be that the unary could use a larger context (so x_t really consists of a large context surrounding time t), a typical strategy to do this is to use a smoothing prior to ensure the right word sense in each case.

In general, the goal in inference is to come up with a assignment to the hidden variables that is a compromise between the assignments that are best for the unaries and that which is most smooth from the smoothing prior.

Now an HMM can be seen in the same way. Consider again the HMM's probability distribution:

$$\text{score}(x_{1:n}, y_{1:n}) = \log p(x_{1:n}, y_{1:n}) = \sum_t \log p(x_t|y_t) + \sum_t \log p(y_t|y_{t-1}) \quad (8.100)$$

Hence, the HMM using its standard Bayesian network decomposition can be seen as just smoothed unary potentials, where the unaries are precisely the HMM's observation distributions.

So far, we've seen the simplest case, but a smoothing prior might do more than smoothing. It might also help to transform the unary outputs y_i into still higher level variables. In fact, this is the way hierarchical models are formed. Lets say that $y_{1:n}$ are intermediary variables and $z_{1:n}$ are final output variables. We might form a multi-level HMM as in the following:

$$\sum_t \text{unaries}(x_t, y_t) + \text{smoothing prior}(y_{1:n}) + \sum_t \text{unaries}(y_t, z_t) + \text{smoothing prior}(z_{1:n}) \quad (8.101)$$

In one sense, this can be still seen as a normal HMM if it is the case that, say the variables in the original single-layer HMM y_t was really vector valued so that $y_t = (y_t^{(1)}, y_t^{(2)})$ and that smoothing prior($y_{1:n}$) asked for smoothness both over time between the same elements of the vectors at different times, and also over multiple elements of each vector at each time. In some cases, this is called a hierarchical HMM which we consider again in §8.9.2.

8.4.9 HMM as Graphical Models

In fact, HMMs are both dynamic Bayesian networks and dynamic graphical models as we will define them in, respectively, § 8.10 and § 8.11. But at this point we know about graphical models and Bayesian networks so lets jump from HMMs to them directly in this section.

In the case of a given fixed time length T , an HMM is a probability distribution over $2T$ variables: 1) a sequence of T discrete state variables $Q_{1:T}$ organized as a first order Markov chain; and 2) a set of T observed variables $X_{1:T}$ each conditionally independent of everything else when conditioned on its corresponding state variable. Figure 8.10-left shows the Bayesian network view of an HMM, where it is implied that the factors are locally normalized. That is, the locally normalized view of an HMM is any distribution $p(x_{1:n}, y_{1:n})$ that factories as follows:

$$p(x_{1:T}, y_{1:T}) = p(y_1)p(x_1|y_1)p(y_2|y_1) \prod_{t=2}^T p(x_t|y_t)p(y_t|y_{t-1}) \quad (8.102)$$

The undirected graphical model view of an HMM does not require the factors to be normalized, and hence can be written as:

$$p(x_{1:T}, y_{1:T}) = \frac{1}{Z} \prod_{t=1}^T \phi(y_t, y_{t-1})\phi(x_t, y_t) \quad (8.103)$$

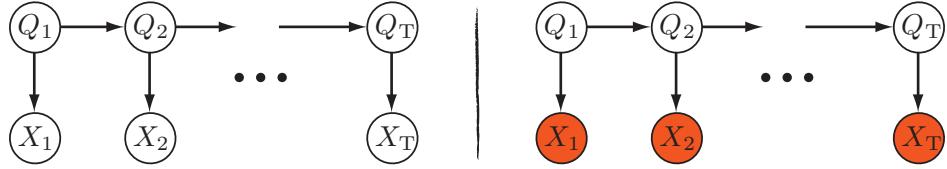


Figure 8.43: Viewing an HMM as a family of probability distributions, we say that for any $p \in \mathcal{F}(\text{HMM}, R)$, from the directed local Markov property and the HMM graph shown here, we can immediately write down the joint distribution as $p(x_{1:T}, q_{1:T}) = p(q_1)p(x_1|q_1)\prod_{t=2}^T p(x_t|q_t)p(q_t|q_{t-1}) = \prod_t p(x_t|q_t)p(q_t|q_{t-1})$.

where Z is the partition function

$$Z = \sum_{x_{1:T}, y_{1:T}} \prod_{t=1}^T \phi(y_t, y_{t-1})\phi(x_t, y_t), \quad (8.104)$$

and can be computed using dynamic programming (either the forward algorithm or the backward algorithm).

When the x variables are observed, $\bar{x}_{1:T}$ then there is a commonality between computing Z and computing $p(\bar{x}_{1:T})$. To compute Z in this case we sum only over the y variables as in:

$$Z = \sum_{x_{1:T}} \prod_{t=1}^T \phi(y_t, y_{t-1})\phi(x_t, y_t). \quad (8.105)$$

When we wish to compute $p(\bar{x}_{1:T})$ from Equation (8.102), we compute

$$p(x_{1:T}) = \sum_{y_{1:T}} \prod_{t=1}^T p(x_t|y_t)p(y_t|y_{t-1}) \quad (8.106)$$

Since the factorization into local potential functions is the same in Equation (8.102) and Equation (8.103), computing the partition function under the Markov random field form of an HMM is computationally equivalent to computing the probability of evidence (also called PE).

The family of models represented by Figure 8.10-left and Figure 8.10-right are in fact identical, as the local normalization constraints do not impose any further constraints on the family. To show this, note that going from the Bayesian network to the directed graphical model is easy since in that case $Z = 1$ and we just rename factors $\phi(y_t, y_{t-1}) \leftarrow p(y_t|y_{t-1})$ and $\phi(x_t, y_t) \leftarrow p(x_t|y_t)$. Going the other direction, from an HMM's undirected graphical model to a Bayesian network requires a bit more work, but it entails starting with $p(x_{1:n}, y_{1:n})$, forming all of the locally normalized factors of the Bayesian network ($p(x_t|y_t)$ and $p(y_t|y_{t-1})$ for all t) and then multiplying them together. Once this is done, it can be seen that we get back exactly the same distribution $p(x_{1:n}, y_{1:n})$ and hence nothing is lost. We derive these mappings in full in §8.8.2.

It is well known that for a time signal of length T , and under an HMM that has N states, the time complexity of HMM inference is $O(N^2T)$ and the memory complexity is $O(NT)$. This can be easily seen by looking at a triangulation of the HMM graph, one of which is shown in Figure 8.44-A. The observation variables \bar{x}_t only contribute multiplicatively to the HMM scores and not to its state space or complexity. It is therefore possible to simplify the HMM by defining factors $\phi'(q_{t-1}, q_t) \triangleq \phi(q_{t-1}, q_t)\phi(\bar{x}_t, q_t)$ yielding factorization $p(\bar{x}_{1:T}, q_{1:T}) \propto \prod_{t=1}^T \phi'(q_{t-1}, q_t)$. This is shown in Figure 8.44-B. The junction tree for this graphical model is shown in Figure 8.44-C and we see that the tree is just a sequential chain of repeated cliques and separators.

In junction tree, each clique has two variables and there are T such cliques leading to the time complexity $O(N^2T)$ of HMMs. On the other hand, the HMM memory complexity is only $O(NT)$ so this must mean

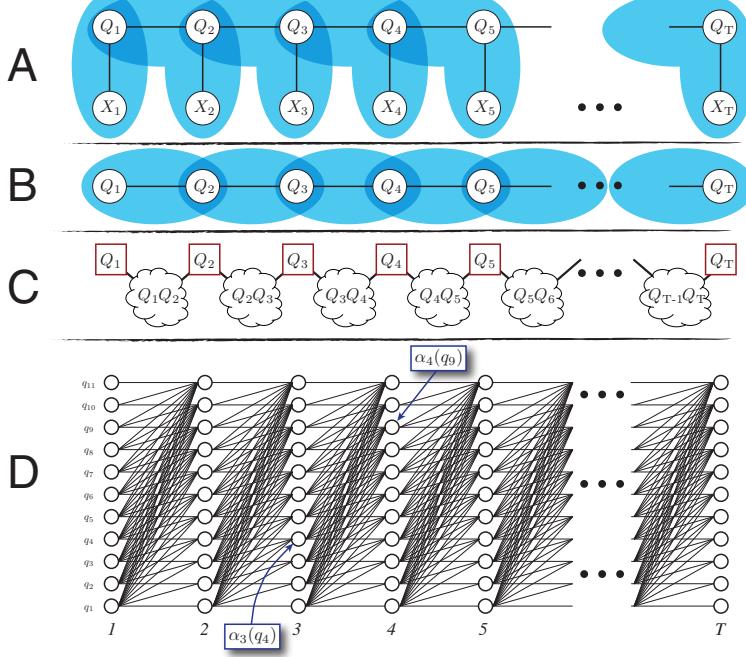


Figure 8.44: A: An HMM as a Markov random field along with the cliques associated with a triangulation of the HMM, where the cliques are depicted as blue regions. B: The graphical model for the effective state space of the HMM where the observations X_t have been removed since they do not contribute to the state space. C: a junction tree corresponding to the HMM, where each clique has two random variables (shown as cloud shapes) and each separator has one random variable (shown as squares). D: the trellis graph corresponding to the search space of the HMM, and also to the junction tree. The trellis is shown aligned so that the separators and cliques in the junction tree vertically line up with the columns or transitions of the trellis.

that the cliques are never stored, only the separators. Indeed, consider the standard forward (or alpha) recursion for HMMs:

$$\alpha_t(q) \triangleq p(\bar{x}_{1:t}, Q_t = q) = p(\bar{x}_t | q) \sum_r p(Q_t = q | Q_{t-1} = r) \alpha_{t-1}(r) \quad (8.107)$$

The forward recursion is often described using a rectangular trellis graph (not a graphical model) as shown in Figure 8.44-D. Each recursion $\{\alpha_t(q)\}_q$ corresponds to a column of nodes in this graph — in the figure, the elements corresponding to $\alpha_3(q_4)$ and $\alpha_4(q_9)$ are both marked with arrows.

From Figure 8.44, we see that the separators in the junction tree correspond to the columns of the trellis, while the cliques in the junction tree correspond to two successive columns (i.e., the transitions between columns). The reason for the $O(NT)$ memory usage is that the forward computation is only implicitly computing the clique expansions. Each set of computations for $\alpha_t(q)$ over all q corresponds to a clique *expansion* (i.e., to an enumeration of all of the values of the variables in the clique) and then immediate reduction. Since the cliques are never stored, it is not necessary to use $O(N^2)$ memory. Forward recursion thus corresponds to a form of message in the junction tree between successive separators. Of course, the computation is still $O(N^2T)$ since each $\alpha_t(q)$ requires $O(N^2)$ steps.

8.4.10 Rabiner vs. Jelinek HMMs as graphical models

The HMMs described in Figure 8.10 and again in Figure 8.44 are such that each observation is associated with a state random variable. Speaking generatively, there is one observation per individual state value. These therefore correspond to the Rabiner style of HMMs as described in §8.4.4.7.1.

It is also possible to use a graphical model to describe Jelinek HMMs as shown in Figure 8.45. Here,

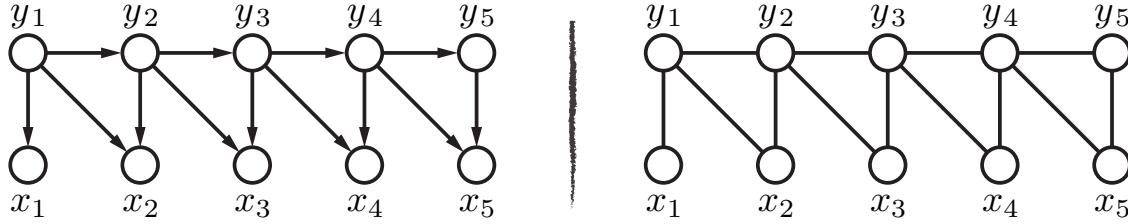


Figure 8.45: HMMs viewed as graphical models in the Jelinek/Mealy style. Here, each observation is a function of a transition between two states. Left is the dynamic Bayesian network view, and on the right is the equivalent Markov random field depiction. This figure should be contrasted with the “Rabiner/Moore” style of HMM having graphical models shown in Figure 8.10. It is important to note that the computational complexity $O(TN^2)$ is the same for this style of HMM as well as Figure 8.10. The reason is that the $\bar{x}_{1:T}$ variables are observed and so they do not contribute to the state space even though the maxclique size (of three) in this figure is larger than the maxclique size in Figure 8.10.

we see that each observation is a function of a pair of states, and the model therefore factors as:

$$p(x_{1:T}, y_{1:T}) = \prod_{t=1}^T p(x_t | y_t, y_{t-1}) p(y_t | y_{t-1}) \quad (8.108)$$

As mentioned in §8.4.4.6, Rabiner and Jelinek HMMs can be transformed into each other possibly with an increase in the number of states. It is also important that both Rabiner and Jelinek HMMs have computational complexity of $O(TN^2)$ even though the clique size of the Jelinek HMM is larger — one variable in each clique in a Jelinek HMM is observed so it does not contribute to the state space.

In a Jelinek HMM, since the observations are a function of a pair of state transitions, other scores besides the observation distribution can be easily incorporated as well. For example, in §8.4.7, we saw that DTW systems are comparable to log HMM scores, except in the DTW system there is a multiplicative score associated with a transition. If we have a factor of the form $\phi(x_t, q_t, q_{t-1})$ as shown in Figure 8.45-right, we can just as easily include a multiplicative factor there as well. In fact, this can even be done using the Rabiner HMM. In either case, we will have a factor involving two state variables and we ensure that $\phi(q_t, q_{t-1}) \propto e^{\omega(q_t, q_{t-1})}$ where $\omega(q_t, q_{t-1})$ is the desired multiplicative weight. Hence, we see that HMMs really are a strict generalization of the DTW described earlier.

8.4.11 HMM inference offline and online, the forward/backward algorithm, Viterbi, etc.

Probabilistic inference is the process of computing the probability of one set of random variables given values for some other random variables, an area that receives wide interest in machine learning and its applications.

HMM inference is particularly nice due to its strong factorization properties which allows dynamic programming to compute efficient solutions. There are two styles of inference, offline and online.

Offline inference consists of computing quantities such as the following. We are given a fixed length- T sequence of observations $x_{1:n}$ and we wish to compute quantities such as the following:

- The joint probability $p(\bar{x}_{1:T}, \bar{y}_{1:T})$ given values $\bar{x}_{1:T}, \bar{y}_{1:T}$. Since everything is observed, this is a simple process of multiplying the local scores together using either Equation (8.102) or Equation (8.103), where in this latter case we must also compute Z using dynamic programming via either the forward or the backwards algorithm.

- Computing the probability of the evidence $p(x_{1:n})$ or, as we saw above, the partition function Z . For this, we use the forward or the backwards algorithm.
- The Viterbi path $\text{argmax}_{y_{1:n}} p(x_{1:n}, y_{1:n})$, also known as the MPE (most probable estimate) or inference under the max-product semi-ring, or the highest probability configuration of random variables known as the maximum a posteriori (MAP) inference problem. Why this process has so many names is testament to its usefulness since it has been addressed independently in many different communities. A variant of this problem is finding the k -best random variable assignments rather than just the 1-best assignment.
- We might be interested in certain marginal probabilities that are useful for either parameter learning or for answering questions. Typical ones that are desired include $p(y_t | \bar{x}_{1:T})$ or $p(y_t, y_{t-1} | \bar{x}_{1:T})$.

8.4.11.1 The HMM Forward Algorithm and Collect Evidence

We next define the forward algorithm, also known as collect evidence and the α -recursion. This can be used to compute $p(x_{1:T})$ for any value $x_{1:T}$.

Consider the quantity $p(x_{1:t})$ for any t and then proceed as follows:

$$p(x_{1:t}, q_t, q_{t-1}) = p(x_{1:t-1}, q_{t-1}, x_t, q_t) \quad (8.109)$$

$$\stackrel{(A)}{=} p(x_t, q_t | x_{1:t-1}, q_{t-1}) p(x_{1:t-1}, q_{t-1}) \quad (8.110)$$

$$= p(x_t | q_t, x_{1:t-1}, q_{t-1}) p(q_t | x_{1:t-1}, q_{t-1}) p(x_{1:t-1}, q_{t-1}) \quad (8.111)$$

$$\stackrel{(B)}{=} p(x_t | q_t) p(q_t | q_{t-1}) p(x_{1:t-1}, q_{t-1}) \quad (8.112)$$

where (A) follows from the chain rule of probability and hence requires no assumptions, and (B) follows since in an HMM $X_t \perp\!\!\!\perp \{X_{1:t-1}, Q_{1:t-1}\} | Q_t$ and $Q_t \perp\!\!\!\perp \{X_{1:t-1}, Q_{1:t-2}\} | Q_{t-1}$

This yields,

$$p(x_{1:t}, q_t) = \sum_{q_{t-1}} p(x_{1:t}, q_t, q_{t-1}) \quad (8.113)$$

$$= \sum_{q_{t-1}} p(x_t | q_t) p(q_t | q_{t-1}) p(x_{1:t-1}, q_{t-1}) \quad (8.114)$$

If the following quantity is defined $\alpha_t(q) \triangleq p(x_{1:t}, Q_t = q)$, then the preceding equations imply that

$$\alpha_t(q) = p(x_t | Q_t = q) \sum_r p(Q_t = q | Q_{t-1} = r) \alpha_{t-1}(r) \quad (8.115)$$

This is the α (or forward) recursion for HMMs. It is also identical to message passing from left-to-right in the loopy belief propagation algorithm.

Thus, $p(x_{1:T}) = \sum_q \alpha_T(q)$, and the entire computation requires only $O(|D_Q|^2 T) = O(TN^2)$ operations where $N = |D_Q|$ is the number of states.

To derive this recursion, it was necessary to use only the fact that X_t was independent of its past given Q_t — in an HMM, X_t is also independent of the future given Q_t , but this was not yet used. This later assumption, however, is obligatory for the beta or backward recursion in HMMs as we will next see.

Exercise 124. Are there models where X_t is independent of its past but not its future given Q_t where such a forward recursion can be defined?

8.4.11.2 The HMM Backward Algorithm and Distribute Evidence

To derive the backward algorithm, also known as distribute evidence for HMMs, or the β -recursion, we consider the quantity $p(x_{t+1:T}|q_t)$ for any t and proceed as follows:

$$\begin{aligned} p(x_{t+1:T}|q_t) &= \sum_{q_{t+1}} p(q_{t+1}, x_{t+1}, x_{t+2:T}|q_t) \\ &\stackrel{(A)}{=} \sum_{q_{t+1}} p(x_{t+2:T}|q_{t+1}, x_{t+1}, q_t) p(x_{t+1}|q_{t+1}, q_t) p(q_{t+1}|q_t) \\ &\stackrel{(B)}{=} \sum_{q_{t+1}} p(x_{t+2:T}|q_{t+1}) p(x_{t+1}|q_{t+1}) p(q_{t+1}|q_t) \end{aligned}$$

where (A) follows from the chain rule probability, and (B) follows since $X_{t+2:T} \perp\!\!\!\perp \{X_{t+1}, Q_t\} | Q_{t+1}$ and $X_{t+1} \perp\!\!\!\perp Q_t | Q_{t+1}$.

Using the definition $\beta_q(t) \stackrel{\Delta}{=} p(x_{t+1:T}|Q_t = q)$, the above equations imply the beta-recursion

$$\beta_q(t) = \sum_r \beta_r(t+1) p(x_{t+1}|Q_{t+1} = r) p(Q_{t+1} = r|Q_t = q) \quad (8.116)$$

hence giving us another expression for the full probability

$$p(x_{1:T}) = \sum_q \beta_q(1) p(q) p(x_1|q). \quad (8.117)$$

Exercise 125. *The above derivations for the forward and backwards recursions were done using a Rabiner HMM. Show that analogous recursions can be derived for the Jelinek HMMs from Figure 8.45, and show that your recursions still have overall computational cost $O(TN^2)$.*

8.4.11.3 The α and β initializations

The alpha recursion initialization is fairly straightforward. At time $t = 1$, we have that $\alpha_t(q) = p(x_1, x_2, \dots, x_t, Q_t = q)$ and so when $t = 1$, we just start with $\alpha_1(q) = p(x_1, Q_1 = q) = p(x_1|Q_1 = q)p(Q_1 = q)$ which is the product of the observation distribution at time 1 (i.e., $p(x_1|Q_1 = q)$) which in our current case is the scalar score computed by the diagonal covariance Gaussian distribution at time 1 for state q , and the initial state distribution $p(Q_1 = q) = \pi_q$.

Note that we initialize the beta recursion as $\beta_q(T) = 1$ for all q . The initialization of the beta recursion might be a little bit harder to understand. Basically, we have that $\beta_t(q) = p(x_{t+1}, x_{t+2}, \dots, x_T|Q_t = q)$. So the question is, since this is a backwards recursion, we need something to initialize it to when $t = T$. Here, $\beta_T(q) = p(\text{something}|Q_t = q)$ and the question is, what should go in the something part. Firstly, by convention, whenever we have a range of the form $t + 1, t + 2, \dots, T$, and then we set t to be something where $(t + 1) > T$, then the range is empty. This means that $\beta_T(q) = p(\text{empty}|Q_t = q)$, but this answers a question is a question. What does “empty” mean? “empty” in this case is the empty event, it is something that isn’t there. To compute this quantity lets use conditional probability. I.e., for events A and B , $p(A|B) = p(A \cup B)/p(B)$. When A is the empty event, $p(A|B) = p(A \cup B)/p(B) = p(B)/p(B)$ (and this holds since $A \cup B = B$ when $A = \emptyset$). Therefore, $\beta_T(q) = p(\text{empty}|Q_t = q) = p(\text{empty}, Q_t = q)/p(Q_t = q) = p(Q_t = q)/p(Q_t = q) = 1$ (this assumes none of the probs are zero). To summarize, the initialization of the backwards (beta) recursion starts with $\beta_T(q) = 1$ for all q .

8.4.11.4 The HMM Probability of Evidence

The HMM's probability may be computed using a combination of the alpha and beta values for **any** t since:

$$\begin{aligned} p(x_{1:T}) &= \sum_{q_t} p(q_t, x_{1:t}, x_{t+1:T}) = \sum_{q_t} p(x_{t+1:T}|q_t, x_{1:t})p(q_t, x_{1:t}) \\ &\stackrel{(A)}{=} \sum_{q_t} p(x_{t+1:T}|q_t)p(q_t, x_{1:t}) = \sum_{q_t} \beta_{q_t}(t)\alpha_t(q_t) \end{aligned}$$

where (A) follows since $X_{t+1:T} \perp\!\!\!\perp X_{1:t}|Q_t$ in an HMM.

8.4.11.5 The HMM Probabilistic Queries

The quantities we typically wish to compute for an HMM include:

- Compute $p(q_t|x_{1:t})$, or the *filtering problem*
- Compute $p(q_t|x_{1:s})$, with $t > s$, or the *prediction problem*.
- Compute $p(q_t|x_{1:u})$, with $t < u$, or the *smoothing problem*.
- Above three named from linear systems literature from Electrical Engineering (e.g., from Kalman filters).
- Note, above queries includes $p(q_t|x_{1:T})$ which is important for learning.
- Another needed query is $p(q_t, q_{t+1}|x_{r:s})$, and often this is for $r = 1$ and $s = T$, as mentioned above.
- In all above cases, we need to sum out hidden variables from joint distributions. E.g., $p(q_t|x_{1:T}) = p(q_t, x_{1:T})/p(x_{1:T})$ and $p(q_t, q_{t-1}|x_{1:T}) = p(q_t, q_{t-1}, x_{1:T})/p(x_{1:T})$ so we also need $p(x_{1:T})$ which, as mentioned above, can be computed using either the forward or the backward recursion. Hence, for many queries we will need to compute both the numerator and denominator.

Recall that in the HMM's homogeneous case, we have that $P(Q_t = i|Q_{t-1} = j) = a_{ij}$ or $[A]_{ij}$ is a first-order time-homogeneous transition matrix where $P(Q_1 = i) = \pi_i$ is the initial state distribution, and $P(X_t = x|Q_t = i) = b_i(x)$ is the observation distribution for the current state being in configuration i . Hence, there are a fixed number of parameters regardless of the length T . In other words, parameters are **shared** across all time. This is a property that is often true for many dynamic graphical models.

8.4.11.6 The HMM Probabilistic Queries and Parameter Learning

Now given this fixed set of parameters in the time-homogeneous HMM case, what probabilistic queries would we need to learn them? This very well may depend on the learning method, so let's consider the EM algorithm first.

Let the the variables $X_{1:T} = \bar{x}_{1:T}$ be observed and $Q_{1:T}$ be the hidden Markov chain. Let λ designate the full set of parameters that we wish to learn, and let λ^p be some previous parameters that we have obtained from somewhere. The EM algorithm optimizes an objective that is parameterized by both the previous parameters and the set of parameters we wish to optimize over. Indeed, the EM algorithm repeatedly optimizes the following objective:

$$f(\lambda) = Q(\lambda, \lambda^p) = E_{p(x_{1:T}, q_{1:T} | \lambda^p)} [\log p(x_{1:T}, q_{1:T} | \lambda)] \quad (8.118)$$

$$= E_p [\log \prod_t p(q_t | q_{t-1}, \lambda) p(x_t | q_t, \lambda)] \quad (8.119)$$

$$= E_p [\sum_t \log p(q_t | q_{t-1}, \lambda) + \sum_t \log p(x_t | q_t, \lambda)] \quad (8.120)$$

$$\begin{aligned} &= \sum_t \sum_{ij} p(Q_t = j, Q_{t-1} = i | x_{1:T}, \lambda^p) \log p(Q_t = j | Q_{t-1} = i, \lambda) \\ &\quad + \sum_t \sum_i p(Q_t = i | x_{1:T}, \lambda^p) \log p(x_t | Q_t = i, \lambda) \end{aligned} \quad (8.121)$$

So this means that for EM learning, we need **for all** t , the queries $p(Q_t = i | x_{1:T})$ and $p(Q_t = j, Q_{t-1} = i | x_{1:T})$ in an HMM. Note that these are clique posteriors for the triangulation shown in Figure 8.44. Indeed, in graphical model inference, what one typically needs for learning is posteriors over only the cliques in the graph and nothing more, and an HMM is no exception to this rule.

But of course, the EM algorithm isn't the only way to learn parameters and some would argue even that it can be a quite poor way, depending on your application. For example, discriminative training algorithms might optimize the parameters of an HMM using an objective that is based on how well the HMM classifies data in the training set. Such methods are often in need of gradients of the HMM probability with respect to the parameters.

Hence, suppose we wanted to use a gradient descent like algorithm on $f(\lambda) = \log p(x_{1:T} | \lambda)$, as in

$$\frac{\partial}{\partial \lambda} f(\lambda) = \frac{\partial}{\partial \lambda} \log p(x_{1:T} | \lambda) = \frac{\partial}{\partial \lambda} \log \sum_{q_{1:T}} p(x_{1:T}, q_{1:T} | \lambda) \quad (8.122)$$

$$= \frac{\frac{\partial}{\partial \lambda} \sum_{q_{1:T}} p(x_{1:T}, q_{1:T} | \lambda)}{\sum_{q_{1:T}} p(x_{1:T}, q_{1:T} | \lambda)} = \frac{\frac{\partial}{\partial \lambda} \sum_{q_{1:T}} p(x_{1:T}, q_{1:T} | \lambda)}{p(x_{1:T} | \lambda)} \quad (8.123)$$

Say we're interested in $\partial / \partial a_{ij}$. Lets expand the numerator in Equation (8.123):

$$\text{numerator} = \frac{\partial}{\partial a_{ij}} \sum_{q_{1:T}} p(x_{1:T}, q_{1:T} | \lambda) = \frac{\partial}{\partial a_{ij}} \sum_{q_{1:T}} \prod_t p(x_t | q_t) p(q_t | q_{t-1}) \quad (8.124)$$

Define $\mathcal{T}_{ij}(q_{1:T}) \triangleq \{t : q_{t-1} = i, q_t = j\}$ in the following:

$$\text{numerator} = \frac{\partial}{\partial a_{ij}} \sum_{q_{1:T}} \prod_t p(x_t | q_t) \prod_{t \in \mathcal{T}_{ij}(q_{1:T})} a_{ij} \prod_{t \notin \mathcal{T}_{ij}(q_{1:T})} p(q_t | q_{t-1}) \quad (8.125)$$

We get

$$\text{numerator} = \sum_{q_{1:T}} \prod_t p(x_t|q_t) \frac{\partial}{\partial a_{ij}} a_{ij}^{|\mathcal{T}_{ij}(q_{1:T})|} \prod_{t \notin \mathcal{T}_{ij}(q_{1:T})} p(q_t|q_{t-1}) \quad (8.126)$$

$$= \sum_{q_{1:T}} \prod_t p(x_t|q_t) |\mathcal{T}_{ij}(q_{1:T})| a_{ij}^{|\mathcal{T}_{ij}(q_{1:T})|-1} \prod_{t \notin \mathcal{T}_{ij}(q_{1:T})} p(q_t|q_{t-1}) \quad (8.127)$$

$$= \sum_{q_{1:T}} \prod_t p(x_t|q_t) p(q_t|q_{t-1}) \frac{|\mathcal{T}_{ij}(q_{1:T})|}{a_{ij}} = \sum_{q_{1:T}} p(x_{1:T}, q_{1:T}) \frac{|\mathcal{T}_{ij}(q_{1:T})|}{a_{ij}} \quad (8.128)$$

$$= \frac{1}{a_{ij}} \sum_{q_{1:T}} p(x_{1:T}, q_{1:T}) \sum_t \mathbf{1}\{q_{t-1} = i, q_t = j\} \quad (8.129)$$

$$= \frac{1}{a_{ij}} \sum_t \sum_{q_{1:T}} p(x_{1:T}, q_{1:T}) \mathbf{1}\{q_{t-1} = i, q_t = j\} \quad (8.130)$$

$$= \frac{1}{a_{ij}} \sum_t p(x_{1:T}, q_{t-1} = i, q_t = j) \quad (8.131)$$

This allows us to express the gradient as:

$$\frac{\partial}{\partial \lambda} f(\lambda) = \frac{\frac{\partial}{\partial \lambda} \sum_{q_{1:T}} p(x_{1:T}, q_{1:T} | \lambda)}{p(x_{1:T} | \lambda)} = \frac{1}{a_{ij}} \frac{\sum_t p(x_{1:T}, q_{t-1} = i, q_t = j)}{p(x_{1:T} | \lambda)} \quad (8.132)$$

$$= \frac{1}{a_{ij}} \sum_t p(q_{t-1} = i, q_t = j | x_{1:T}) \quad (8.133)$$

Hence, this means that, like with the EM algorithm, for gradient descent learning, we also need, for all t , the queries $p(Q_t = j, Q_{t-1} = i | x_{1:T})$ from the HMM. A similar analysis shows that we also need $\forall t p(Q_t = i | x_{1:T})$. These are also needed when performing discriminative training. So clique posteriors are fundamental, we must have a procedure that can produce them efficiently. We'll revisit this subject in §8.4.11.9.

8.4.11.7 The HMM forward/backward recursions vs. the graphical model elimination algorithm

The HMM forward recursion algorithm is, unsurprisingly, just the graphical model elimination algorithm run on the HMM graph. First, we must choose an elimination order that generates this algorithm and we choose order: $X_1, X_2, Q_1, X_3, Q_2, X_4, Q_3, X_5, \dots, X_T, Q_{T-1}, Q_T$.

We treat evidence (or observed values) as delta functions as we discussed in §. That is, whenever $x_1 = \bar{x}_1$ we multiply in the Dirac delta function $\delta(x_1, \bar{x}_1)$ to the HMM probability equation. When $x_2 = \bar{x}_2$, we multiply in $\delta(x_2, \bar{x}_2)$, and so on. We note again that this is done for mathematical description, we would never really do summations like the below over such a large number of known zero values. We proceed with the elimination algorithm as follows, were we sum out the first variable x_1 :

$$\dots \sum_{x_4} \sum_{q_2} \sum_{x_3} \sum_{q_1} \sum_{x_2} \sum_{x_1} \prod_{t=1}^T p(x_t|q_t) p(q_t|q_{t-1}) \delta(x_t, \bar{x}_t) \quad (8.134)$$

$$= \dots \sum_{x_3} \sum_{q_1} \sum_{x_2} \left(\prod_{t=2}^T p(x_t|q_t) p(q_t|q_{t-1}) \delta(x_t, \bar{x}_t) \right) \underbrace{\sum_{x_1} p(x_1|q_1) \delta(x_1, \bar{x}_1) p(q_1)}_{p(\bar{x}_1|q_1)p(q_1) \stackrel{\Delta}{=} \alpha_1(q_1)} \quad (8.135)$$

This produces a new factor $\alpha_1(q_1)$ which we have, not coincidentally, named α like in the α -recursion of §8.4.11.1.

We next eliminate x_2 and then q_1 in:

$$\dots \sum_{x_3} \sum_{q_1} \left(\prod_{t=3}^T p(x_t|q_t) p(q_t|q_{t-1}) \delta(x_t, \bar{x}_t) \right) \sum_{x_2} p(x_2|q_2) p(q_2|q_1) \delta(x_2, \bar{x}_2) \alpha_1(q_1) \quad (8.136)$$

$$= \dots \sum_{x_4} \sum_{q_2} \sum_{x_3} \left(\prod_{t=3}^T p(x_t|q_t) p(q_t|q_{t-1}) \delta(x_t, \bar{x}_t) \right) \underbrace{\sum_{q_1} p(q_2|q_1) p(\bar{x}_2|q_2) \alpha_1(q_1)}_{\alpha_2(q_2)} \quad (8.137)$$

producing new factor $\alpha_2(q_2)$. We continue in this fashion, and iteration τ is as follows:

$$= \dots \quad (8.138)$$

$$= \dots \sum_{x_{\tau+2}} \sum_{q_{\tau}} \sum_{x_{\tau+1}} \left(\prod_{t=\tau+1}^T p(x_t|q_t) p(q_t|q_{t-1}) \delta(x_t, \bar{x}_t) \right) \underbrace{\sum_{q_{\tau-1}} p(q_{\tau}|q_{\tau-1}) p(\bar{x}_{\tau}|q_{\tau}) \alpha_{\tau-1}(q_{\tau-1})}_{\alpha_{\tau}(q_{\tau})} \quad (8.139)$$

It should be clear at this point that we are just redefining the α -recursion via the graphical model elimination algorithm:

$$\alpha_{t+1}(j) = \sum_i \alpha_t(i) p(Q_{t+1} = j | Q_t = i) p(\bar{x}_{t+1} | Q_{t+1} = j) \quad (8.140)$$

and

$$\alpha_1(j) = p(Q_1 = j) p(\bar{x}_1 | Q_1 = j) \quad (8.141)$$

We have that $\alpha_1(Q_1 = j) = p(Q_1 = j) p(\bar{x}_1 | Q_1 = j) = p(\bar{x}_1, Q_1 = j)$, and $\alpha_1(q_1) = p(\bar{x}_1, q_1)$. Also, $\alpha_2(q_2) = \sum_{q_1} p(q_2|q_1) p(\bar{x}_2|q_2) \alpha_1(q_1) = \sum_{q_1} p(q_2|q_1) p(\bar{x}_2|q_2) p(\bar{x}_1, q_1) = \sum_{q_1} p(q_1, q_2, \bar{x}_1, \bar{x}_2) = p(\bar{x}_1, \bar{x}_2, q_2)$. Like before, we see that the forward (α) recursion has the following probabilistic interpretation: $\alpha_t(j) = p(x_{1:t}, Q_t = j)$. Hence, the α (forward) recursion is just an instance of the elimination algorithm run on the GM for the HMM graph.

But the elimination order we just used and that lead to the α -recursion is only one of the exponentially many elimination orders we could use. Different elimination orders lead to different inference algorithms. For example, consider elimination order $X_T, Q_T, X_{T-1}, Q_{T-1}, \dots$ in the following:

$$\dots \sum_{x_{T-2}} \sum_{q_{T-1}} \sum_{x_{T-1}} \sum_{q_T} \sum_{x_T} \prod_{t=1}^T p(x_t|q_t) p(q_t|q_{t-1}) \delta(x_t, \bar{x}_t) \quad (8.142)$$

$$= \dots \sum_{x_{T-1}} \sum_{q_T} \left(\prod_{t=1}^{T-1} p(x_t|q_t) p(q_t|q_{t-1}) \delta(x_t, \bar{x}_t) \right) \sum_{x_T} p(x_T|q_T) p(q_T|q_{T-1}) \delta(x_T, \bar{x}_T) \quad (8.143)$$

$$= \dots \sum_{x_{T-1}} \sum_{q_T} \left(\prod_{t=1}^{T-1} p(x_t|q_t) p(q_t|q_{t-1}) \delta(x_t, \bar{x}_t) \right) p(\bar{x}_T|q_T) p(q_T|q_{T-1}) \quad (8.144)$$

I.e., eliminating x_T leads to instantiating the evidence, and then eliminating q_T leads to the first term of the (backwards) β -recursion.

$$\dots \sum_{x_{T-1}} \sum_{q_T} \left(\prod_{t=1}^{T-1} p(x_t|q_t) p(q_t|q_{t-1}) \delta(x_t, \bar{x}_t) \right) p(\bar{x}_T|q_T) p(q_T|q_{T-1}) \underbrace{1}_{\beta_T(q_T)} \quad (8.145)$$

$$= \dots \sum_{q_{T-1}} \sum_{x_{T-1}} \left(\prod_{t=1}^{T-1} p(x_t|q_t) p(q_t|q_{t-1}) \delta(x_t, \bar{x}_t) \right) \underbrace{\sum_{q_T} p(\bar{x}_T|q_T) p(q_T|q_{T-1}) \beta_T(q_T)}_{\beta_{T-1}(q_{T-1})} \quad (8.146)$$

And continuing with this elimination order, we get the standard β recursion:

$$\beta_t(q_t) = \sum_{q_{t+1}} \beta_{t+1}(q_{t+1}) p(q_{t+1}|q_t) p(\bar{x}_{t+1}|q_{t+1}) \quad (8.147)$$

with

$$\beta_T(j) = 1 \quad \forall j \quad (8.148)$$

Note again, we can get $p(x_{1:t})$ in many different ways:

$$\begin{aligned} p(x_{1:T}) &= \sum_{q_t} p(x_{1:T}, q_t) \\ &= \sum_{q_t} p(x_{1:t}, x_{t+1:T}, q_t) \\ &= \sum_{q_t} p(x_{t+1:T}|q_t, x_{1:t}) p(q_t, x_{1:t}) \\ &= \sum_{q_t} p(x_{t+1:T}|q_t) p(q_t, x_{1:t}) \\ &= \sum_{q_t} \beta_t(q_t) \alpha_t(q_t) \end{aligned}$$

So this means that for any t , we can get $p(x_{1:T})$ by using the corresponding $\alpha_t(i)$ and $\beta_t(i)$ quantities, for **any** t . How do we choose which one to use? The reason for each is primarily finite-precision arithmetic (i.e., numerical) related and to a smaller extent speed related. For example, if we used $p(x_{1:T}) = \sum_{q_T} \beta_t(q_T) \alpha_t(q_T)$ for all time, this would be a bit faster (since the normalization could be computed once) but on the other hand the posteriors $p(q_t|\bar{x}_{1:T})$ might not be perfectly numerically normalized. This issue in fact comes up with any dynamic graphical model, and GMTK has options to allow for both options (cf. §).

Hence, we have derived both the HMM forward recursion and the HMM backward recursion as variable elimination orders. These are summarized in Figure 8.46. Moreover, since an HMM is a tree, there are no additional fill-in edges added via the two elimination orders we have chosen, and hence both of these orders are optimal.

Now suppose we eliminated only the hidden variables without also eliminating the x variables. For example, if we did elimination using the order (Q_1, Q_2, \dots, Q_T) . This would then lead to fill-in edges as shown in Figure 8.47. Note, however, that even this elimination order would still have the same complexity $O(TN^2)$, the reason being that the observations (given that they are observed) do not increase the state space of any clique within which they live.

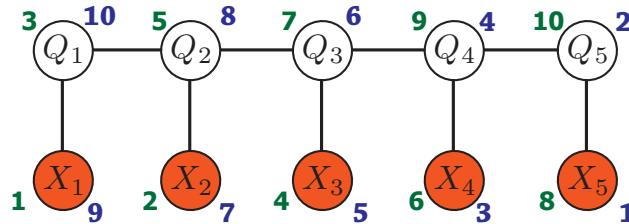


Figure 8.46: An HMM as a graphical model where different variable elimination orders result in different HMM recursions. The green numbers give the order for the α -recursion, and the blue numbers give the order for the β -recursion.

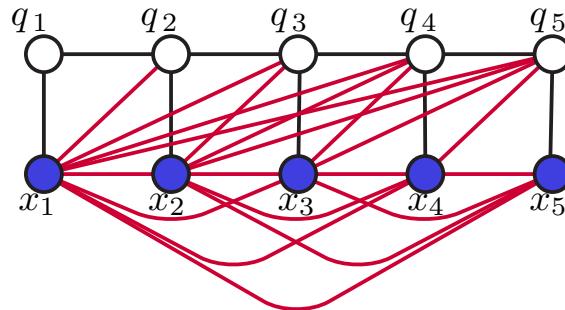


Figure 8.47: Eliminating only the variables Q_1, Q_2, \dots, Q_T in an HMM for $T = 5$. Fill-in edges are shown as red.

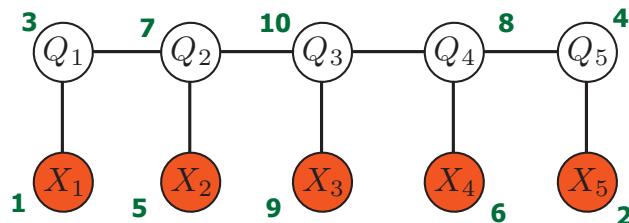


Figure 8.48: Outside-inside elimination order for HMMs.

There are other elimination orders as well that could be used for HMMs. Other orders might produce other recursions. An *outside-inside* algorithm of sorts can be defined (normally inside-outside is used for computing marginals for stochastic context-free grammars) via the elimination order shown in Figure 8.48. This order is particularly attractive since it means that inference can proceed starting from both the left-end and the right-end of the HMM simultaneously and can proceed independently (without communication) until the orders meet in the middle. This then gives an embarrassingly parallel factor of two speedup for HMMs.

Exercise 126. Derive the equations and a recursion for the elimination order shown in Figure 8.48. How is it different than standard forward/backward?

8.4.11.8 HMM filtering, smoothing, & prediction as elimination

Recall from §8.4.11.5 that the filtering HMM query is $p(q_t|x_{1:t})$ for all t . If we need this for one particular t , this is identical to one of the posteriors we already have $p(q_T|x_{1:T})$. More likely, however, we need recursion to get from having this quantity at time t to producing it at time $t + 1$. Note that this is obtained immediately from α -recursion, since $\alpha_q(t) = p(x_{1:t}, Q_t = q)$, we have

$$p(q_t|x_{1:t}) = \frac{\alpha_{q_t}(t)}{\sum_{q'_t} \alpha_{q'_t}(t)} \quad (8.149)$$

Hence, the normalized α s are just the filtering queries.

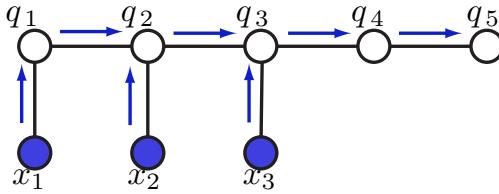


Figure 8.49: Elimination (or belief propagation) messages for the HMM prediction problem for computing $p(q_5|x_{1:3})$.

The HMM prediction (extrapolation) problem is only a little bit more complex. Here, we wish to compute $p(q_t|x_{1:s})$, with $t > s$. This, however, is also easily obtained from the α s values since we have

$$p(q_t, x_{1:s}) = \sum_{q_s, q_{s+1}, \dots, q_{t-1}} p(q_t, q_{t-1}, \dots, q_{s+1}, q_s, x_{1:s}) \quad (8.150)$$

$$= \sum_{q_s, q_{s+1}, \dots, q_{t-1}} p(q_t|q_{t-1}) p(q_{t-1}|q_{t-2}) \dots p(q_{s+1}|q_s) \alpha_{q_s}(s) \quad (8.151)$$

$$= \sum_{q_{t-1}} p(q_t|q_{t-1}) \sum_{q_{t-2}} (q_{t-1}|q_{t-2}) \dots \sum_{q_s} p(q_{s+1}|q_s) \alpha_{q_s}(s) \quad (8.152)$$

Thus, HMM prediction is like running elimination (or message passing) on a graph where some of the observations are removed, as shown in Figure 8.49.

The HMM smoothing problem is defined as $p(q_t|x_{1:u})$, with $t < u$. Like in the case of filtering, if we need this for one particular t , this is identical to one of the posteriors we already have, namely $p(q_t|x_{1:T})$. Otherwise, we might need to update $p(q_t|x_{1:u})$ for each $t < u$ as u increases (observations come in). This scenario is shown in Figure 8.50. Note that this involves re-sending messages (red). Each step requires $O(T)$ additional messages, so in total an $O(T^2)$ computation.

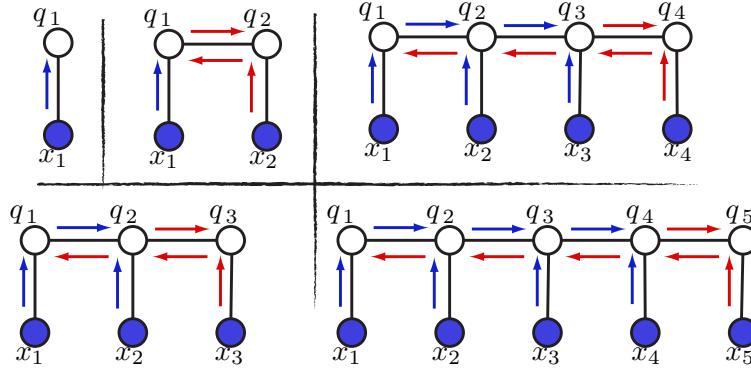


Figure 8.50: Messages needed for HMM smoothing. On the top left, we have the initial smoothed value. The second from the left on top shows the initial set of backwards messages needed. The bottom right shows the third step — note that since the third observation has come in at this point, we need to send messages $q_2 \rightarrow q_1$ again (shown in red), since it carries information from x_3 . The forth stage is shown in the upper right and the fifth in the lower right, again noting that message are needing to be sent again since new information, in the form of new observations, have arrived.

8.4.11.9 HMM clique posteriors

As mentioned in S8.4.11.6, we want to be able to compute more than just the probability of evidence $p(\bar{x}_{1:T})$. Indeed, we also need the clique posteriors $\gamma_t(i) = p(Q_t = i|x)$ and $\xi_t(i, j) = p(Q_{t-1} = i, Q_t = j|x)$. We can also get these from α and β quantities as follows:

$$\gamma_t(q_t) = p(q_t|x_{1:T}) = p(x_{1:T}|q_t)p(q_t)/p(x_{1:T}) \quad (8.153)$$

$$= p(x_{1:t}, x_{t+1:T}|q_t)p(q_t)/p(x_{1:T}) \quad (8.154)$$

$$= p(x_{t+1:T}|q_t, x_{1:t})p(x_{1:t}|q_t)p(q_t)/p(x_{1:T}) \quad (8.155)$$

$$= p(x_{1:t}, q_t)p(x_{t+1:T}|q_t)/p(x) \quad (8.156)$$

$$= \alpha(q_t)\beta(q_t)/p(x) \quad (8.157)$$

$$= \alpha(q_t)\beta(q_t)/\sum_{q'_t} \alpha(q'_t)\beta(q'_t) \quad (8.158)$$

We also want the posterior $p(y_t, y_{t-1}|\bar{x}_{1:T})$ for all values of t . This can also be obtained from the α and β recursions in the following procedure. First note that $p(y_t, y_{t-1}|\bar{x}_{1:T}) = p(y_t, y_{t-1}, \bar{x}_{1:T})/p(\bar{x}_{1:T})$ and we already know how to get $p(\bar{x}_{1:T})$, so we concentrate on finding $p(y_t, y_{t-1}, \bar{x}_{1:T})$.

$$p(y_t, y_{t-1}, \bar{x}_{1:T}) = p(x_t|q_t, q_{t-1}, q_t, x_{1:t-1}, x_{t+1:T})p(q_{t-1}, q_t, x_{1:t-1}, x_{t+1:T}) \quad (8.159)$$

$$\stackrel{(a)}{=} p(x_t|q_t)p(q_{t-1}, q_t, x_{1:t-1}, x_{t+1:T}) \quad (8.160)$$

$$= p(x_t|q_t)p(x_{t+1:T}|q_t, q_{t-1}, x_{1:t-1})p(q_t, q_{t-1}, x_{1:t-1}) \quad (8.161)$$

$$\stackrel{(b)}{=} p(x_t|q_t)p(x_{t+1:T}|q_t)p(q_t, q_{t-1}, x_{1:t-1}) \quad (8.162)$$

$$= p(x_t|q_t)\beta_t(q_t)p(q_t|q_{t-1}, x_{1:t-1})p(q_{t-1}, x_{1:t-1}) \quad (8.163)$$

$$\stackrel{(c)}{=} p(x_t|q_t)\beta_t(q_t)p(q_t|q_{t-1})\alpha_{t-1}(q_{t-1}) \quad (8.164)$$

where (a) follows since in an HMM X_t is independent of all else given Q_t , (b) follows since $X_{t+1:T} \perp\!\!\!\perp Q_{t-1}, X_{1:t-1}|Q_t$, and (c) follows since $Q_t \perp\!\!\!\perp X_{1:t-1}|Q_{t-1}$ and by the definitions of the α and β terms. Hence, all of the edge marginals can be computed using the standard recursions.

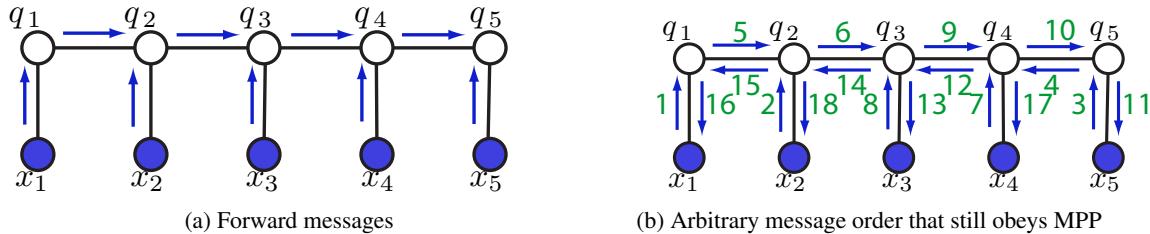


Figure 8.51: Messages corresponding to the forward procedure. Right, messages that are numbered (in green) and that obey the message passing protocol (MPP) but that are otherwise arbitrary, and correspond to an inference algorithm for HMMs that is neither forward nor backwards (nor apparently useful for anything other than pedagogy).

We want to be able to $\gamma_t(i)$ for all t rather than just a particular t . How best should this be done? One possible strategy is highly suboptimal and would proceed as in the following algorithm:

```

1 for  $t = 1 \dots T$  do
2   Compute  $\alpha_t(j)$  starting at time 1 up to time  $t$ 
3   Compute  $\beta_t(j)$  starting at time  $T$  down to time  $t$ 
4   Compute  $\gamma_t(j)$ 
5   Use  $\gamma_t(j)$  as needed (e.g., for parameter learning).

```

But this would be extremely wasteful. Once we have computed $\alpha_t(j)$, for time t , we should retain it for the next time step, $\alpha_{t+1}(j)$. Similarly, once we have $\beta_t(j)$ save it for previous time $\beta_{t-1}(j)$. Indeed, this is the idea of dynamic programming's optimal substructure & common subproblems: each previous time step of $\alpha_t(j)$ is both optimal and common for computation of the next time step $\alpha_{t+1}(j)$.

In fact, this is even more obvious when viewed as messages in the HMM's graphical model in that messages do not proceed until they have received appropriate incoming messages.

Recall the generic message definition for an arbitrary $p \in \mathcal{F}(G, R)$

$$\mu_{i \rightarrow j}(x_j) = \sum_{x_i} \psi_{i,j}(x_i, x_j) \prod_{k \in \delta(i) \setminus \{j\}} \mu_{k \rightarrow i}(x_i) \quad (8.165)$$

If graph is a tree, and if we obey the message-passing protocol order, then we will reach a point where all clique potentials are marginals. I.e., where

$$p(x_i) \propto \prod_{j \in \delta(i)} \mu_{j \rightarrow i}(x_i) \quad (8.166)$$

and where

$$p(x_i, x_j) \propto \psi_{i,j}(x_i, x_j) \prod_{k \in \delta(i) \setminus \{j\}} \mu_{k \rightarrow i}(x_i) \prod_{\ell \in \delta(j) \setminus \{i\}} \mu_{\ell \rightarrow j}(x_j) M \quad (8.167)$$

Now belief propagation messages on an HMM are straightforward, since an HMM is a tree. Hence, we can use any message order that obeys the message-passing protocol in order to produce potentials that are marginals. A few examples are shown in Figure 8.51.

Any of the message orders that we have considered above can be viewed using the junction tree algorithm operating on a junction tree with cliques and separators. The variety here depends on how the HMM is turned into a junction tree. For example, one solution has a junction tree separator for each HMM hidden variable.

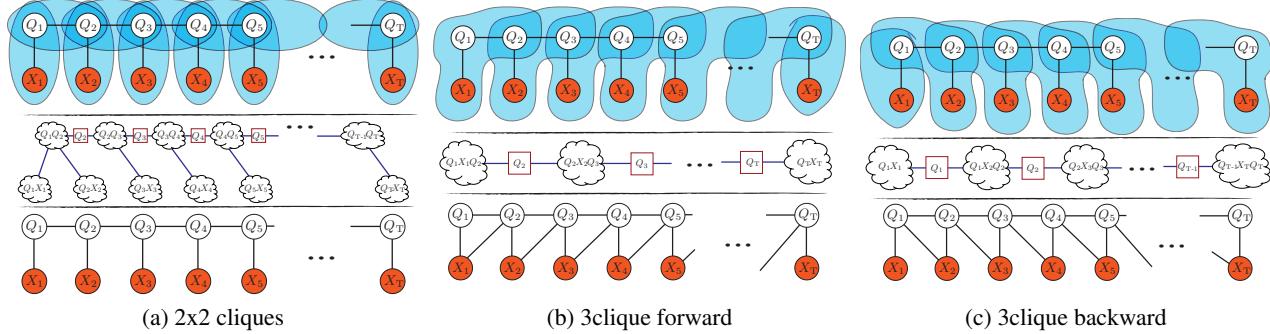


Figure 8.52: Three possible triangulations and junction trees associated with HMMs. In each figure, the top shows the HMM as a Markov random field along with blue regions indicating the clusterings of the variables that will constitute cliques. In the middle, we show the junction tree where clusters are shown as clouds and separators are enclosed in square boxes. On the bottom we show the reconstituted HMM with additional edges if any. On the left, we see that each observed variable X_t is paired with its hidden variable Q_t and each hidden variable Q_t is paired with its neighbor Q_{t+1} . In the middle, we have 3-cliques consisting of Q_t, Q_{t+1}, X_t . On the right, we have 3-cliques consisting of Q_t, Q_{t+1}, X_{t+1} . Note that in each case, the computational cost is $O(TN^2)$ and (assuming only the separators are stored) a memory cost of $O(TN)$. The right figure here should be compared with Figure 8.44 since it also shows the trellis.

and uses a 2-clique for each edge in the HMM's graphical model. This is shown in Figure 8.52a. Another solution clusters together two hidden variables and an observed variable, Figures 8.52b and 8.52c shows two variants of this procedure. Using any of the junction trees, we can define a forward and/or backwards recursion, and we'll again, in some cases, either recover exactly the α, β HMM recursions, or computations that are quite similar (and produce equivalent results). For example, the Hugin message passing procedure computes the backward messages based on the forward message (if Hugin's collect evidence corresponds to moving from left to right and distribute evidence corresponds to moving from right to left) and this is different than the standard α, β recursions which are independent of each other (i.e., values of the β recursion do not need α recursion values). If we apply the Hugin procedure so that collect evidence corresponds to moving from right to left, we'll get yet another recursion for HMMs.

8.4.11.10 Viterbi path, Viterbi decoding, and most probable explanation (MPE)

In this section, we discuss how to compute the Viterbi path $\operatorname{argmax}_{y_{1:n}} p(x_{1:n}, y_{1:n})$, which is also known as the MPE (most probable estimate) or inference under the max-product semi-ring, or the highest probability configuration of random variables known as the maximum a posteriori (MAP) inference problem.

Indeed, this is a crucial problem in HMMs is to solve since it is the primary way one maps from an unknown string $\bar{x}_{1:T}$ to a crisp decision among states, as in:

$$q_{1:T}^* \in \operatorname{argmax}_{q_{1:T} \in D_{Q_{1:T}}} p(x_{1:T}, q_{1:T}) \quad (8.168)$$

The first thing to note is that computing the **value** of the MPE can be done just with an alternate form of

the α -recursion. Since

$$\max_{q_{1:T} \in D_{Q_{1:T}}} p(\bar{x}_{1:T}, q_{1:T}) = \max_{q_{1:T} \in D_{Q_{1:T}}} \prod_t p(\bar{x}_t | q_t) p(q_t | q_{t-1}) \quad (8.169)$$

$$= \max_{q_T} p(x_T | q_T) \dots \left(\max_{q_2} p(\bar{x}_2 | q_2) p(q_3 | q_2) \left(\max_{q_1} p(\bar{x}_1 | q_1) p(q_2 | q_1) \right) \right) \quad (8.170)$$

$$= \max_{q_T} p(x_T | q_T) \dots \left(p(x_3 | q_3) \max_{q_2} p(q_3 | q_2) \left(p(\bar{x}_2 | q_2) \max_{q_1} p(q_2 | q_1) (p(\bar{x}_1 | q_1)) \right) \right) \quad (8.171)$$

we see that computing the max value is just using the max-product semi-ring rather than the sum-product semi-ring (cf. §8.4.4.5).

Based on this, we can thus define a modified form of the α -recursion that, rather than uses summation, uses a max operator.

$$\alpha_q^m(1) = p(\bar{x}_t | Q_t = q) \quad (8.172)$$

$$\alpha_q^m(t) = p(\bar{x}_t | Q_t = q) \max_r p(Q_t = q | Q_{t-1} = r) \alpha_r^m(t-1) \quad (8.173)$$

We then get

$$\max_{q_{1:T} \in D_{Q_{1:T}}} p(\bar{x}_{1:T}, q_{1:T}) = \max_q \alpha_q^m(T) \quad (8.174)$$

Note that the max operator is quite similar to sum, in that it “marginalizes” out hidden variables. Hence, given $p(\bar{x}, q)$ when we form $\max_q p(\bar{x}, q)$ we can think of this as a marginal, and in fact is called the *max marginal* and has the form:

$$p^m(\bar{x}) = \max_q p(\bar{x}, q) \quad (8.175)$$

Given this, we can view the $\alpha_q^m(t)$ as the max marginals up to time t

$$\alpha_q^m(t) = p^m(x_{1:t}, Q_t = q) \quad (8.176)$$

so that the above final maximization makes sense. We've defined a recursive way to compute the max marginal.

Since dynamic programming works on any commutative semi-ring, we're just defining the α recursion using the max-product semi-ring rather than the previous sum-product semi-ring but we in fact could compute a marginal under any semiring.

So far, however, this computes only the value of the Viterbi assignment but how do we get the actual states values? It turns out, we will need to do a forward and a backward pass, like α and β recursions, but where they are modified. Note that the argmax operator also distributes in a fashion. The true max assignment at time t will depend on what the true max at time $t+1$ is. Hence, we can pre-compute the max for all q at time t when going forward, and then when going backwards, once we know the true max at time $t+1$, we backtrack and then used the previously computed max at time t . We repeat this process from T back to 1 and in doing so produce the MPE.

Lets do the above process directly:

$$\operatorname{argmax}_{q_{1:T} \in D_{Q_{1:T}}} p(\bar{x}_{1:T}, q_{1:T}) = \operatorname{argmax}_{q_{1:T} \in D_{Q_{1:T}}} \prod_t p(\bar{x}_t | q_t) p(q_t | q_{t-1}) \quad (8.177)$$

$$= \operatorname{argmax}_{q_T} p(x_T | q_T) \dots \left(\operatorname{argmax}_{q_2} p(\bar{x}_2 | q_2) p(q_3 | q_2) \left(\operatorname{argmax}_{q_1} p(\bar{x}_1 | q_1) p(q_2 | q_1) \right) \right) \quad (8.178)$$

$$= \operatorname{argmax}_{q_T} p(x_T | q_T) \dots \left(p(x_3 | q_3) \operatorname{argmax}_{q_2} p(q_3 | q_2) \left(p(\bar{x}_2 | q_2) \operatorname{argmax}_{q_1} p(q_2 | q_1) (p(\bar{x}_1 | q_1)) \right) \right) \quad (8.179)$$

That is, like the max operator, the argmax operator can be sent into the product as far as possible so that it operates only on factors that involve its own operands. Note that the inner-most argmax depends on true max assignment for q_2 ; the next inner-most argmax depends on the true max assignment for q_3 , and this continues until the outer-most argmax over q_T . This means that it is possible to define a recursion for this process that stores the integer state (or backpointer) indices based on the max marginal that we've previously computed. That is we define $\check{\alpha}_q^m(t)$ to be the quantity defined as:

$$\check{\alpha}_{q_t}^m(t) \in \operatorname{argmax}_{q_{t-1}} p(q_t|q_{t-1}) \alpha_{q_{t-1}}^m(t-1) \quad (8.180)$$

where ties are broken arbitrarily. Note that $\check{\alpha}_q^m(t)$ takes on integer values rather than a probability score, and $\check{\alpha}_q^m(t)$ is computed along with $\alpha_q^m(t)$.

Note that $\check{\alpha}_{q_t}^m(t)$'s recursion does not explicitly use $p(\bar{x}_{t-1}|q_{t-1})$ as this is already incorporated into $\alpha_{q_{t-1}}^m(t-1)$. Also, $\check{\alpha}_q^m(t)$'s recursion also does not explicitly use $p(\bar{x}_t|q_t)$ the way that $\alpha_q^m(t)$'s recursion uses $p(\bar{x}_t|q_t)$. The basic issue is that $\check{\alpha}_q^m(t)$ for each q holds a pointer back to the previous time step which is the maximum way of getting to the state q . To find the max way to get to the current state q_t at time t , we do not need to know $p(\bar{x}_t|q_t)$ since it is a constant at that time (the path leading to q_t at time t will be the same for all values of $p(\bar{x}_t|q_t)$). But we do need the observation score at time $t-1$ for each state q_{t-1} since the observation helps determine the score at time $t-1$ state q_{t-1} . At the final time step T , we use the observation at time T to find the max state q_T (and we use the normal max marginal at time t for this purpose) which is then used to start the trace-back, as we see below.

With the above max marginals and back pointers precomputed, we can then compute the Viterbi path via backtracing as is done in the following algorithm:

```

1 Compute  $q_T^* \in \operatorname{argmax}_q \alpha_q^m(T)$ 
2 for  $t = T \dots 2$  do
3   Set  $q_{t-1}^* \leftarrow \hat{\alpha}_{q_t^*}^m(t)$ 

```

The resulting sequence $(q_1^*, q_2^*, \dots, q_T^*)$ is the Viterbi path. Note that the algorithm entirely deterministic in that it uses only array lookups (except for the initial case in line 1 where we first find the maximum state).

So, to summarize the computation of the Viterbi path, we first compute the following equations:

$$\alpha_q^m(1) = p(\bar{x}_1|Q_1 = q) \quad (8.181)$$

$$\alpha_q^m(t) = p(\bar{x}_t|Q_t = q) \max_r p(Q_t = q|Q_{t-1} = r) \alpha_r^m(t-1) \quad (8.182)$$

$$\check{\alpha}_q^m(t) \in \operatorname{argmax}_r p(Q_t = q|Q_{t-1} = r) \alpha_r^m(t-1) \quad (8.183)$$

and then run the above backward algorithm to get the path. It should be clear that the backward path is exactly analogous to the dynamic programming backtracking process we saw in Figure 8.39. In fact, this figure could just as easily be used to show the HMM trellis and the Viterbi path (in green).

One question might be why is the process of Viterbi decoding called “decoding?”. The reason stems from the original motivation for computing the Viterbi path, and relates to the source-channel model of communication (from Information Theory) as shown in Figure 8.66. Consider the source being generated by Markov chain, and the “channel” being each symbol corrupted by some channel noise (observation distribution) as shown in Figure 8.53. The process of “decoding” then is to map from the received information (the observations, or $x_{1:T}$) back to the sent information (the source, or the string $y_{1:T}$) which is modeled as a Markov chain. $x_{1:T}$ is viewed as a noisy version of $y_{1:T}$

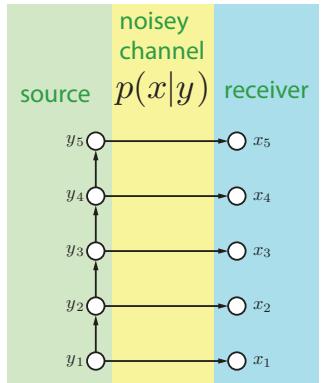


Figure 8.53: HMM as source-channel model, compared to Figure 8.66.

and we need to decode the received signal to recover $y_{1:T}$. Hence, the name Viterbi decoding [432].

8.4.11.11 Other HMM recursions

It is also possible to derive a temporal recursion for quantities other than α and β . For example, what follows is a backwards recursion for the posterior quantity $\gamma_t(j) = p(Q_t = j|x_{1:T})$ based on only having previously computed the α quantities:

$$\gamma(q_t) = \sum_{q_{t+1}} p(q_t, q_{t+1}|x_{1:T}) = \sum_{q_{t+1}} p(q_t|q_{t+1}, x_{1:T})p(q_{t+1}|x_{1:T}) \quad (8.184)$$

$$= \sum_{q_{t+1}} p(q_t|q_{t+1}, x_{1:T})\gamma(q_{t+1}) = \sum_{q_{t+1}} p(q_t|q_{t+1}, x_{1:t})\gamma(q_{t+1}) \quad (8.185)$$

$$= \sum_{q_{t+1}} \frac{p(q_t, q_{t+1}, x_{1:t})}{\sum_{q'_t} p(q'_t, q_{t+1}, x_{1:t})}\gamma(q_{t+1}) \quad (8.186)$$

$$= \sum_{q_{t+1}} \frac{p(q_{t+1}|q_t)p(q_t, x_{1:t})}{\sum_{q'_t} p(q_{t+1}|q'_t)p(q'_t, x_{1:t})}\gamma(q_{t+1}) \quad (8.187)$$

$$= \sum_{q_{t+1}} \frac{p(q_{t+1}|q_t)\alpha_t(q_t)}{\sum_{q'_t} p(q_{t+1}|q'_t)\alpha_t(q'_t)}\gamma(q_{t+1}) \quad (8.188)$$

Therefore, there is a backwards pass recursion using just the α 's without directly touching the observations again. This can be useful if the observations must, for some reason, be ephemeral.

The Hugin message passing scheme was developed for general graphical models but it can be equally applied in the case of the HMM. Hugin message passing is based on a junction tree and so it is necessary to first decide which junction tree to use, and there are of course a number of possibilities. Each of these triangulations have identical computational and memory requirements for performing inference as shown in Figure 8.52, namely $O(TN^2)$ compute costs and $O(TN)$ memory costs.¹⁰ In the following, we'll chose Figure 8.52c.

Hugin message passing maintains both clique and separator potentials one potential for each of the cloud and the square shapes in Figure 8.52c in the middle row. The cloud potentials are called $\psi(q_t, q_{t_1})$ and the separator potentials are called $\phi(q_t)$. There are different instances of these potentials, as summarized in Figure 2.6-right, depending on which phase of message sending we are in.

We will construct these messages for this junction tree and assume that MPP has been followed in left-to-right and then right-to-left order.

First, we make the following initializations over the entire sequence:

$$\phi(q_t) \leftarrow 1 \quad \forall t \quad (8.189)$$

$$\psi(q_t, q_{t-1}) \leftarrow \begin{cases} p(q_t|q_{t-1})p(\bar{x}_t|q_t) & \text{if } t > 1 \\ p(\bar{x}_1|q_1)p(q_1) & \text{if } t = 1 \end{cases} \quad (8.190)$$

¹⁰It should be noted that while inference cost is the same for these three triangulations, other operations using this model under different triangulations might have a significant practical difference in performance. For example, when learning the parameters of the observation distributions $p(x_t|q_t)$, the two triangulations that involve X_t with two successive Q_t variables can be more costly. The reason for this is that a clique with variables $\{X_t, Q_t, Q_{t+1}\}$ implies summing over N^2 values to compute the accumulated statistics for $p(x_t|q_t)$. An example is the computation $\sum_{q_t, q_{t+1}} x_t p(q_t, q_{t+1}|x_{1:n})$ which is obviously unnecessarily slower by a factor N than the computation $\sum_{q_t} x_t p(q_t|x_{1:n})$, especially when x_t is a large continuous valued vector. While this is still $O(N^2)$, the additional constant cost of accumulating large vectors can be appreciable in an implementation.

We see that forward messages (or collect evidence) proceeds as follows:

$$\phi^*(q_t) \leftarrow \sum_{q_{t-1}} \psi(q_t, q_{t-1}), \quad \text{for } t = 2 \dots T \quad (8.191)$$

$$\psi^*(q_t, q_{t-1}) \leftarrow \psi(q_t, q_{t-1}) \frac{\phi^*(q_{t-1})}{\phi(q_{t-1})}, \quad \text{for } t = 2 \dots T \quad (8.192)$$

$$(8.193)$$

Hence, we can see that if we form messages $\phi^*(q_t)$ in temporal order, this is identical to running the elimination algorithm on the hidden variables of the HMM and we get the interpretation:

$$\phi^*(q_t) = p(\bar{x}_1, \dots, \bar{x}_t, Q_t = q_t) = \alpha_t(q_t) \quad (8.194)$$

and

$$\psi^*(q_{t-1}, q_t) = \alpha_{t-1}(q_{t-1}) p(x_t | q_t) p(q_t | q_{t-1}) = p(x_1, \dots, x_t, q_{t-1}, q_t) \quad (8.195)$$

Interestingly, it is the Hugin separator potential $\phi^*(q_t)$ which corresponds exactly to the α recursion for an HMM and so we see that from Figure 8.52c and more specifically Figure 8.44, aligning the square separator potential boxes in Figure 8.44-(C) with the state columns in Figure 8.44-(D) makes sense. We see that if we were to store the clique potentials $\psi^*(q_{t-1}, q_t)$ going forward, the memory complexity of HMM inference would be $O(N^2T)$ — by only storing the separator potentials $\phi^*(q_t)$, we get the HMM memory complexity of $O(NT)$.

Now going backwards (via distribute evidence again obeying MPP), we get:

$$\phi^{**}(q_t) \leftarrow \begin{cases} \sum_{q_{t+1}} \psi^{**}(q_t, q_{t+1}) & \text{if } 1 \leq t < T \\ \phi^*(q_t) & \text{otherwise} \end{cases} \quad (8.196)$$

$$\psi^{**}(q_{t-1}, q_t) \leftarrow \psi^*(q_{t-1}, q_t) \frac{\phi^{**}(q_t)}{\phi^*(q_t)} \quad \text{for } 2 \leq t \leq T \quad (8.197)$$

The interpretation of ϕ^{**} and ψ^{**} is as follows:

$$\phi^{**}(q_T) = p(x_1, x_2, \dots, x_T, q_T) \quad (8.198)$$

$$\psi^{**}(q_{T-1}, q_T) = p(x_1, \dots, x_T, q_{T-1}, q_T) \quad (8.199)$$

$$\phi^{**}(q_{T-1}) = \sum_{q_T} p(x_1, x_2, \dots, x_T, q_{T-1}, q_T) \quad (8.200)$$

$$= p(x_1, \dots, x_T, q_{T-1}) \quad (8.201)$$

$$\psi^{**}(q_{T-2}, q_{T-1}) = p(x_1, \dots, x_{T-1}, q_{T-2}, q_{T-1}) \frac{p(x_1, \dots, x_T, q_{T-1})}{p(x_1, \dots, x_{T-1}, q_{T-1})} \quad (8.202)$$

$$= p(x_1, \dots, x_{T-1}, q_{T-2}, q_{T-1}) p(x_T | x_1, \dots, x_{T-1}, q_{T-1}) \quad (8.203)$$

$$= p(x_1, \dots, x_T, q_{T-2}, q_{T-1}). \quad (8.204)$$

For general t going backwards, we get

$$\phi^{**}(q_t) = p(x_1, x_2, \dots, x_T, q_t) \quad (8.205)$$

and

$$\psi^{**}(q_{t-1}, q_t) = p(x_1, x_2, \dots, x_T, q_{t-1}, q_t) \quad (8.206)$$

Hence, from $\phi^{**}(q_t)$ we can get the state posteriors $p(q_t|\bar{x}_{1:T}) = \phi^{**}(q_t)/\sum_{q'_t} \phi^{**}(q_t)$ and edge posteriors $p(q_{t-1}, q_t|\bar{x}_{1:T}) = \psi^{**}(q_{t-1}, q_t)/\sum_{q'_{t-1}, q_t} \psi^{**}(q_{t-1}, q_t)$. Moreover, we can define a direct backwards recursion from $\phi^{**}(q_t)$ to $\phi^{**}(q_{t-1})$ as follows

$$\phi^{**}(q_t) \leftarrow \sum_{q_{t+1}} \psi^*(q_t, q_{t+1}) \frac{\phi^{**}(q_{t+1})}{\phi^*(q_{t+1})} \quad (8.207)$$

$$(8.208)$$

We see, therefore, that this is a recursion that we have not seen before.

Exercise 127. Write down the HMM recursions that result from applying the Hugin procedure's collect/distribute evidence to other triangulations in of the HMM graph in Figure 8.52.

Exercise 128. Write down the HMM recursions that result from applying Pearl's $\lambda - \pi$ inference updates [336] to the Bayesian network view of an HMM.

Next, we consider normalized versions of the α and β recursions, starting with the α case. Recall that $\alpha_t(j) = p(x_1, \dots, x_t, Q_t = j)$. Suppose we compute a form of normalized alpha, defined as follows:

$$\hat{\alpha}_t(j) \triangleq \frac{p(x_1, \dots, x_t, Q_t = j)}{\sum_j p(x_1, \dots, x_t, Q_t = j)} = \frac{p(x_1, \dots, x_t, Q_t = j)}{p(x_1, \dots, x_t)} = p(Q_t = j|x_1, \dots, x_t) \quad (8.209)$$

One thing to note is that $0 \leq \hat{\alpha}_t(j) \leq 1$ for all t and j and hence the dynamic range of $\hat{\alpha}$ is limited unlike α (we discuss this more in §8.4.11.12).

Is there a direct way to get from $\hat{\alpha}_t(j)$ to $\hat{\alpha}_{t+1}(j)$? Let us define

$$\nu_t \triangleq p(x_1, x_2, \dots, x_t) \quad (8.210)$$

We have

$$\hat{\alpha}_{t+1}(j) = p(Q_{t+1} = j|x_1, \dots, x_{t+1}) = \frac{p(Q_{t+1} = j, x_1, \dots, x_{t+1})}{p(x_1, \dots, x_{t+1})} \quad (8.211)$$

$$= \frac{\sum_i p(Q_{t+1} = j, Q_t = i, x_1, \dots, x_{t+1})}{p(x_1, \dots, x_{t+1})} \quad (8.212)$$

$$= \frac{\sum_i p(x_{t+1}|Q_{t+1} = j)p(Q_{t+1} = j, Q_t = i, x_1, \dots, x_t)}{p(x_1, \dots, x_{t+1})} \quad (8.213)$$

$$= \frac{\sum_i p(x_{t+1}|Q_{t+1} = j)p(Q_{t+1} = j|Q_t = i)p(Q_t = i, x_1, \dots, x_t)}{p(x_1, \dots, x_{t+1})} \quad (8.214)$$

$$= \frac{\sum_i p(x_{t+1}|Q_{t+1} = j)p(Q_{t+1} = j|Q_t = i)\hat{\alpha}_t(i)\nu_t}{\nu_{t+1}} \quad (8.215)$$

$$= \frac{\sum_i p(x_{t+1}|Q_{t+1} = j)p(Q_{t+1} = j|Q_t = i)\hat{\alpha}_t(i)}{p(x_{t+1}|x_1, x_2, \dots, x_t)} \quad (8.216)$$

since $\nu_{t+1}/\nu_t = p(x_{t+1}|x_1, x_2, \dots, x_t)$. One thing to note is that since ν_{t+1}/ν_t is not a function of j , this normalization constraint can be computed directly, i.e., by forming

$$\hat{\alpha}_{t+1}(j) = \frac{\sum_i p(x_{t+1}|Q_{t+1} = j)p(Q_{t+1} = j|Q_t = i)\hat{\alpha}_t(i)}{\sum_{j'} \sum_i p(x_{t+1}|Q_{t+1} = j')p(Q_{t+1} = j'|Q_t = i)\hat{\alpha}_t(i)} \quad (8.217)$$

which implies that

$$\eta_{t+1} \triangleq p(x_{t+1}|x_1, x_2, \dots, x_t) = \sum_{j'} \sum_i p(x_{t+1}|Q_{t+1} = j') p(Q_{t+1} = j'|Q_t = i) \hat{\alpha}_t(i) \quad (8.218)$$

This is actually not surprising since by the HMM conditional independence properties:

$$p(x_{t+1}|x_1, \dots, x_t) = \sum_{q_{t+1}, q_t} p(q_t, q_{t+1}, x_{t+1}|x_1, \dots, x_t) \quad (8.219)$$

$$= \sum_{q_{t+1}, q_t} p(x_{t+1}|q_{t+1}) p(q_{t+1}|q_t) p(q_t|x_1, \dots, x_t) \quad (8.220)$$

$$= \sum_{q_{t+1}, q_t} p(x_{t+1}|q_{t+1}) p(q_{t+1}|q_t) \hat{\alpha}_t(q_t) \quad (8.221)$$

Hence, we see that Equation (8.217) is a recursion for our normalized alphas. Note that the computation of $\hat{\alpha}_t(j)$ for all j is a bit more expensive than computing $\alpha_t(j)$. While both require $O(N^2)$, the normalization constant in $\hat{\alpha}_t(j)$ itself requires N^2 steps.

Next, we derive a normalized version of the β recursion. Recall that $\beta_t(q) = p(x_{t+1:T}|Q_t = q)$. Note that if we were to normalize as in $\beta_t(q)/\sum_q \beta_t(q)$ we would get a normalized quantity, but this would not be as useful as the normalization that we will now introduce. Consider forming

$$\hat{\beta}_t(q) = \frac{\beta_t(q)}{\prod_{\tau=t+1}^T \eta_\tau} = \frac{p(x_{t+1:T}|Q_t = q)}{\prod_{\tau=t+1}^T p(x_\tau|x_1, \dots, x_{\tau-1})} \quad (8.222)$$

The first thing to note is that with this normalization value of $\prod_{\tau=t+1}^T \eta_\tau$ the dynamic range of $\hat{\beta}$ will not grow unboundedly like β does as t moves down from T (we discuss this more in §8.4.11.12). Secondly, given $\hat{\beta}_t(q)$ we can form $\hat{\beta}_{t-1}(q)$ as follows:

$$\hat{\beta}_{t-1}(q) = \frac{\beta_t(q)}{\prod_{\tau=t+1}^T \eta_\tau} \quad (8.223)$$

$$= \frac{\sum_r \beta_{t+1}(r) p(x_{t+1}|Q_{t+1} = r) p(Q_{t+1} = r|Q_t = q)}{\prod_{\tau=t+1}^T \eta_\tau} \quad (8.224)$$

$$= \frac{\sum_r [\hat{\beta}_{t+1}(r) \prod_{\tau=t+2}^T \eta_\tau] p(x_{t+1}|Q_{t+1} = r) p(Q_{t+1} = r|Q_t = q)}{\prod_{\tau=t+1}^T \eta_\tau} \quad (8.225)$$

$$= \frac{\sum_r \hat{\beta}_{t+1}(r) p(x_{t+1}|Q_{t+1} = r) p(Q_{t+1} = r|Q_t = q)}{\eta_{t+1}} \quad (8.226)$$

which follows from the definition of the β recursion in Equation (8.116). Hence, we have a backwards recursion for $\hat{\beta}$. Note that this backward recursion uses the values η_t that were computed during the forward recursion — since the η_t sequence only requires one additional scalar to be stored per time step, this usually is not a significant increase in memory usage.

What can we do with $\hat{\alpha}$ and $\hat{\beta}$ other than the scaling issues discussed in §???. For one thing we can immediately get normalized clique posteriors without needing to compute the normalization constant. That is, since

$$\hat{\alpha}_t(j) = \frac{\alpha_t(j)}{p(x_1, \dots, x_t)} \quad \text{and} \quad \hat{\beta}_t(j) = \frac{\beta_t(j)}{\prod_{\tau=t+1}^T p(x_\tau|x_1, \dots, x_{\tau-1})} \quad (8.227)$$

we have that

$$\hat{\alpha}_t(j)\hat{\beta}_t(j) = \frac{\alpha_t(j)\beta_t(j)}{p(x_1, x_2, \dots, x_T)} = p(q_t|x_{1:T}) \quad (8.228)$$

meaning we can form the clique posteriors $p(q_t|x_{1:T})$ without needing to perform a normalization step as is done in Equation (8.158).

It is also possible to get the state pair (or edge) posterior from these recursions.

$$p(q_t, q_{t-1}|\bar{x}_{1:T}) = \frac{p(q_t, q_{t-1}, \bar{x}_{1:T})}{p(x_{1:T})} \quad (8.229)$$

$$= \frac{\alpha_{t-1}(q_{t-1})p(x_t|q_t)p(q_t|q_{t-1})\beta_t(q_t)}{p(x_1, \dots, x_{t-1})p(x_t|x_1, \dots, x_{t-1})\prod_{\tau=t+1}^T p(x_\tau|x_1, \dots, x_{\tau-1})} \quad (8.230)$$

$$= \frac{\hat{\alpha}_{t-1}(i)p(x_t|q_t)p(q_t|q_{t-1})\hat{\beta}_t(j)}{\eta_t} \quad (8.231)$$

which follows from Equation (8.164) and the definitions of $\hat{\alpha}$, $\hat{\beta}$, and η_t . Thus, this posterior can be obtained with only one scalar divide.

The $\hat{\alpha}$ and $\hat{\beta}$ recursions can be useful if we want only a sparse set of the posterior values meaning if we only want to compute some of the values of the posteriors rather than all of them, without having to do a renormalization to get the posteriors. On the other hand, due to the extra N^2 steps of computation needed during the $\hat{\alpha}$ recursion, it might not be worth it since that computation could be used to normalize the normal $\alpha\beta$ values.

Perhaps a key benefit of $\hat{\alpha}$ and $\hat{\beta}$ is that they have better dynamic range properties and so address a problem we consider in the next section.

8.4.11.12 HMM scaling issues, numerical dynamic range, underflow, normalization, and log arithmetic

When we construct $\alpha_t(j) = p(x_1, \dots, x_t, Q_t = j)$, we can easily see that as t gets big, the probability value of $\alpha_t(j)$ gets smaller and smaller. In fact, the typical value of $\alpha_t(j)$ gets exponentially smaller in t — that is $\alpha_t(j) \approx O(r^{-t})$ where r is the number of states. As a simple example of this, suppose for the moment that all variables are independent, so that $p(x_1, \dots, x_t, Q_t = j) = p(Q_t = j) \prod_{\tau=1}^t p(x_\tau)$ (it is very important to realize that this is **not** the case with an HMM, but we are doing this here just as an example). Then if each $p(x_\tau) = 0.5$ then we see that $\alpha_t(j) \propto 0.5^t$ which gets exponentially small in t . The fact that an HMM is not independent can make this either better or worse, depending on the distribution. I.e., it could be that x_1, x_2, \dots are such that the probability does not decrease as fast (it's even mathematically possible for $p(x_1, \dots, x_t, Q_t = j) = p(Q_t = j)$), but it is very common in practice that this decreases very fast (much faster than the additional factor of 0.5 at each time step, as suggested above). This is especially true when the HMM uses Gaussian distributions, as is done often in speech recognition systems.

Similarly, $\beta_t(j)$ gets smaller as t goes down from T , that is $\beta_t(j)$ gets exponentially small with $T - t$. Hence it is not long, even for moderate values of t , before we will run out of numeric precision. The smallest single precision 32-bit IEEE floating point number is approximately 10^{-33} the smallest double precision 64-bit IEEE floating point is about 10^{-308} . So if $r = 10$, this means that an HMM can not be longer than about 308 frames before running out of numeric precision, and this is when using double precision arithmetic. Hence, we are unable to practically compute $\alpha_t(j)$ and $\beta_t(j)$ for all but very small values of T , unless we do something smarter.

There are many possible solutions and which solution is desirable depends on which operations one wishes to perform. For example, one may wish to compute $p(\bar{x}_{1:T})$, or one may only wish to compute the

posteriors mentioned above (say for parameter learning), or one may wish to compute the Viterbi path (for decision making). Lets consider each in turn.

First, to compute the probability of evidence $p(\bar{x}_{1:T})$, this quantity itself presents a problem. As T gets larger, $p(\bar{x}_{1:T})$ gets (typically exponentially in T) smaller and so would not have enough precision to even store the answer to this query. One solution is to compute $\log p(\bar{x}_{1:T})$ directly, and this is possible using log arithmetic which we describe below.

To compute a quantity like $p(q_t|\bar{x}_{1:T})$ or $p(q_t, q_{t-1}|\bar{x}_{1:T})$ we need both a numerator such as $p(q_t, \bar{x}_{1:T})$ (or $p(q_t, q_{t-1}, \bar{x}_{1:T})$) and a denominator $p(\bar{x}_{1:T})$. Each of these themselves would have values too small to store (i.e., numeric underflow issues would cause them to be truncated to zero in most finite-precision arithmetic systems, including IEEE floating point, which is the standard). If we were able to compute quantities that are proportional to each (say $ctp(q_t, \bar{x}_{1:T})$ and $ctp(\bar{x}_{1:T})$) where both numerator and denominator have the same constant c_T that also scales roughly exponentially with T , then we should be able to compute the desired quantities. Note that the proportional constant c_T is a function of T which means that it scales as the sequence gets longer. Moreover, this strategy never actually computes $p(q_t, \bar{x}_{1:T})$ but only quantities that are a known proportionality constant away from it.

Lastly, to compute the Viterbi path, we need to produce a structure that allows us to obtain the same quantity but without needing to produce such small values. This, as we will soon see, is also possible.

There are two general strategies for dealing with the above problems: 1) scaling the scores, and 2) log arithmetic, and we'll discuss them in this order.

We've in fact already seen a form of scaling in that the $\hat{\alpha}_t(j)$ and $\hat{\beta}_t(j)$ that we produced in §8.4.11.11 are already scaled. I.e., we have a guarantee that $0 \leq \alpha_t(j) \leq 1$ for all j and so we are certain that the dynamic range of $\hat{\alpha}_t(j)$ will not only be bounded but also quite manageable. Also, $\hat{\beta}_t(j) = p(x_{t+1:T}|q)/\prod_{\tau=t+1}^T p(x_\tau|x_1, \dots, x_{\tau-1})$. While we do not have a bounded range guarantee in this case, it is the case that both the numerator and denominator scales with the number of variables. As an example, if it was the case that all the variables were independent (which it is not, but just for the sake of this discussion), then both numerator and denominator would be the product of $T - t$ probability values and would be in a reasonable range. Given that the random variables are not independent, the numerator is not such a product but in general they won't be too far away from each other, and so $\hat{\beta}_t(j)$ will also not need enormous dynamic range.

Consider another form of scaled $\alpha_t(j)$ where at each time step in the recursion we form:

$$\bar{\alpha}_t(q) = \frac{\sum_r p(x_t|q)p(q|r)\bar{\alpha}_{t-1}(r)}{d_t} \quad (8.232)$$

Hence, we see that

$$\bar{\alpha}_t(q) = \frac{\alpha_t(q)}{\prod_{\tau=1}^t d_\tau} \quad (8.233)$$

and if d_t is chosen wisely then $\bar{\alpha}_t(j)$ will have a reasonable scale. One choice of d_t would be the maximum at each step as in

$$d_t = \max_q \left[\sum_r p(x_t|q)p(q|r)\bar{\alpha}_{t-1}(r) \right] \quad (8.234)$$

and this tends to work well in practice. But other (and even easier) choices for d_t would be $d_t = 1/|D_Q|$, or

$$d_t = \sum_q \left[\sum_r p(x_t|q)p(q|r)\bar{\alpha}_{t-1}(r) \right] \quad (8.235)$$

(which of course would then give us $\hat{\alpha}_t(j)$) or even the median of the quantity. In any event, we'll get a resulting version of α that typically would not decrease exponentially in T .

For the backwards recursion, there are similar options. That is, each backwards recursion would produce $\bar{\beta}$ and would be of the form

$$\bar{\beta}_t(q_t) = \frac{\sum_{q_{t+1}} \bar{\beta}_{t+1}(q_{t+1}) p(x_{t+1}|q_{t+1}) p(q_{t+1}|q_t)}{e_{t+1}} \quad (8.236)$$

where we could use

$$e_{t+1} = \max_{q_t} \left[\sum_{q_{t+1}} \beta_{t+1}(q_{t+1}) p(x_{t+1}|q_{t+1}) p(q_{t+1}|q_t) \right] \quad (8.237)$$

or

$$e_{t+1} = \sum_{q_t} \left[\sum_{q_{t+1}} \beta_{t+1}(q_{t+1}) p(x_{t+1}|q_{t+1}) p(q_{t+1}|q_t) \right] \quad (8.238)$$

or something else. In each case, we have the relationship

$$\bar{\beta}_t(q_t) = \frac{\beta_t(q_t)}{\prod_{\tau=t+1}^T e_\tau} \quad (8.239)$$

and so we see that $\hat{\beta}_t(q_t)$ is likely going to have a well behaved dynamic range.

With $\bar{\alpha}_t$ and $\bar{\beta}_t$ it is easy to get posteriors. For example,

$$p(q_t|\bar{x}_{1:T}) = \frac{\alpha_t(q_t)\beta_t(q_t)}{\sum_{q_t} \alpha_t(q_t)\beta_t(q_t)} = \frac{\prod_{\tau=1}^t d_\tau \bar{\alpha}_t(q_t) \prod_{\tau=t+1}^T e_\tau \bar{\beta}_t(q_t)}{\sum_{q_t} \prod_{\tau=1}^t d_\tau \bar{\alpha}_t(q_t) \prod_{\tau=t+1}^T e_\tau \bar{\beta}_t(q_t)} = \frac{\bar{\alpha}_t(q_t)\bar{\beta}_t(q_t)}{\sum_{q_t} \bar{\alpha}_t(q_t)\bar{\beta}_t(q_t)} \quad (8.240)$$

It is just as easy to get the edge posterior $p(q_t, q_{t-1}|\bar{x}_{1:T})$.

Another option for the backwards pass is to use the direct backward recursion between the state posteriors in Equation (8.188). Since this recursion uses $\alpha_t(q_t)$ both in the numerator or denominator, it is possible to in its place use any of the $\bar{\alpha}_t(q_t)$ quantities that we have defined.

Note that Hugin style forward/backwards messages described in § 8.4.11.11 also suffers from the numerical dynamic range problem, but it would be easy to normalize either the clique potential $\psi^*(q_{t-1}, q_t)$ from Equation (8.195) or the separator potential $\phi^*(q_t)$ from Equation (8.194), the latter case of course being the same as the forming $\bar{\alpha}_t(q_t)$.

To compute Viterbi path, consider the max marginal we computed in Equation (8.182). If we have any construct that is proportional to this max marginal, then the resulting Viterbi path will be the same since argmax operator in Equation (8.183) is immune to the proportionality constant. Hence, we can compute any $\bar{\alpha}_q^m(t) = \alpha_q^m(t)/d_t^m$ where d_t^m is a time-specific constant. One possibility would be the sum $d_t^m = \sum_q \alpha_q^m(t)$ but we could use the max as well. Thus, computing the Viterbi is no different using the normalized max-marginal from using the un-normalized max-marginal.

Scaling of HMM scores is critically important for most if not all HMM systems. For online inference and Kalman-style filtering to be used in streaming applications, scaling the scores is essential to ensure that the unboundedly long sequences that we might encounter will not run out of numeric precision.

We have not yet specified how to compute $p(\bar{x}_{1:T})$. In this case, the value we wish to compute itself is not within the limited range of 64-bit IEEE floating point arithmetic, so the scaling options mentioned above will not work. In this case, therefore, we need a numeric representation that has enough dynamic

range to be able to represent $p(\bar{x}_{1:T})$. One possibility is log-arithmetic where rather than working with probabilities $0 \leq p \leq 1$ we work with log-probabilities $-\infty \leq p \leq 0$. More generally, since a probability score might actually be a likelihood (e.g., a Gaussian could be used for $p(x_t|q_t)$), rather than scores of the form $0 \leq p < \infty$, we have log scores of the form $-\infty \leq \log p < \infty$. Indeed, this corresponds to a form of the log semiring mentioned in Table 8.1.

So let $a = \log p_a$ and $b = \log p_b$ be two log probabilities. In general, there are two operations we may wish to compute with them: 1) multiplication $p_a p_b$ and addition $p_a + p_b$. Multiplication of two log probabilities is particularly easy, since $\log(p_a p_b) = \log p_a + \log p_b = a + b$, so log arithmetic turns multiplication into addition.

Addition of the probabilities $p_a + p_b$ is a bit more tricky however. An obvious way of doing this would be to do convert them back to probabilities from log probabilities, do the addition, and then re-take the log as represented by $a \boxplus b = \log(\exp(a) + \exp(b))$ where $a \boxplus b$ is meant to read “ a log-add b ”. Unfortunately, this does not at all solve the scaling problem, since it might never be the case that we can actually represent p_a . For example, if $a = \log(\alpha_t(q))$ for a large value of t , then just taking $\exp(a)$ would result in zero. In fact, using standard numerical math libraries, $\exp(a) = 0$ roughly whenever $a < -745$, which means once again that when T is large, this approach will fail. We need therefore a smarter strategy for computing a log add.

Now using log arithmetic would not mean that we want to avoid using standard IEEE floating point formats for representing numbers. That is, we still want to be able to store values a and b in standard floating point registers (32 or 64 bit) and use floating point operations whenever possible – this is to ensure that our log arithmetic operations are as fast as possible.

Throughout this discussion, we’ll assume that the base of the logarithm is e — in general, the base can be arbitrary but we need to be sure that we have optimized routines that, for base b , we can compute $\log_b(p)$ and b^p , and base e is a good bet.

The first thing we need is a representation when $p_a = 0$. We use a special value ℓ_0 which is our representation of log zero. This value should be smaller than the smallest possible value we are willing to represent, so something like $\ell_0 = -1 \times 10^{30}$ would be reasonable for single precision. What this says then is that we are not able to represent numbers smaller than $e^{-10^{30}}$. This of course does not mean we’ll ever attempt to evaluate $e^{-10^{30}}$ (we can’t evaluate anything smaller than about e^{-745} as mentioned above) — what it does mean is that any log probability $a \geq -10^{30}$ and if for some reason it falls below this value, it is considered for all intents and purposes as log-zero.

Assume that $a \leq b$ and if not, swap a and b so that it is true. The first thing we check when wanting to compute $a \boxplus b$ is $d = a - b \leq 0$. If $d < -\log(-\ell_0)$ then b is significantly larger than a , so much so that numerically $\log(p_b/p_x) > \log(\ell_0)$ or $p_b/p_a > -\ell_0$. Since $\ell_0 \equiv -\infty$, $d < -\log(-\ell_0)$ hence essentially means that $p_b > \infty \cdot p_a$. So p_b is so much larger than p_a that it is not worth even adding p_a to p_b and we just say that $a \boxplus b = b$.

Our goal is to compute $c = a \boxplus b$. Consider the following derivations:

$$c = \log(p_b + p_a) \tag{8.241}$$

$$= \log(\exp(b) + \exp(a)) \tag{8.242}$$

$$= \log(\exp(b) + \exp(b + a - b)) \tag{8.243}$$

$$= \log(\exp(b)(1 + \exp(a - b))) \tag{8.244}$$

$$= b + \log(1 + \exp(a - b)) \tag{8.245}$$

Using the last expression in Equation (8.245) to compute c is beneficial for a number of reasons:

1. It uses only two calls to transcendental functions as would Equation (8.241).

2. It is amenable to integer representation of the log probabilities meaning that a and b can be quantized to integer values
3. Whether a or b are integer valued or not, the operation $\log(1 + \exp(n))$ can be done efficiently via a table lookup via a rounded value of $a - b$. For example, we can compute a table that stores values for $\log(1 + \exp(\alpha_i))$ which can be looked up with integer i . We can set $\alpha_i \leftarrow \frac{-\log(-\ell_0)}{M}i$ where M is the size of the table.
4. It is numerically much better behaved than the original expression since. For example, if $a \approx b$ and they are both less than -745 , we'll get the much more accurate $b + \log(2)$ rather than $\log(0)$.
5. If a happens to be much smaller than b , then we don't lose precision by needlessly evaluating $b = \log(\exp(b) + 0)$
6. Perhaps most importantly of all, if a or b are less than -745 (which will easily happen), then $c = \log(\exp(a) + \exp(b))$ would at best produce garbage, but this last expression in Equation (8.245) would not.

It is elucidating moreover to consider the function $f(\alpha) = \log(1 + \exp(\alpha))$ for $\alpha \leq 0$. First, we have that $f'(\alpha) = \frac{\exp(\alpha)}{1+\exp(\alpha)} = \frac{1}{1+\exp(-\alpha)}$ which is a sigmoid function. Since $f''(\alpha) = f'(\alpha)(1 - f'(\alpha)) \geq 0$, for all $\alpha \in \mathbb{R}$, we see therefore that $f(\alpha)$ is everywhere convex. Second, when we look at $f(\alpha)$ for $\alpha \leq 0$, and since $\log(1 + \epsilon) \approx \epsilon$ for $\epsilon \approx 0$, we see it appears as a smooth exponential decaying to the left. Hence, it is possible to approximate this function with fast methods more sophisticated than the table lookup mentioned above.

To conclude, log arithmetic is a useful alternative to the issues that arise when working with HMMs and other dynamic graphical models on long sequences. GMTK uses a form of combined log arithmetic and scaling algorithms.

8.4.12 Profile HMMs

Profile HMMs are widely used in computational biology.

8.5 Computing the k best assignments

The most-probable explanation (e.g., Viterbi assignment for Viterbi path) is something that can be computed using a forward recursion under the max-product semi-ring. We know the Viterbi-decoding/MPE for HMMs corresponds to the following:

$$q_{1:T}^*(1) \in \underset{q_{1:T} \in \mathcal{D}_{Q_{1:T}}}{\operatorname{argmax}} p(\bar{x}_{1:T}, q_{1:T}). \quad (8.246)$$

The second best assignment corresponds to the maximum scoring assignment not equal to the highest assignment, i.e.:

$$q_{1:T}^*(2) \in \underset{q_{1:T} \in \mathcal{D}_{Q_{1:T}}, q_{1:T} \neq q_{1:T}^*(1)}{\operatorname{argmax}} p(\bar{x}_{1:T}, q_{1:T}) \quad (8.247)$$

The third best assignment is similar, and can be defined as:

$$q_{1:T}^*(3) \in \underset{q_{1:T} \in \mathcal{D}_{Q_{1:T}}, q_{1:T} \notin \{q_{1:T}^*(1), q_{1:T}^*(2)\}}{\operatorname{argmax}} p(\bar{x}_{1:T}, q_{1:T}) \quad (8.248)$$

Most generally, the k^{th} best assignment, $q_{1:T}^{*(k)}$, can be expressed as:

$$q_{1:T}^{*(k)} \in \underset{q_{1:T} \in \mathcal{D}_{Q_{1:T}}, q_{1:T} \notin \{q_{1:T}^{*(1)}, q_{1:T}^{*(2)}, \dots, q_{1:T}^{*(k-1)}\}}{\operatorname{argmax}} p(\bar{x}_{1:T}, q_{1:T}). \quad (8.249)$$

Our goal is to derive practical and scalable computations for computing top k assignments, i.e., $(q_{1:T}^{*(1)}, q_{1:T}^{*(2)}, \dots, q_{1:T}^{*(k)})$.

The k -best assignment is an important problem for HMMs, for DGMs in general. It may be that $q_{1:T}^{*(1)}$ is very poorly representative of the posterior distribution $p(q_{1:T}|\bar{x}_{1:T})$ of the HMM. For example, knowing just $q_{1:T}^{*(1)}$ we can't tell the difference between a case where the top scores are almost the same (i.e., $p(q_{1:T}^{*(1)}|\bar{x}_{1:T}) \approx p(q_{1:T}^{*(k)}|\bar{x}_{1:T})$ for $k > 1$, HMM posterior distribution is relatively high entropy) while still the top k assignments are wildly different (meaning the HMM is not very confident) vs. a case where the top scores are quite different (i.e., $p(q_{1:T}^{*(1)}|\bar{x}_{1:T}) \gg p(q_{1:T}^{*(k)}|\bar{x}_{1:T})$ for $k > 1$, meaning the HMM posterior distribution is low entropy) while still the top k assignments are quite similar.

There are a number of things we might wish to know about the distribution $p(q_{1:T}|\bar{x}_{1:T})$. Due to the exponential number of paths, we can't look at this entire distribution, we may wish to learn more about the following:

- Confidence: we would like to make decisions based on the HMM and know how confident the HMM is. The Viterbi path and score alone does not tell us how confident the HMM is. The k -best paths and scores, on the other hand, allow us to estimate this confidence, based on properties like the above, where we compare the highest and next highest scores.
- Diversity of paths: If the top k paths are very diverse, this also indicates a lack of confidence as it means that as we move down the ranks, the hypothesis changes. If on the other hand the top k paths are mostly the same (and there are only a small number of locations where a change occurs) this again indicates high confidence in the top path. One example of a diversity function at time t might be something like: $\text{diversity}(t) = \text{number of different states in the set } \{q_t^*(k) : k \in \{1, 2, \dots, K\}\}$. Then the diversity of the top k paths might average diversity(t) over all t .
- Like any distribution, the most probable assignment is not indicative of the typical set [97]. The typical set might have most of the probability, and properties of a “typical” path of the HMM might be very different than properties of the Viterbi path. Computing the k -best paths can help to better estimate the typical set.
- The concentration of scores of the top- k paths can also help to estimate a form of confidence. If the top k scores have very low variance (high concentration), then the HMM might be said to be more confident.
- One can also derive localized versions of the above. That is, there might be some time regions where there is either more or less diversity. In time regions with high (respectively low) diversity among the k best paths, the HMM might be seen to be less (respectively more) confident. This can be useful for many applications. For example, in time regions where the diversity of the top k assignments goes up, this might be a good indication of locations where any decision based on the Viterbi path would make errors. In such case, we might want to trigger some other action for these time regions (such as a more powerful, but computationally costly, model, or might wish to just indicate to a user regions where there is a lack of confidence).

In this section, we will review algorithms for computing the k most probable distinct assignments to the hidden variables, known as the “ k -max” operation. In fact, we will review two quite different algorithms for

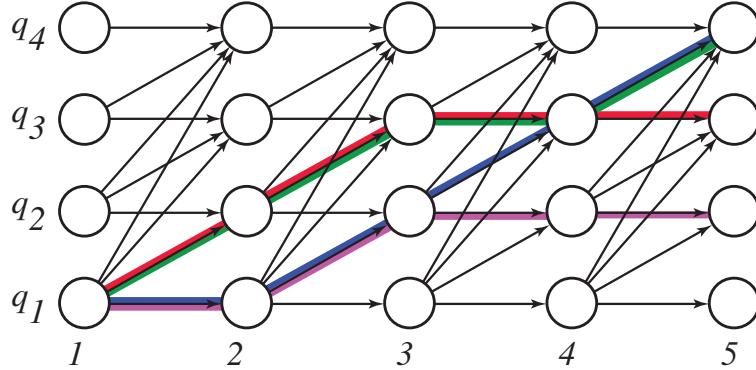


Figure 8.54: 4-best paths in an HMM trellis. Best path is green, next best is blue, third best is red, and forth best is magenta.

doing so, the first one (see §8.5) is quite similar to the standard Viterbi algorithm, and after a bit of setup to define certain algebraic operations (an R-semi-module), will look almost identical. Of interest here how it might be possible to define semi-ring-like operators (e.g., a “ \oplus ” and “ \otimes ”) such that the computation can be viewed as a generalization of finding the 1-best path. In this algorithm, moreover, there will be an additional computational and memory cost of a factor of k . Another attractive property of this algorithm is that it combines very nicely with other time-space tradeoff algorithms, such as the island algorithm (see § 8.6).

The second algorithm (see §8.5.3.4) has better computational and memory properties but is a bit more complicated to explain and implement. However, when time and space is at a premium, this algorithm can be superior.

8.5.1 Finding k -best states in HMMs using R-semi-modules

Let $\overrightarrow{\mathbb{R}}_+^k$ be the set of (descending-order) sorted k -tuples of non-negative real numbers. That is, if $s \in \overrightarrow{\mathbb{R}}_+^k$, then

$$s = (s^1, s^2, \dots, s^k) \quad (8.250)$$

with $s^1 \geq s^2 \geq \dots \geq s^k$. Given two such k -tuples, $s_1, s_2 \in \overrightarrow{\mathbb{R}}_+^k$, we can define an operation corresponding to merging them, re-sorting them, and then truncating the result so that it is once again of length k (and thus is in $\overrightarrow{\mathbb{R}}_+^k$).

That is

$$\text{merge}(s_1, s_2) = (s_1^1, s_1^2, \dots, s_1^k, s_2^1, s_2^2, \dots, s_2^k) \quad (8.251)$$

We sort the result

$$\text{sort}(\text{merge}(s_1, s_2)) = (s^1, s^2, \dots, s^{2k}) \quad (8.252)$$

where $s^1 \geq s^2 \geq \dots \geq s^{2k}$. Then a truncation is such that

$$k\text{-truncate}(\text{sort}(\text{merge}(s_1, s_2))) = (s^1, s^2, \dots, s^k) = s \quad (8.253)$$

which just takes the first k items in the length $2k$ tuple.

In this document, we will be using superscripts, such as s^i to indicate an element of a sorted k -tuple s , and we'll use subscripts, such as s_1, s_2 to indicate particular k -tuples. All k -tuples will be assumed to be sorted in descending order.

We define the process of merging, sorting, and then truncating the ordered tuple to retain the largest k elements, as the operation $\oplus : \overrightarrow{\mathbb{R}}_+^k \times \overrightarrow{\mathbb{R}}_+^k \rightarrow \overrightarrow{\mathbb{R}}_+^k$. That is, the above sequence of operations defines the operation

$$s = (s^1, s^2, \dots, s^k) = (s_1^1, s_1^2, \dots, s_1^k) \oplus (s_2^1, s_2^2, \dots, s_2^k) = s_1 \oplus s_2 = k\text{-truncate}(\text{sort}(\text{merge}(s_1, s_2))) \quad (8.254)$$

First note that if $k = 1$, then this is essentially just the max operator, since merging creates a list of size two, sorting places a maximum element in position one, and then truncation down to one element selects that maximum.

For example, when $k = 1$, we have that:

$$(1) \oplus (3) = (3) \quad (8.255)$$

$$(2) \oplus (3) = (3) \quad (8.256)$$

$$(0) \oplus (1) = (1) \quad (8.257)$$

$$(0) \oplus (0) = (0) \quad (8.258)$$

$$(8.259)$$

As another example, when $k = 3$, we have

$$(3, 2, 1) \oplus (4, 2, 1) = (4, 3, 2) \quad (8.260)$$

$$(3, 2, 1) \oplus (3, 2, 1) = (3, 3, 2) \quad (8.261)$$

$$(0, 0, 0) \oplus (3, 2, 1) = (3, 2, 1) \quad (8.262)$$

$$(1, 0, 0) \oplus (3, 2, 1) = (3, 2, 1) \quad (8.263)$$

$$(10, 9, 0) \oplus (3, 2, 1) = (10, 9, 3) \quad (8.264)$$

$$(10, 0, 0) \oplus (3, 0, 0) = (10, 3, 0) \quad (8.265)$$

$$(0, 0, 0) \oplus (1, 0, 0) = (1, 0, 0) \quad (8.266)$$

$$(8.267)$$

We also define the operation of converting a non-negative scalar to a sorted k -tuple as follows. For $r \in R_+$, which is the set of non-negative reals, we have

$$\overrightarrow{\mathbb{R}}_+^k(r) = (r, 0, 0, \dots, 0) \quad (8.268)$$

where there are $k - 1$ zeros following the r . Thus, any scalar value is considered a k -tuple with potentially only one non-zero entry. E.g., the value 3 is represented as $(3, 0, 0)$.

Thus, we can repeatedly “insert” elements $r_1, r_2, \dots, r_\ell \in \mathbb{R}_+$ (not necessarily sorted) into a k tuple by forming:

$$\overrightarrow{\mathbb{R}}_+^k(r_1) \oplus \overrightarrow{\mathbb{R}}_+^k(r_2) \oplus \dots \oplus \overrightarrow{\mathbb{R}}_+^k(r_\ell) = (r_{\sigma_1}, r_{\sigma_2}, \dots, r_{\sigma_k}) \quad (8.269)$$

where $r_{\sigma_1} \geq r_{\sigma_2} \geq \dots \geq r_{\sigma_k}$ and r_{σ_k} is not strictly less than any of the elements that may have been truncated from r_1, r_2, \dots, r_ℓ .

Note that the operation \oplus has a number of properties. For any $s_1, s_2, s_3 \in \overrightarrow{\mathbb{R}}_+^k$, we have that

$$s_1 \oplus s_2 \in \overrightarrow{\mathbb{R}}_+^k \quad (\text{closure}) \quad (8.270)$$

$$s_1 \oplus s_2 = s_2 \oplus s_1 \quad (\text{commutativity}) \quad (8.271)$$

$$(s_1 \oplus s_2) \oplus s_3 = s_1 \oplus (s_2 \oplus s_3) \quad (\text{associativity}) \quad (8.272)$$

Also, there exists an (“additive”, meaning based on operator \oplus) identity, call it \emptyset , such that

$$s_1 \oplus \emptyset = s_1 \quad (\text{identity element}) \quad (8.273)$$

with $\emptyset = (0, 0, \dots, 0)$.

Therefore, $(\overrightarrow{\mathbb{R}}_+^k, \oplus)$ itself forms a *commutative semi-group*. It is a commutative semi-group rather than a commutative group since we are not requiring an inverse element to exist (an inverse element would require that for all $s \in \overrightarrow{\mathbb{R}}_+^k$, $\exists \bar{s}$ such that $s \oplus \bar{s} = \emptyset$, something which clearly does not exist here due to the non-negativity and the max operations).

Given a set of elements $s_1, s_2, \dots, s_\ell \in \overrightarrow{\mathbb{R}}_+^k$, we can repeatedly compute \oplus as

$$\bigoplus_{i=1}^{\ell} s_i = s = (s^1, s^2, \dots, s^k) \quad (8.274)$$

in which case (s^1, s^2, \dots, s^k) is the k -tuple containing the k largest elements in the union of all of the elements in s_1, s_2, \dots, s_ℓ . This operation is well defined due to the above closure and associativity properties — each operation remains in $\overrightarrow{\mathbb{R}}_+^k$, and the order of the “sum” doesn’t matter.

Next, we define scalar multiplication. Given $r \in \mathbb{R}_+$, and $s \in \overrightarrow{\mathbb{R}}_+^k$, define the scalar left-multiplication as an operation $\mathbb{R}_+ \times \overrightarrow{\mathbb{R}}_+^k \rightarrow \overrightarrow{\mathbb{R}}_+^k$ called as follows:

$$r \cdot s = (rs^1, rs^2, \dots, rs^k) \quad (8.275)$$

I.e., we scalar multiply each element in s by r , and since $r \geq 0$, the order does not change. When clear from the context, we drop the \cdot notation as is done in standard scalar-vector multiplication, so that $r \cdot s = rs$.

For $s_1, s_2 \in \overrightarrow{\mathbb{R}}_+^k$, we also define standard vector addition as $s = (s^1, s^2, \dots, s^k) = s_1 + s_2$ where $s^i = s_1^i + s_2^i$. Since input operands are both sorted, so is result. I.e., since $s_1^1 \geq s_1^2$ and $s_2^1 \geq s_2^2$ we have $s^1 = s_1^1 + s_2^1 \geq s_1^2 + s_2^2 = s^2$. Note that $s_1 + s_2$ using the “+” operator is standard vector addition, while $s_1 \oplus s_2$ is the merge, sort, and truncate operation we mentioned above.

Note that for all $r_1, r_2 \in \mathbb{R}_+$, and $s_1, s_2 \in \overrightarrow{\mathbb{R}}_+^k$, we have that

$$r_1(s_1 \oplus s_2) = r_1s_1 \oplus r_1s_2 \quad \text{distributive property} \quad (8.276)$$

$$(r_1 + r_2)s_1 = (r_1s_1) + (r_2s_1) \quad \text{distributive property} \quad (8.277)$$

$$(r_1r_2)s_1 = r_1(r_2s_1) \quad \text{associativity} \quad (8.278)$$

We note that the system can be seen as what is known as a mathematical R -semi-module.

We now have all the tools necessary to compute the following quantity.

8.5.2 Finding the k best paths

We have an HMM of the form $p(x_{1:T}, q_{1:T})$. The Viterbi calculation is meant to compute

$$q_{1:T}^* \in \underset{q_{1:T}}{\operatorname{argmax}} p(\bar{x}_{1:T}, q_{1:T}) \quad (8.279)$$

but what if we want more than just the maximum assignment? I.e., like we saw, the most probable explanation might not be particularly typical, and one way to address this issue is to take the k top assignments. That is, we find a set of distinct assignments

$$q_{1:T}^*(1), q_{1:T}^*(2), \dots, q_{1:T}^*(k) \quad (8.280)$$

where

$$p(\bar{x}_{1:T}, q_{1:T}^*(1)) \geq p(\bar{x}_{1:T}, q_{1:T}^*(2)) \geq \dots \geq p(\bar{x}_{1:T}, q_{1:T}^*(k)) \quad (8.281)$$

and any other assignments not mentioned have score no higher than $p(\bar{x}_{1:T}, q_{1:T}^*(k))$.

Note that each of $p(\bar{x}_{1:T}, q_{1:T}^*(i))$ is just a scalar value in \mathbb{R}_+ , so therefore, we have that

$$\begin{aligned} \bigoplus_{q_{1:T} \in \mathcal{D}_{Q_{1:T}}} \overrightarrow{\mathbb{R}}_+^k(p(\bar{x}_{1:T}, q_{1:T})) \\ = (p(\bar{x}_{1:T}, q_{1:T}^*(1)), p(\bar{x}_{1:T}, q_{1:T}^*(2)), \dots, p(\bar{x}_{1:T}, q_{1:T}^*(k))) \end{aligned} \quad (8.282)$$

For simplicity, we will assume that a scalar $r \in \mathbb{R}_+$ will automatically graduate from a scalar into a k -tuple via the operator \oplus , so that $r_1 \oplus r_2 \equiv \overrightarrow{\mathbb{R}}_+^k(r_1) \oplus \overrightarrow{\mathbb{R}}_+^k(r_2)$.

The first thing to note, once again, that if $k = 1$, then this is just the score of the standard Viterbi (MPE) path. That is, we have given an exponential calculation that expresses the Viterbi path score. If $k > 1$ we have an exponential computation that expresses a vector consisting of the k highest scoring path scores. In either case, however, there is of course quite a bit of optimization we can do thanks to the aforementioned properties which allows for dynamic programming to be used.

Because of the distributed property above, we have that

$$\bigoplus_{q_{1:T} \in \mathcal{D}_{Q_{1:T}}} p(\bar{x}_{1:T}, q_{1:T}) \quad (8.283)$$

$$= \bigoplus_{q_{1:T} \in \mathcal{D}_{Q_{1:T}}} \prod_t p(x_t | q_t) p(q_t | q_{t-1}) \quad (8.284)$$

$$= \bigoplus_{q_T} p(x_T | q_T) p(q_T | q_{T-1}) \dots \left(\bigoplus_{q_2} p(x_2 | q_2) p(q_3 | q_2) \left(\bigoplus_{q_1} p(x_1 | q_1) p(q_2 | q_1) \right) \right) \quad (8.285)$$

$$= \bigoplus_{q_T} p(q_T | q_{T-1}) \dots \left(p(x_3 | q_3) \bigoplus_{q_2} p(q_3 | q_2) \left(p(x_2 | q_2) \bigoplus_{q_1} p(x_1 | q_1) p(q_2 | q_1) \right) \right) \quad (8.286)$$

$$(8.287)$$

Like in the case for the Viterbi path, we can define a k -best recursion of the following form: This is a form of k -best forward equations and we get the following recursion:

$$\alpha_1^{(k)}(q) = \overrightarrow{\mathbb{R}}_+^k(p(\bar{x}_1 | Q_t = q)) \quad (8.288)$$

and for $t > 1$:

$$\alpha_t^{(k)}(q) = p(\bar{x}_t | Q_t = q) \bigoplus_r p(Q_t = q | Q_{t-1} = r) \alpha_{t-1}^{(k)}(r) \quad (8.289)$$

It should be clear that $\alpha_t^{(k)}(q) \in \overrightarrow{\mathbb{R}}_+^k$ is a k -tuple for all q, t . Therefore, unlike in the standard Viterbi path case, at each time and state position (t, q) we need a tuple of size k to store the k values. Hence, $\{\alpha_t^{(k)}(q)\}_{t,q}$ constitutes a $N \times T \times k$ tensor.

In the Viterbi path case, the forward computation has as an interpretation a max marginal. That is,

$$\alpha_t^m(q_t) = \max_{q_{1:t-1}} p(q_{1:t-1}, q_t, \bar{x}_{1:t}) \quad (8.290)$$

where the states $q_{1:t-1}$ have been “max marginalized” out. This case corresponds to the k -best when $k = 1$. For general $k > 1$, the value of $\alpha_t^{(k)}(q)$ also has a probabilistic meaning, consisting of a length k list of probabilities of t states and t observations. Suppose that

$$\alpha_t^{(k)}(q_t) = (p(\bar{x}_{1:t}, q_{1:t-1}(1), q_t), p(\bar{x}_{1:t}, q_{1:t-1}(2), q_t), \dots, p(\bar{x}_{1:t}, q_{1:t-1}(k), q_t)) \quad (8.291)$$

Then we have that

$$p(\bar{x}_{1:t}, q_{1:t-1}(1), q_t) = \max_{q_{1:t-1}} p(q_{1:t-1}, q_t, \bar{x}_{1:t}) \quad (8.292)$$

which is the Viterbi score up to q_t at time t . We also have that

$$p(\bar{x}_{1:t}, q_{1:t-1}(1), q_t) \geq p(\bar{x}_{1:t}, q_{1:t-1}(2), q_t) \geq \dots \geq p(\bar{x}_{1:t}, q_{1:t-1}(k), q_t) \quad (8.293)$$

and moreover that $p(\bar{x}_{1:t}, q_{1:t-1}(k), q_t) \geq p(\bar{x}_{1:t}, q'_{1:t-1}, q_t)$ where $q'_{1:t-1}$ is any other sequence of states up to time $t-1$ that have not been included in the above. In other words, $\alpha_t^{(k)}(q)$ gives us the top k probabilities of sequences of states leading up to and ending at state q at time t . This is a generalization of the Viterbi score, $\alpha_t^m(q)$, which gives us the top one scoring sequence of states leading up to and ending at state q at time t .

At the end of the sequence (at time T) we have a set of $|D_Q|$ states each of which contains a k -tuple. If we then compute

$$\bigoplus_{q_T} \alpha_T^{(k)}(q_T) = (p(\bar{x}_{1:T}, q_{1:T}^*(1)), p(\bar{x}_{1:T}, q_{1:T}^*(2)), \dots, p(\bar{x}_{1:T}, q_{1:T}^*(k))) \quad (8.294)$$

we'll get the desired values, i.e., the scores of the k -best paths through the HMM.

8.5.3 From Scores to Paths

Now of course, we want more than the scores, we want the paths themselves, so how is this done? In the case of the standard max there is an argmax that returns the index of the maximum, and we can generalize \oplus as well in the same way. For lack of a better term, let's call this $\text{arg}\oplus$, pronounced “arg-k-max”. Note that $\text{arg}\oplus$ has an implicit (but not expressed) dependence on k .

If we have $s_1, s_2 \in \overrightarrow{\mathbb{R}}_+^k$, and $k = 1$, then

$$\text{arg}\oplus(s_1, s_2) \quad (8.295)$$

should produce the index, 1 or 2, depending on which of s_1 and s_2 contain the max. Generalizing this to $k > 1$, $\text{arg}\oplus$ should produce a set of k indices indicating where the k -largest values are. In this case, however, the values could come from either s_1 or s_2 , so the indices themselves have to be a 2-tuple, stating 1) which k -tuple (either s_1 or s_2) the entry lies in and 2) where in that selected k -tuple the entry is. In other words, $\text{arg}\oplus$ returns a list of pairs of integers, where each pair is a tuple-identifier i and a position $1 \leq j \leq k$ within that tuple. The list of pairs returned is ordered so that: 1) the first pair indexes the tuple and tuple entry of the largest element in the ordered lists of numbers; 2) the second pair indexes the tuple and tuple entry of the second largest element in the ordered lists of numbers; and so on. For example, suppose $k = 3$, then

$$\text{arg}\oplus((3.0, 2.0, 1.0), (6.0, 5.0, 2.0)) = \{(2, 1), (2, 2), (1, 1)\} \quad (8.296)$$

indicating that the maximum element within the two tuples is in the second tuple at position 1, the next highest element within the two tuples is in the second tuple in position 2, and the third largest element is in the first tuple at position 1.

Another example with $k = 4$ would be:

$$\arg\oplus((4.0, 3.0, 2.1, 0.5), (7.0, 6.0, 2.0, 1.0), (5.0, 2.0, 1.0, 1.0)) \quad (8.297)$$

$$= \{((2, 1), (2, 2), (3, 1), (1, 1))\} \quad (8.298)$$

indicating that the maximum element is in the first position of tuple two, the next largest element is in the second position of tuple two, the third largest element is in the first position in tuple one, and the forth largest element is in the first position of tuple one.

In the following example, with $k = 3$, we have more than one list that achieves the same max k values:

$$\arg\oplus((3.0, 2.0, 1.0), (6.0, 5.0, 3.0)) \quad (8.299)$$

$$= \{((2, 1), (2, 2), (1, 1)), ((2, 1), (2, 2), (2, 3))\} \quad (8.300)$$

Thus, $\arg\oplus$ should be interpreted as a set of lists, each list in the set is a list of pairs.

8.5.3.1 A recursion for computing the backtracking pointers during the forward pass

Consider now the following recursion, where at each step we are computing a set of k pairs of integers.

$$\check{\alpha}_t^{(k)}(q) \in \arg\oplus_r p(Q_t = q | Q_{t-1} = r) \alpha_{t-1}^{(k)}(r) \quad (8.301)$$

this recursion allows us to keep track at each time step and state where the k -best comes from leading to each state at each time. Note that the recursion in Equation (8.301) does not use the observation at time t — rather, it uses the observations at time $t - 1$. Therefore, in the final time step in $\check{\alpha}_T^{(k)}(q)$, we have not yet used the observations at time T , we'll use them below (in Equation (8.305)). Also note the use of “ \in ” in the argmax rather than “ $=$ ” — the reason for this is that there could be multiple k -tuples of pairs that achieve the k -maximum values (as we saw above), and the use of \in is to indicate that if there are more than one, we choose amongst them arbitrarily.

Collectively, $\left\{ \check{\alpha}_t^{(k)}(q) \right\}_{t \in \{1, 2, \dots, T\}, q \in \{1, 2, \dots, N\}}$ is a $|D_Q| \times T$ matrix of k -tuples of pairs (so $2kTN$ entries in total). Moreover, $\check{\alpha}_t^{(k)}(q)$ stores a set of k pairs, where the first element of every pair indexes a state's k -tuple at time $t - 1$, and the second element of every pair is an entry in the k -tuple at that state. The set of pairs at $\check{\alpha}_t^{(k)}(q)$ identify the location of the k largest items in the collective set of k -tuples in the previous time step that: 1) lead to entry (q, t) in the matrix; and 2) have accounted for the transition matrix $p(q|r)$ for r ranging over the first element in every pair $\check{\alpha}_t^{(k)}(q)$.

Thus, there is a size k set of pairs of values $\{(\check{q}^\ell, \check{k}^\ell)\}_{\ell=1}^k$ such that

$$\check{\alpha}_t^{(k)}(q) = \left((\check{q}^1, \check{k}^1), (\check{q}^2, \check{k}^2), \dots, (\check{q}^k, \check{k}^k) \right) \quad (8.302)$$

where $1 \leq \check{q}^i \leq |D_Q|$, $1 \leq \check{k}^i \leq k$. Also, we can denote quantities like the ℓ^{th} entry in this tuple of pairs using double arguments “ $(q)(\ell)$ ” as in:

$$\check{\alpha}_t^{(k)}(q)(\ell) \triangleq (\check{q}^\ell, \check{k}^\ell) \quad (8.303)$$

where $1 \leq \ell \leq k$, $1 \leq \check{q} \leq |D_Q|$, and $1 \leq \check{k} \leq k$. This notationally makes sense since $\check{\alpha}_t^{(k)}(q)$ is an array of pairs and $\check{\alpha}_t^{(k)}(q)(\ell)$ is the ℓ^{th} entry in this array, which itself is a pair (\check{q}, \check{k}) .

At any time t , a given state might be used more than once, and in fact a given state might be the only one that is used in a tuple (i.e., not only might it be used more than once, but it might be chosen as many as k times). That is, if

$$\check{\alpha}_t^{(k)}(q) = \left((q^1, \ell^1), (q^2, \ell^2), \dots, (q^k, \ell^k) \right) \quad (8.304)$$

then it might be that any subset of $\{q^\ell : \ell \in \{1, 2, \dots, k\}\}$ might be the same. If all the states are the same, then this means that all of the top k paths through the HMM that end at state q at time t must go through that state at time $t - 1$.

Another slightly subtle property, owing to the fact that the list of pairs point back to the maximum scoring k paths in descending sorted order, is the following. In the above, without loss of generality, we may always assume that $\ell^1 = 1$. That is, the first entry in the order at time t state q can always be set to point back to the top entry at time $t - 1$ at state q^1 . The reason is as follows. Suppose we had $\ell^1 > 1$. Since we know that $\alpha_{t-1}^{(k)}(q^1)(1) \geq \alpha_{t-1}^{(k)}(q^1)(\ell^1)$, the score will never go down if we set $\ell^1 = 1$. We know we can set $\ell^1 = 1$ since it is the element of the first pair in the list of pairs $\check{\alpha}_t^{(k)}(q)$.

We can in fact generalize this a bit further: consider $\check{\alpha}_t^{(k)}(q)$ again as a list of pairs as given in Equation (8.304). Let $j \in \{1, \dots, k\}$ be any index in this list corresponding to the first time that q^j has been used within the list $\check{\alpha}_t^{(k)}(q)$ (one assured case is $j = 1$). Then without loss of generality, we may assume that $\ell^j = 1$. The reason is the same as above: we do not decrease the score by pointing back to the first entry.

There are instances where we for each such j , we must have $\ell^j = 1$. If all the scores at $\alpha_{t-1}^{(k)}$ are unique (i.e., there are no score ties), then such j with $\ell^j > 1$ could not be the j^{th} maximum leading to state q at time t since a strictly better path can be obtained by pointing back with $\ell^j = 1$. I.e., the first time a state is encountered in the list, the element must index the first one.

As an example, suppose $k = 4$. When all scores are unique, we can never have as list $((5, 2), (5, 3), (5, 4), (6, 3))$ meaning that state 5 at time $t - 1$ starts with position 2, and the first time we encounter state 6 we use entry 3. This would be an error since it is always better to backtrack to position one of state 5 and position one at state 6 at time $t - 1$. One possible corrected instance would be $((5, 1), (5, 2), (5, 3), (6, 1))$. In other words, if we point back to a state, we must always first use the first (top, or most probable) element in the list of pairs.

Stated most generally, when all scores $\alpha_{t-1}^{(k)}$ are unique, then whenever we have $\ell^i > 1$ for some $i > 1$, then we must have all entries in $\check{\alpha}_{t-1}^{(k)}(q^i)$ from 1 to ℓ^i already pointed to earlier in the list of pairs $\check{\alpha}_t^{(k)}(q)$. When the scores are not unique, then whenever we have $\ell^i > 1$ for some $i > 1$, then (w.l.o.g.) we may assume that all entries in $\check{\alpha}_{t-1}^{(k)}(q^i)$ from 1 to ℓ^i area already pointed to earlier in the list of pairs $\check{\alpha}_t^{(k)}(q)$.

It turns out that this property can lead to memory savings. In each pair, we don't need to store the elements, only the states, since the elements are implied. That is, in any list of pairs, the first time a state is encountered the element is 1, the second time that state is encountered, the element is 2, and so on. Therefore, rather than storing a list of state-element pairs, such as $((5, 1), (5, 2), (2, 1), (5, 3), (2, 2), (6, 1), (2, 3), (6, 2))$, we can instead store a list of singleton states $((5), (5), (2), (5), (2), (6), (2), (6))$. More on this is discussed in §8.5.3.4.

8.5.3.2 After final time T recursion

After the recursion for the final time step T has been computed in Equation (8.301), we then compute

$$\check{\alpha}_T^{(k)} \in \arg \oplus \alpha_T^{(k)}(r) \quad (8.305)$$

which then gives a length- k list of pairs of states and positions, each of which refers to an entry at time T . The states in this list correspond to the states of the k maximum scoring paths through the HMM, and the index into the k -tuple gives the position within the k -tuple from which we can obtain the value. The first entry in this list corresponds to the Viterbi path. Note that the observation scores $(p(\bar{x}_T | q))$ over q) at time T are finally (implicitly) used in Equation (8.305) to start the backpointers, as they were not used in the computation of the backpointers in Equation (8.301). This is analogous to the standard Viterbi backtracking algorithm.

8.5.3.3 Backward pass for k -best backtracking

Given the time T list of k -tuples of state-index pairs $\check{\alpha}_T^{(k)}$, it is possible to produce a backtracking algorithm that is reminiscent of the 1-best Viterbi backtracking algorithm we have already seen. Recall, in the 1-best case, we had the final state q_T^* of the Viterbi path which was used to choose the backpointer to the penultimate state of the Viterbi path q_{T-1}^* via the backpointer at the location q_T^* , and this deterministic process is repeated, each state being used to look up the backpointer to the previous state until we have reached the beginning. As a reminder, we give this algorithm here.

```

1 Compute  $q_T^* \in \operatorname{argmax}_q \alpha_q^m(T)$ 
2 for  $t = T \dots 2$  do
3   Set  $q_{t-1}^* \leftarrow \check{\alpha}_{q_t^*}^m(t)$ 
```

The generalization to k -best now is fairly easy, using the notation that we have developed. At the final step T , we need to find the k top scoring positions, like above.

$$\left((q_T^{*1}, k_T^{*1}), (q_T^{*2}, k_T^{*2}), \dots, (q_T^{*k}, k_T^{*k}) \right) = \check{\alpha}_T^{(k)} \in \arg\oplus_r \alpha_T^{(k)}(r) \quad (8.306)$$

Once we have these k state-element pairs (each of which points to an entry in the collection of k -tuples at time T) we look up the index at each of those k state-element pairs, to get a new set of k state-element pairs. That is, we construct a new k -tuple of pairs as follows:

$$\left(\check{\alpha}_T^{(k)}(q_T^{*1})(k_T^{*1}), \check{\alpha}_T^{(k)}(q_T^{*2})(k_T^{*2}), \dots, \check{\alpha}_T^{(k)}(q_T^{*k})(k_T^{*k}) \right) \quad (8.307)$$

Note that each entry in this k -tuple is still a pair, but it a pair that points to an entry in the collection of k -tuples at time $T - 1$.

Each entry in a k -tuple at time $T - 1$ also contains a pair, so the above k -tuple of pairs can be converted into a new tuple of pairs.

$$\begin{aligned} & \left((q_{T-1}^{*1}, k_{T-1}^{*1}), (q_{T-1}^{*2}, k_{T-1}^{*2}), \dots, (q_{T-1}^{*k}, k_{T-1}^{*k}) \right) \\ &= \left(\check{\alpha}_T^{(k)}(q_T^{*1})(k_T^{*1}), \check{\alpha}_T^{(k)}(q_T^{*2})(k_T^{*2}), \dots, \check{\alpha}_T^{(k)}(q_T^{*k})(k_T^{*k}) \right) \end{aligned} \quad (8.308)$$

This can then be used to construct the k -tuple of pairs at time step $T - 2$ as follows:

$$\begin{aligned} & \left((q_{T-2}^{*1}, k_{T-2}^{*1}), (q_{T-2}^{*2}, k_{T-2}^{*2}), \dots, (q_{T-2}^{*k}, k_{T-2}^{*k}) \right) \\ &= \left(\check{\alpha}_{T-1}^{(k)}(q_{T-1}^{*1})(k_{T-1}^{*1}), \check{\alpha}_{T-1}^{(k)}(q_{T-1}^{*2})(k_{T-1}^{*2}), \dots, \check{\alpha}_{T-1}^{(k)}(q_{T-1}^{*k})(k_{T-1}^{*k}) \right) \end{aligned} \quad (8.309)$$

This process is then repeated back to $t = 1$, leading to the following k -best backtracking recursion:

```

1 Compute  $\left( (q_T^{*1}, k_T^{*1}), (q_T^{*2}, k_T^{*2}), \dots, (q_T^{*k}, k_T^{*k}) \right) = \check{\alpha}_T^{(k)} \in \arg\oplus_r \alpha_T^{(k)}(r)$ 
2 for  $t = T \dots 2$  do
3   Set  $\left( (q_{t-1}^{*1}, k_{t-1}^{*1}), (q_{t-1}^{*2}, k_{t-1}^{*2}), \dots, (q_{t-1}^{*k}, k_{t-1}^{*k}) \right) \leftarrow$ 
      $\left( \check{\alpha}_t^{(k)}(q_t^{*1})(k_t^{*1}), \check{\alpha}_t^{(k)}(q_t^{*2})(k_t^{*2}), \dots, \check{\alpha}_t^{(k)}(q_t^{*k})(k_t^{*k}) \right)$ 
```

To summarize the above, we describe the complete forward/backward algorithm for computing the k -top assignments in Algorithm 14.

Algorithm 14: A forward/backward algorithm for computing the k -best paths and their scores in an HMM

```

/* forward pass
1 for  $q = 1 \dots |\mathcal{D}_Q|$  do *
2    $\alpha_1^{(k)}(q) \leftarrow \overrightarrow{\mathbb{R}}_+^k(p(\bar{x}_1|Q_t = q))$  ;
3 for  $t = 2 \dots T$  do
4   for  $q = 1 \dots |\mathcal{D}_Q|$  do
5      $\alpha_t^{(k)}(q) \leftarrow p(\bar{x}_t|Q_t = q) \bigoplus_r p(Q_t = q|Q_{t-1} = r) \alpha_{t-1}^{(k)}(r)$  ;
6      $\check{\alpha}_t^{(k)}(q) \in \arg\bigoplus_r p(Q_t = q|Q_{t-1} = r) \alpha_{t-1}^{(k)}(r)$  ;
7    $\check{\alpha}_T^{(k)} \in \arg\bigoplus_r \alpha_T^{(k)}(r)$  ;
/* backward pass
8 Identify  $((q_T^{*1}, k_T^{*1}), (q_T^{*2}, k_T^{*2}), \dots, (q_T^{*k}, k_T^{*k})) = \check{\alpha}_T^{(k)}$  ;
9 for  $t = T \dots 2$  do
10  Set  $((q_{t-1}^{*1}, k_{t-1}^{*1}), (q_{t-1}^{*2}, k_{t-1}^{*2}), \dots, (q_{t-1}^{*k}, k_{t-1}^{*k})) \leftarrow$ 
     $(\check{\alpha}_t^{(k)}(q_t^{*1})(k_t^{*1}), \check{\alpha}_t^{(k)}(q_t^{*2})(k_t^{*2}), \dots, \check{\alpha}_t^{(k)}(q_t^{*k})(k_t^{*k}))$  ;

```

8.5.3.4 Resource Complexity

We analyze the computation resources (time and memory complexity) for Algorithm 14.

From a computational perspective, we are still doing one forward pass of T steps and one backwards pass of T steps, so the compute is still linear in T . Also, at each time step t we need storage for a fixed number ($|\mathcal{D}_Q|k$) of cells, so the memory is also still linear in T .

Now, however, at each forward pass we need to compute k top entries. Finding the single max entry at each state and time leads to a given per state cost of $O(|\mathcal{D}_Q|)$. Now, we see that we need, for each time t and state q in lines 5 and 6 of Algorithm 14, we need to find the k max entries out of a total of $|\mathcal{D}_Q|k$ entries in time $t - 1$.

One might think that to find the top k out of N entries would cost something like $N \log N$ or at least involve k , but it turns out we can find the top k out of N entries in $O(N)$ time (yes, linear in N independent of $k \leq N$) using a variant of the quick-sort algorithm. Such an algorithm finds the k -top entries in an arbitrary order (i.e., it doesn't find the k top in sorted order). Thus, if we wish to sort the top k items, we pay an additional $O(k \log k)$ (but typically $k \ll |\mathcal{D}_Q|$).

Hence, for each q, t we can find and then sort the top k entries out of the $|\mathcal{D}_Q|k$ entries at time $t - 1$ in $O(|\mathcal{D}_Q|k + k \log k)$ time. Therefore, the overall time-complexity is now $O(T(|\mathcal{D}_Q|^2k + |\mathcal{D}_Q|k \log k))$. If we can think of k as a constant (i.e., it does not change with the other parameters), then the time complexity is $O(T|\mathcal{D}_Q|^2)$ which is the same complexity as the 1-best algorithm.

On the other hand, at each (t, q) position, we need now to store a k -tuple and in each k -tuple entry we need three values (the score value and the two integers for the state-element backpointer). Therefore, the memory has gone from $O(T|\mathcal{D}_Q|)$ to $O(kT|\mathcal{D}_Q|)$, which is k -times worse. In cases when we are quite memory limited, then this can be a problem.

some variants: need not sort, rather can only compute top k at each step in arbitrary order, other than the very last step (and additive $k \log k$). This reduces the main complexity by a factor $k \log k$ but also means that we can't store only a list of states using the compression idea mentioned in the previous section, we always need to store a list of state-element pairs.

Note that there are other algorithms for computing k -best that do not require k times the memory, but

they do require k times the time-cost. But there are other time-space tradeoffs as well, such as the Island algorithm as we will see.

8.5.4 k -best without memory penalty

It is possible to develop another k -best algorithm for which the memory stays $O(TN^2)$ (no extra factor of k) but in such case we need to pay additional time cost, but the time cost does not increase by much, as we will see. The algorithm is based on the computation of the max marginals which we first review in the context of HMMs.

The max marginals over the hidden nodes in an HMM are defined as follows.

$$m_t(q) = \max_{q_{1:t-1}, q_{t+1:T}} p(q_{1:t-1}, Q_t = q, q_{t+1:T}, \bar{x}_{1:T}) = \max_{q_{1:T}|q_t=q} p(q_{1:T}, \bar{x}_{1:T}) \quad (8.310)$$

The max marginals over the node pairs, or graphical model edges, in an HMM are defined as follows.

$$m_{t,t+1}(q, q') = \max_{q_{1:t-1}, q_{t+2:T}} p(q_{1:t-1}, Q_t = q, Q_{t+1} = q', q_{t+2:T}, \bar{x}_{1:T}) \quad (8.311)$$

$$= \max_{q_{1:T}|(q_t, q_{t+1})=(q, q')} p(q_{1:T}, \bar{x}_{1:T}) \quad (8.312)$$

We will see how we can use these max marginals to produce the k -best paths, but first, how do we get the marginals? All of the below equations follow from the conditional independence properties of an HMM.

Recall the forward max equations from before.

$$\alpha_q^m(1) = p(\bar{x}_1|Q_1 = q) \quad (8.313)$$

$$\alpha_q^m(t) = p(\bar{x}_t|Q_t = q) \max_r p(Q_t = q|Q_{t-1} = r) \alpha_r^m(t-1) \quad (8.314)$$

and

$$\check{\alpha}_t^m(q) \in \operatorname{argmax}_r p(Q_t = q|Q_{t-1} = r) \alpha_r^m(t-1) \quad (8.315)$$

Recall also how we can view the $\alpha_q^m(t)$ as the max marginals up to time t , as in the following

$$\alpha_t^m(q) = p^m(\bar{x}_{1:t}, Q_t = q) = \max_{q_{1:t-1}} p(\bar{x}_{1:t}, q_{1:t-1}, Q_t = q) \quad (8.316)$$

We can also define a backwards recursion similar to the β calculation but where we replace the sum with the max as in the following.

$$\beta_T^m(q) = 1 \quad (8.317)$$

$$\beta_t^m(q) = \max_r p(\bar{x}_{t+1}|Q_{t+1} = r) p(Q_{t+1} = r|Q_t = q) \beta_{t+1}^m(r) \quad (8.318)$$

In this case, we have the following interpretation of $\beta_t^m(q)$

$$\beta_t^m(q) = \max_r p(\bar{x}_{t+1}|Q_{t+1} = r) p(Q_{t+1} = r|Q_t = q) \beta_{t+1}^m(r) \quad (8.319)$$

$$= \max_r p(\bar{x}_{t+1}|Q_{t+1} = r) p(Q_{t+1} = r|Q_t = q) \max_{q_{t+2:T}} p(x_{t+2:T}, q_{t+2:T}|Q_{t+1} = r) \quad (8.320)$$

$$= \max_{q_{t+1}} \max_{q_{t+2:T}} p(\bar{x}_{t+1}|Q_{t+1} = q_{t+1}) p(Q_{t+1} = q_{t+1}|Q_t = q) p(x_{t+2:T}, q_{t+2:T}|Q_{t+1} = q_{t+1}) \quad (8.321)$$

$$= \max_{q_{t+1:T}} p(x_{t+1:T}, q_{t+1:T}|Q_t = q) \quad (8.322)$$

Now consider the max marginals we defined above

$$m_t(q) = \max_{q_{1:T}|q_t=q} p(q_{1:T}, \bar{x}_{1:T}) \quad (8.323)$$

$$= \max_{q_{1:T}|q_t=q} p(q_{1:t}, \bar{x}_{1:t}, q_{t+1:T}, \bar{x}_{t+1:T}) \quad (8.324)$$

$$= \max_{q_{1:T}|q_t=q} p(\bar{x}_{t+1:T}, q_{t+1:T}|q_{1:t}, \bar{x}_{1:t}) p(q_{1:t}, \bar{x}_{1:t}) \quad (8.325)$$

$$= \max_{q_{1:T}|q_t=q} p(\bar{x}_{t+1:T}, q_{t+1:T}|q_t) p(q_{1:t}, \bar{x}_{1:t}) \quad (8.326)$$

$$= \max_{q_{t+1:T}|q_t=q} p(\bar{x}_{t+1:T}, q_{t+1:T}|q_t) \max_{q_{1:t}|q_t=q} p(q_{1:t}, \bar{x}_{1:t}) \quad (8.327)$$

$$= \beta_t^m(q) \alpha_t^m(q) \quad (8.328)$$

The pairwise (edge) max marginal is

$$m_{t,t+1}(q, q') = \max_{q_{1:T}|(q_t, q_{t+1})=(q, q')} p(q_{1:T}, \bar{x}_{1:T}) \quad (8.329)$$

$$= \max_{q_{1:T}|(q_t, q_{t+1})=(q, q')} p(q_{1:t-1}, q_t, q_{t+1}, q_{t+2:T}, \bar{x}_{1:t}, \bar{x}_{t+1}\bar{x}_{t+2:T}) \quad (8.330)$$

$$= p(x_{t+1}|q') \max_{q_{t+2:T}|q_{t+1}=q'} p(\bar{x}_{t+2:T}, q_{t+2:T}|q_{t+1}) p(q'|q) \max_{q_{1:t}|q_t=q} p(q_{1:t}, \bar{x}_{1:t}) \quad (8.331)$$

$$= p(x_{t+1}|q') \beta_{t+1}^m(q') p(q'|q) \alpha_t^m(q) \quad (8.332)$$

Therefore, either max marginal is easily obtained with the max versions of the forward and backward recursions.

Note that given the max marginals, it is quite easy to compute the Viterbi assignment. In fact, if we are certain that the Viterbi assignment is unique (i.e., all other state assignments have strictly lower probability), then the Viterbi path can be computed directly as follows:

$$\forall t, \{q_t^*(1)\} = \operatorname{argmax}_q m_t(q). \quad (8.333)$$

Why is it the case that this gives the Viterbi path in the case it is unique? This is because any other state sequence, with a strictly lower probability, cannot achieve the maximum value at each max marginal. To see this, let $p^1 = \max_{q_{1:T}} p(q_{1:T}, \bar{x}_{1:T})$ be the score value of the Viterbi path. Then we clearly have that, for any t ,

$$p^1 = \max_q m_t(q) = m_t(q_t(1)) \quad (8.334)$$

where $q_t(1) = \operatorname{argmax}_q m_t(q)$. Since the Viterbi path is unique, we must have that

$$m_t(q_t(1)) > m_t(q) \quad \forall q \neq q_t(1) \quad (8.335)$$

which means that no value other than $q_t(1)$ can be part of a Viterbi path. Hence, $q_t(1) = q_t^*(1)$ is the Viterbi path value at time t . Restating the above as a theorem, we have:

Proposition 129. *If the Viterbi path score is unique, meaning $p(\bar{x}_{1:T}, q_{1:T}^*(1)) > p(\bar{x}_{1:T}, q_{1:T}^*(k))$ for any $k > 1$, then we have the relationship:*

$$\forall t, \{q_t^*(1)\} = \operatorname{argmax}_q m_t(q). \quad (8.336)$$

If, on the other hand, the Viterbi assignment is not unique, we can easily obtain a Viterbi assignment from the max marginals. We first need the following:

Proposition 130. Let $q_t^*(1)$ be the state value of any Viterbi path of an HMM at time t . Then we can find a state value $q_{t+1}^*(1)$ at $t+1$ such that $(q_t^*(1), q_{t+1}^*(1))$ is a pair of state values of a Viterbi path using the following procedure:

$$q_{t+1}^*(1) \in \operatorname{argmax}_q m_{t,t+1}(q_t^*(1), q). \quad (8.337)$$

Proof. $\operatorname{argmax}_{r,q} m_{t,t+1}(r, q) = \{(r_i^*, q_i^*)\}_i$ is a set of pairs, each of which is compatible with some Viterbi path. Moreover, any Viterbi path must, at times t and $t+1$, have value corresponding to one of the pairs. Therefore, there is some i such that $q_t^*(1) = r_i^*$, and the argmax in Equation (8.337) chooses the corresponding q_i^* which hence is compatible with some Viterbi path. \square

We can therefore, in any case, obtain a Viterbi path via a forward pass (or backward pass for that matter) procedure to obtain one of the maximum scoring state sequences. We start this process with the singleton max-marginal on the left

$$q_1^*(1) \in \operatorname{argmax}_q m_1(q), \quad (8.338)$$

and then repeat the following recursion, for $t = 2 \dots T$, as follows

$$q_t^*(1) \in \operatorname{argmax}_q m_{t-1,t}(q_{t-1}^*(1), q), \quad (8.339)$$

which, thanks to to Proposition 130, is guaranteed to be a Viterbi path.

Is it possible to detect beforehand if the top one is unique? Indeed, given the marginals $m_t(q)$, if there are any values of q for any t that are not strictly largest, then there must be at least two Viterbi paths with the same probability score. Otherwise, there is one. The above is an iff condition, meaning the Viterbi path is unique iff each of set $\operatorname{argmax}_q m_t(q)$ is of size one.

Therefore, it seems the max marginals $m_t(q)$ and $m_{t,t+1}(q)$ have quite a bit of information in them. In the next algorithm, in fact, we show how to obtain the k -best paths using only $m_t(q)$ and $m_{t,t+1}(q)$. This means that, given these max marginals, we need not refer back to the HMM trellis as was done in §8.5.

Recall, we denote by $D_{Q_{1:T}}$ the set of all possible length- T state sequences and any state sequence is $q_{1:T} \in D_{Q_{1:T}}$

Suppose we have identified the Viterbi (1st best) path, i.e., $q_{1:T}^*(1) \in D_{Q_{1:T}}$ such that

$$p(\bar{x}_{1:T}, q_{1:T}^*(1)) \geq p(\bar{x}_{1:T}, q_{1:T}) \quad (8.340)$$

for all $q_{1:T} \in D_{Q_{1:T}} \setminus \{q_{1:T}^*(1)\}$. We know that the 2nd best path must exist within the set of sequences $D_{Q_{1:T}}(1) \triangleq D_{Q_{1:T}} \setminus \{q_{1:T}^*(1)\}$. That is, the 2nd best path $q_{1:T}^*(2)$ must have some difference with the 1st best path $q_{1:T}^*(1)$.

As a reminder to the reader, the i^{th} best path is denoted by

$$q_{1:T}^*(i) = (q_1^*(i), q_2^*(i), \dots, q_T^*(i)) \quad (8.341)$$

where $q_t^*(i)$ is the value of the state at time t of the i^{th} -best path.

The second best path must have at least one difference from the first best path. In order to find where that initial difference lies, we can partition $D_{Q_{1:T}}(1)$ into separate sets of paths based on where that happens. The first time (meaning the frame closest to $t = 1$) at which there is a difference between the best path $q_{1:T}^*(1)$ and second best path $q_{1:T}^*(2)$ may be either:

- at time 1,

- or at time 2 with there being no difference at time 1,
- or at time 3 with there being no difference either at time 1 or at time 2,
- ...
- or at time t with there being no difference from time 1 through time $t - 1$
- etc. up to time T .

That is, we partition $D_{Q_{1:T}}(1)$ into T blocks of sequences as follows:

$$D_{Q_{1:T}}(1, 1) = \{q_{1:T} \in D_{Q_{1:T}} : q_1 \neq q_1^*(1)\} \quad (8.342)$$

$$D_{Q_{1:T}}(1, 2) = \{q_{1:T} \in D_{Q_{1:T}} : q_1 = q_1^*(1), q_2 \neq q_2^*(1)\} \quad (8.343)$$

$$D_{Q_{1:T}}(1, 3) = \{q_{1:T} \in D_{Q_{1:T}} : q_1 = q_1^*(1), q_2 = q_2^*(1), q_3 \neq q_3^*(1)\} \quad (8.344)$$

$$\dots \quad (8.345)$$

$$D_{Q_{1:T}}(1, T) = \{q_{1:T} \in D_{Q_{1:T}} : q_1 = q_1^*(1), \dots, q_{T-1} = q_{T-1}^*(1), q_T \neq q_T^*(1)\} \quad (8.346)$$

It should be clear that each block of the partition does not restrict other locations where there might be a difference. That is, $D_{Q_{1:T}}(1, t)$ is the case where the first best path and the second best path are the same up to time $t - 1$, have their first difference at time t , but there could be zero or more differences at times $t' > t$. It should also be clear that this is a partition, meaning that $D_{Q_{1:T}}(1, t) \cap D_{Q_{1:T}}(1, t') = \emptyset$ for $t \neq t'$ and that $D_{Q_{1:T}}(1) = \bigcup_{t=1}^T D_{Q_{1:T}}(1, t)$, where the t^{th} block indicates where the difference lies between the top and second best path.

We note in passing that the sizes of the blocks are decreasing exponentially in t , but we do not use this fact.

It is possible to use the maximum marginals computed above to find the probability of the maximum state sequence within each of these blocks. Then the block that has maximum probability state sequence with maximum probability score is the block that contains the 2nd best state sequence. Once the block is known, we can then use the max marginals in a forward procedure to produce the actual second best state sequence. We outline the detailed steps next.

We first identify the probability of the maximum state sequence in each subset. We define for each $t \in \{1, 2, \dots, T\}$ the following:

$$p^m(D_{Q_{1:T}}(1, t)) \triangleq \max_{q_{1:T} \in D_{Q_{1:T}}(1, t)} p(q_{1:T}, \bar{x}_{1:T}) \quad (8.347)$$

These values can be computed simply from the max marginals as follows:

$$p^m(D_{Q_{1:T}}(1, 1)) = \max_{q | q \neq q_1^*(1)} m_1(q) \quad (8.348)$$

and for $t \in \{2, 3, \dots, T\}$

$$p^m(D_{Q_{1:T}}(1, t)) = \max_{q \neq q_t^*(1)} m_{t-1,t}(q_{t-1}^*(1), q) \quad (8.349)$$

The reason this last equation works is that $m_{t-1,t}(q, q')$ max-marginalizes over all states at time $t - 1$ and t . In block $D_{Q_{1:T}}(1, t)$, where all states are the same as the top $q_{1:T}^*(1)$ up to but not including time t , by maximizing over the remaining variable at time t , this corresponds to a maximization over every state under the constraint that $q_t \neq q_t^*(1)$.

Now given these maximization scores, it is easy to find the probability of the 2nd most probable state sequence by doing

$$p(q_{1:T}^*(2), \bar{x}_{1:T}) = \max_t p^m(\mathcal{D}_{Q_{1:T}}(1, t)) \quad (8.350)$$

Let us suppose that a time that achieved the maximum is $t^{(2)}$, meaning that

$$t^{(2)} \in \operatorname{argmax}_t p^m(\mathcal{D}_{Q_{1:T}}(1, t)). \quad (8.351)$$

In this case, it means that

$$p(q_{1:T}^*(2), \bar{x}_{1:T}) = p^m(\mathcal{D}_{Q_{1:T}}(1, t^{(2)})). \quad (8.352)$$

This information also helps us narrow down the actual second best assignment since we now know that $q_{1:T}^*(2) \in \mathcal{D}_{Q_{1:T}}(1, t^{(2)})$. In fact, we can exactly identify $q_{1:T}^*(2)$ using the following procedure:

$$q_t^*(2) \leftarrow q_t^*(1) \text{ for } t = 1, \dots, t^{(2)} - 1 \quad (8.353)$$

$$q_{t^{(2)}}^*(2) \in \operatorname{argmax}_{\substack{q \neq q_{t^{(2)}}^*(1)}} m_{t-1,t}(q_{t^{(2)}-1}^*(2), q) \quad (8.354)$$

$$q_t^*(2) \in \operatorname{argmax}_q m_{t-1,t}(q_{t-1}^*(2), q) \text{ for } t = t^{(2)} + 1, \dots, T \quad (8.355)$$

From this, we get the 2nd most probable assignment of state variables $q_{1:T}^*(2)$. Note that this is done using only the max marginals, so the memory cost is still $O(TN^2)$. However, to do second best, we needed to perform another $O(TN)$ pass through the set of max marginals.

To get the third most probable sequence $q_{1:T}^*(3)$ we do the following. We know that the 3rd best path must exist within the set of sequences $\mathcal{D}_{Q_{1:T}}(2) \triangleq \mathcal{D}_{Q_{1:T}} \setminus \{q_{1:T}^*(1), q_{1:T}^*(2)\} = \mathcal{D}_{Q_{1:T}}(1) \setminus \{q_{1:T}^*(2)\}$ and that the 3rd best path $q_{1:T}^*(3)$ must have some difference with both the first and the 2nd best paths.

Moreover, from our discussion of finding the 2nd best, we know that the 2nd best $q_{1:T}^*(2) \in \mathcal{D}_{Q_{1:T}}(1, t^{(2)})$ where $t^{(2)}$ is the index where there lies a difference between the first and second best path. Therefore, the third best is either in one of the original blocks $\{\mathcal{D}_{Q_{1:T}}(1, t)\}_{t \neq t^{(2)}}$ that did not contain the second best, or is in the same original block that contained the second best but of course must not equal the second best, so $q_{1:T}^*(3) \in \mathcal{D}_{Q_{1:T}}(1, t^{(2)}) \setminus \{q_{1:T}^*(2)\}$. In this latter case, the third best path must have at least one difference from both the first and second best path. In order to find where that initial difference lies, we can partition $\mathcal{D}_{Q_{1:T}}(1, t^{(2)})$ into separate sets of paths based on where that happens. Since $\mathcal{D}_{Q_{1:T}}(1, t^{(2)})$ is the same as first best up to but not including time $t^{(2)}$, there is no partition before time $t^{(2)}$. In other words, the first time (meaning the frame closest to $t = t^{(2)}$) at which there is a difference between the second best path $q_{1:T}^*(2)$ and third best path $q_{1:T}^*(3)$ may be either:

- at time $t^{(2)}$,
- or at time $t^{(2)} + 1$ with there being no difference at time $t^{(2)}$,
- or at time $t^{(2)} + 2$ with there being no difference either at time $t^{(2)}$, or at time $t^{(2)} + 1$,
- ...
- or at time $t > t^{(2)}$ with there being no difference from time $t^{(2)}$ through time $t - 1$
- etc. up to time T .

What we do, therefore, is partition $D_{Q_{1:T}}(1, t^{(2)})$ into $T - t' + 1$ subsets s

$$D_{Q_{1:T}}(1, t^{(2)}, t^{(2)}), D_{Q_{1:T}}(1, t^{(2)}, t^{(2)} + 1), \dots, D_{Q_{1:T}}(1, t^{(2)}, T) \quad (8.356)$$

where

$$D_{Q_{1:T}}(1, t^{(2)}, t^{(2)}) = \left\{ q_{1:T} \in D_{Q_{1:T}}(1, t^{(2)}) : q_{t^{(2)}} \neq q_{t^{(2)}}^*(2) \right\} \quad (8.357)$$

$$D_{Q_{1:T}}(1, t^{(2)}, t^{(2)} + 1) = \left\{ q_{1:T} \in D_{Q_{1:T}}(1, t^{(2)} + 1) : q_{t^{(2)}} = q_{t^{(2)}}^*(2), q_{t^{(2)}+1} \neq q_{t^{(2)}+1}^*(2) \right\} \quad (8.358)$$

$$\dots \quad (8.359)$$

$$D_{Q_{1:T}}(1, t^{(2)}, T) = \left\{ q_{1:T} \in D_{Q_{1:T}}(1, t^{(2)}) : q_{t^{(2)}} = q_{t^{(2)}}^*(2), \dots, q_{T-1} = q_{T-1}^*(2), q_T \neq q_T^*(2) \right\} \quad (8.360)$$

Note that for any $q_{1:T} \in D_{Q_{1:T}}(1, t^{(2)}, t)$ (where necessarily $t \geq t^{(2)}$), some prefix of $q_{1:T}$ would be the same as the first best path. In other words, we have another characterization of $D_{Q_{1:T}}(1, t^{(2)}, t^{(2)})$ as:

$$D_{Q_{1:T}}(1, t^{(2)}, t^{(2)}) = \left\{ q_{1:T} : q_1 = q_1^*(1), q_2 = q_2^*(1), \dots, q_{t^{(2)}-1} = q_{t^{(2)}-1}^*(1), q_{t^{(2)}} \neq q_{t^{(2)}}^*(1), q_{t^{(2)}} \neq q_{t^{(2)}}^*(2) \right\} \quad (8.361)$$

and also that for $t > t^{(2)}$,

$$D_{Q_{1:T}}(1, t^{(2)}, t) = \left\{ q_{1:T} : q_1 = q_1^*(1), q_2 = q_2^*(1), \dots, q_{t^{(2)}-1} = q_{t^{(2)}-1}^*(1), q_{t^{(2)}} \neq q_{t^{(2)}}^*(1), \right. \\ \left. q_{t^{(2)}} = q_{t^{(2)}}^*(2), q_{t^{(2)}+1} = q_{t^{(2)}+1}^*(2), \dots, q_{t-1} = q_{t-1}^*(2), q_t \neq q_t^*(2) \right\} \quad (8.362)$$

Thus, the blocks $\{D_{Q_{1:T}}(1, t)\}_{t \neq t^{(2)}}$ and $\{D_{Q_{1:T}}(1, t^{(2)}, t)\}_{t \geq t^{(2)}}$ constitute the partitioning of $D_{Q_{1:T}} \setminus \{q_{1:T}(1), q_{1:T}(2)\}$.

Our next task is to find the score of the most probable sequence within each of these blocks. In fact, we have already computed the probabilities of the most probable sequence for each of $\{D_{Q_{1:T}}(1, t)\}_{t \neq t^{(2)}}$. In order to get the probabilities of the most probable sequence in $\{D_{Q_{1:T}}(1, t^{(2)}, t)\}_{t \geq t^{(2)}}$, the following computations suffice. For a bit of notational simplicity, let's set $t' = t^{(2)}$ in the following. Then for $t = t' = t^{(2)}$ we have:

$$p^m(D_{Q_{1:T}}(1, t', t')) = p^m(D_{Q_{1:T}}(1, t')) \frac{\max_{q \notin \{q_{t'}^*(1), q_{t'}^*(2)\}} m_{t'-1, t'}(q_{t'-1}^*(2), q)}{m_{t'-1, t'}(q_{t'-1}^*(2), q_{t'}^*(2))} \quad (8.363)$$

and for $t > t^{(2)} = t'$

$$p^m(D_{Q_{1:T}}(1, t', t)) = p^m(D_{Q_{1:T}}(1, t')) \frac{\max_{q \neq q_t^*(2)} m_{t-1, t}(q_{t-1}^*(2), q)}{m_{t-1, t}(q_{t-1}^*(2), q_t^*(2))} \quad (8.364)$$

Therefore, the third most probable sequence has probability

$$p(q_{1:T}^*(3), \bar{x}_{1:T}) = \max \left\{ \max_{t: t \neq t'} p^m(D_{Q_{1:T}}(1, t)), \max_{t: t \geq t'} p^m(D_{Q_{1:T}}(1, t', t)) \right\} \quad (8.365)$$

The essential idea behind these equations is that if the third best assignment is within the same block as the second best, then the new max score $p(q_{1:T}^*(3), \bar{x}_{1:T})$ is almost the same as the old one $p(q_{1:T}^*(2), \bar{x}_{1:T})$ but with a little adjustment as given by the ratios. Also, we can identify it in a similar fashion to the way that we identified the second most likely, using again, only the maximum marginals. Once we have found the new block, we can use essentially the same procedure as before to identify the sequence. Namely, use the common part (that is the same as the other sequences) and do a step-by-step maximum dynamic programming procedure to generate the rest of the sequence to produce a max.

8.5.4.1 General case

The generalization to finding the k^{th} -best uses essentially the same pattern as above and in some sense gives us a iterative procedure to construct the next best. The basic pattern is: 1) look at the current partition of states sequences and find the block that contains the maximum; 2) partition that subset based on the set of possible differences; 3) make sure that we have the computed max over all new blocks; 4) find the block with the maximum score that is maximum; 5) use a dynamic programming recursion based on the max marginals to construct this new maximum assignment.

8.5.5 Analysis

	compute	memory
original algorithm	$O(kTN^2)$	$O(kTN^2)$
new algorithm	$O(TN^2 + kT \log(kT) + kTN)$	$O(TN^2 + kT)$

Table 8.2: Comparison of computational and memory resources needed by the two algorithms.

We first analyze the computational costs. We first need to construct the max marginals and that costs $O(TN^2)$ using the modified forward-backward max-marginal recursion described above. Once these marginals are computed, they are used over and over again in the computation.

At each iteration of the algorithm, we construct a new set of blocks. In the first iteration, we construct T blocks and in each further iteration, in the worst case, we construct an additional $O(T)$ blocks. The actual construction of the blocks is implicit as we never need to produce a data structure representing them. We do need, however, to find the score of the maximum path within each block, and this costs $O(N)$. Hence, the overall cost here is $O(kTN)$.

We need also to compute the block with the maximum score. At each iteration we have the previous set of blocks and we compute at most $O(T)$ new blocks. Hence, naïvely we would compute the max over all, at most, $O(kT)$ blocks repeatedly leading to a $O(k^2T)$ computation overall. However, we can use a smarter tree data structure (such as a 2-3 tree, or some form of priority queue) that keeps these scores in sorted order. The computation then becomes $O(kT \log(kT))$ in total.

Next, the dynamic programming that is used to generate the maximum assignments $q_{1:T}^*(i)$ themselves for $i \in \{1, 2, \dots, k\}$ each costs $O(TN)$, so overall the computational cost is $O(kTN)$.

Therefore, the overall computational costs of finding the k -best assignments is $O(TN^2 + kT \log(kT) + kTN)$. Notice that the k -dependence does not interact with the quadratic $O(N^2)$ cost to construct the max-marginals.

Regarding memory, the procedure requires at least $O(TN^2)$ because we need to store the maximum marginals $m_{t,t+1}(q, q')$ for every t . We also need to store the k -best sequences, which is an additional $O(T|\mathcal{D}_Q|k)$. At any given time, we have the current partition of $\mathcal{D}_{Q_{1:T}}$. At the first step, we have T blocks of $\mathcal{D}_{Q_{1:T}}$, and at the second step we have, in the worst case, $O(2T)$ blocks (i.e., this is the case that at each step, the difference is at time $t = 1$). In general, at the k^{th} step, we have $kT - k$ blocks. Each block must have its maximum score stored, so this is an additional $O(kT)$ memory cost. The overall memory therefore costs $O(TN^2 + kT)$.

A comparison of the time and memory requirements of this algorithm, and the one given in § 8.5 is given in Table 8.2.

8.6 Trading off speed and memory in dynamic inference: the Island algorithm

In standard forward/backward inference in an HMM, the time complexity is $O(TN^2)$ and the memory complexity is $O(TN)$. Often, this complexity is considered to be very good, and is owing to the treewidth of the HMM model — the maxclique sizes are two. For many applications, however, such as those in computational biology, T can be very large, much larger than what would fit in even modern machine memory. In other applications (such as speech recognition) N might be large, so having T copies of N is to large even if T is not that big. In this section, we talk about an algorithm that allows one to make tradeoffs between the computational and memory complexity in HMM (or any DGM) inference.

In general, one can reduce memory requirements by using more compute resources, and vice versa. The island algorithm is no exception. Indeed, we will see how it is possible to reduce the memory complexity of HMM inference to $O(N \log T)$ while the compute complexity increase to $O(N^2 T \log T)$. That is, we can reduce the memory requirements to be only $O(\log T)$ with the cost of an additional factor of $\log T$ compute complexity. Such a tradeoff might seem too good to be true, as the additional factor of computation $\log T$ is modest while the reduction factor in memory is large $T / \log T$. Many other variants are also possible as we explore below.

We note that it is also possible to achieve the same tradeoff with any dynamic graphical model. GMTK supports the island algorithm for DGMs, as we discuss in §

Basic idea of the island algorithm is the following: we divide a length T segment into b subsegments each of length T/b . During the forward pass, we delete memory for every frame of the forward pass except for “islands” that occur only every T/b frames. When we reach the last (right most) length- T/b subsegment, we perform a forward-backwards pass as normal within that subsegment, and then delete memory associated with this right-most subsegment. To be able to continue with the backwards pass on the penultimate (just left of the right-most) length- T/b subsegment, we need to re-do the forward pass and we can do that since we have stored its left-most frame — we start with that frame and move forward to the end of this penultimate subsegment. Then, we continue the backwards pass of the penultimate subsegment. When done, we delete the memory associated with the penultimate subsegment. To be able to continue the backwards pass with the antepenultimate subsegment (two subsegments to the left of the right-most subsegment), we repeat the process above which we can do since we have stored the left most frame (island) for the antepenultimate subsegment. We continue this process on each subsegment back to the beginning of the segment.

The temporal dependence in memory requirements for the algorithm so far presented include the cost of each of the islands, and there are b islands, and the cost of one of the subsegments T/b (since no more than one subsegment is ever stored at the same time), so the memory cost becomes $O(N(b + T/b))$ which is still linear in T but with a factor b reduction. The compute requirements have increased since we have needed to perform an additional $(b - 1)$ forward steps each of which costs T/b . Hence, the compute requirements have become $O(N^2((b - 1)T/b + T))$, again still linear in T but with an increase.

We are not yet done, however. Notice that the processing of each subsegment looks like a mini forward/backwards step on a length T/b sequence. Indeed, the island algorithm runs the same process recursively on each of these subsegments, thereby further dividing each of the length- T/b subsegments into subsubsegments of length T/b^2 . Then, each of the subsubsegments look again like a standard forward-backwards procedure and so can be still further subdivided into length- T/b^3 sequence. The recursion proceeds this until k recursion steps have been applied where subsegments have length T/b^k is less than some specified threshold ℓ (which can be user specified and can be as small as unity). Then, the recursion bottoms out and we perform a standard forward-backward sweep.

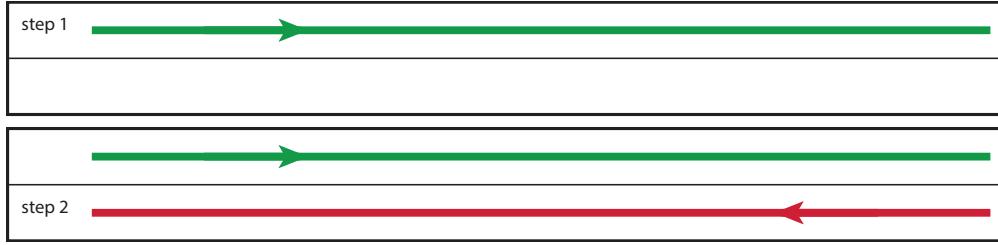


Figure 8.55: A figure showing the standard forward/backward algorithm. The forward algorithm (step 1) is shown in green and the backwards algorithm (step 2) is shown in red.

We next give more details regarding how this works. Recall (for convenience) the alpha-beta recursions:

$$\alpha_t(q_t) = p(\bar{x}_t|q_t) \sum_r p(q_t|r) \alpha_{t-1}(r) \quad (8.366)$$

$$\beta_t(q_t) = \sum_r p(r|q_t) p(\bar{x}_{t+1}|r) \beta_{t+1}(r) \quad (8.367)$$

Normally, we first do a complete length T left-to-right pass (α recursion) first as shown in step 1 in Figure 8.55. That is, and at each time t , we store information the complete message $\alpha_t(q_t)$ requiring $O(N)$ memory. We store it since to compute posteriors such as $p(q_t|\bar{x}_{1:T})$ we need both the forward and backwards values (cf. §8.4.11.5). Next, we do a right-to-left pass, updating that information in the trellis that we have just computed as shown in step 2 in Figure 8.55. We see that this takes compute $O(N^2T)$ and memory $O(TN)$ since at each time step we must store both the α and β values for each state, and we store these for all time steps simultaneously.

We describe the island algorithm in detail for $b = 2$. First, we do the first half of the forward recursion as shown in Step 1 in Figure 8.56. We then store an island as shown as a green oval in Figure 8.56-Step-2, and then free all memory done so far except for the island as shown in Figure 8.56-Step-3. We then run the forward recursion on the second half of the segment as shown in Figure 8.56-Step-4, and at this point we have run the recursion over the entire length- T sequence but we have memory allocated for only half of it.

Next, we start the backward pass and do the first half of the backwards pass as shown in Figure 8.56-Step-5. Once Step 5 is done, we have $\alpha_t(q)$ and $\beta_t(q)$ for t such that $T/2 \leq T \leq T/2$ and we can produce clique marginals $p(q_t|x_{1:T}) \propto \alpha_t(q_t)\beta_t(q_t)$ for the 2nd half of the segment (i.e., for t such that $T/2 \leq t \leq T$). We then store a backwards island shown as a red oval in Figure 8.56-Step-5. Next, we can free the memory used for everything except for the backwards island, as shown in Figure 8.56-Step-6.

Next, we start a forward pass for the second time. We only do the first half of the forward pass as shown in Figure 8.56-Step-7. Starting from the red island we previously stored, we then do the second half of the backwards pass as shown in Figure 8.56-Step-8 (because we stored the island, we can start the backwards pass at frame $T/2$ rather than T). At this point, we have $\alpha_t(q)$ and $\beta_t(q)$ for t such that $1 \leq T \leq T/2$ and we can produce posteriors for the 1st half of the sequence.

In each of the cases where we have the complete α and β values an entire subsegment, we see that it looks like a forward/backwards procedure but on a smaller segment. For example, Step 4 and Step 5 constitute a forward/backward procedure for frames $T/2 \leq t \leq T$, and step 7 and step 8 constitute a forward/backward procedure for frames $1 \leq t \leq T/2$. Hence, we can apply the process we just described recursively to each of these subsequences. Doing this recursively multiple times means we store a number of islands, one for each level of the recursion. Recursively applying it five times, and the corresponding islands that are stored during the forward pass, are shown in Figure 8.57.

Once we are at the deepest nesting level (because the subsegment length becomes 1, or because the subsegment length ℓ has fallen below a prespecified level), we then do the normal non-recursive forward-

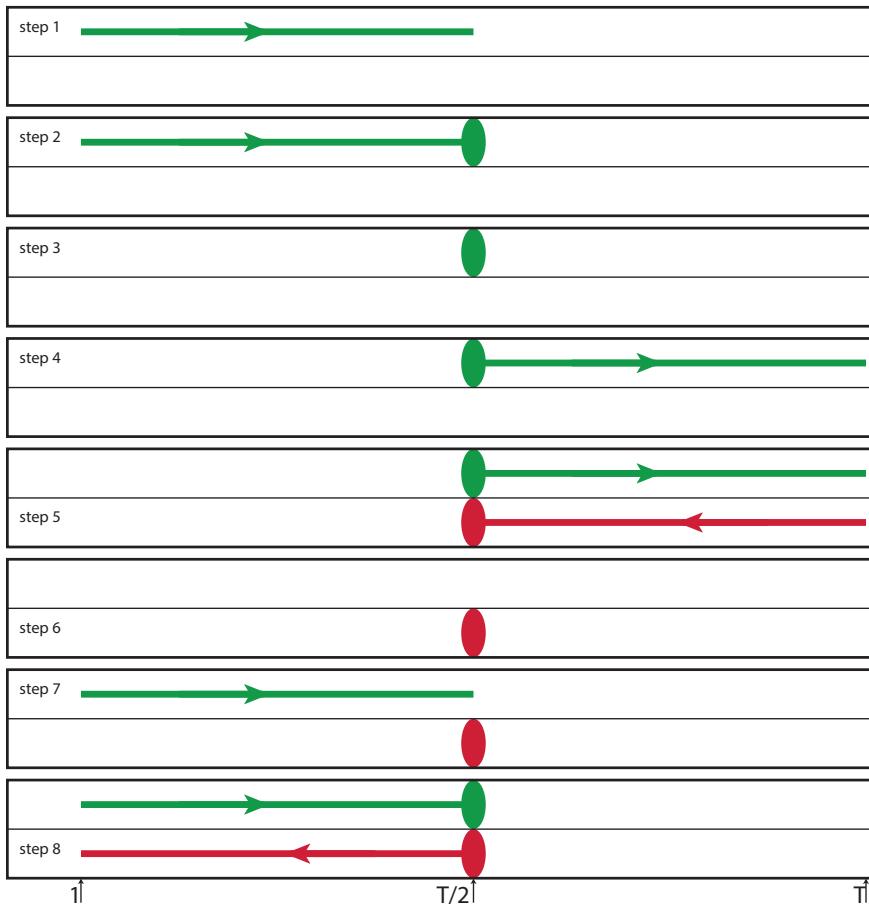


Figure 8.56: The island algorithm for $b = 2$, only the top-level recursion is shown, but we see that step 4 and step 5 constitute a standard forward/backward algorithm on a sequence of length $T/2$ and hence we can run the island algorithm recursively on the right-half subsegment. Steps 7 and 8 also constitute a length $T/2$ standard forward/backward algorithm, so again island can be applied recursively. Within each recursive application lies further standard forward/backwards procedures of successively shorter lengths $T/4$, $T/8$, and so on, until a base-case threshold is reached at which point standard forward-backward is applied.

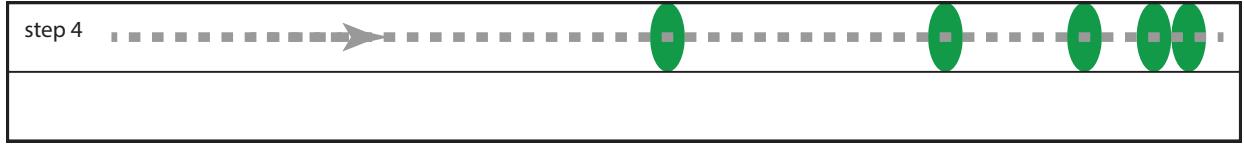


Figure 8.57: The forward pass of the island algorithm and the stored islands (shown as ovals) assuming a nesting depth of five.

backward algorithm which is the base case of the recursion. We then then move left, in chunks of size ℓ where at each case we do a normal forward-backward algorithm, and we repeat this until we have reached the beginning of the sequence.

Note that the island algorithm can of course be used for computing the Viterbi path (cf. §8.4.11.10), as well as the posteriors, and can also be used for generating the k -best paths.

In the next section we analyze this algorithm.

8.6.1 Island Algorithm Analysis

The island algorithm is a way to trade off time and memory and memory complexity when doing dynamic graphical model inference. The algorithm applies to any dynamic graphical model, but we explain it here using HMMs for simplicity. As mentioned above, GMTK supports the island algorithm for DGMs, as described in §

It is typically the case that forward-backward inference in an HMM has time cost $O(TN^2)$ and space cost $O(TN)$, where T is the number of steps and $N = |\mathcal{D}_Q|$. When T is very long (such as in biological sequences or in some speech recognition examples), the $O(TN)$ space might be prohibitive and it would be nice to be able to trade-off some time for some space, i.e., reduce the space cost at the expense of some time. It is important, of course, that the time cost not become prohibitive, however. The island algorithm solves this with quite a wide variety of flexibility.

Most simply, in the linear case we pay $O(T)$ memory and $O(T)$ compute (considering only the time cost for the moment). In the island case, we pay $O(b * \log_b(T))$ memory and $O(T \log_b(T))$ compute, where b is the base of the logarithm, is the number of subsegments a segment is divided into at each recursion level (we used $b = 2$ in Figure 8.56), and is also a crucial parameter in the island algorithm analysis (which we will see). We again note, all costs are of course multiplied by the average state cost, $O(|\mathcal{D}_Q|^2)$ in the case of time, and $O(|\mathcal{D}_Q|)$ in the case of space — we explain things only in terms of the time/space complexity in how it relates to the length of the segment T since the state-space cost does not change with the island algorithm. Still, adjusting the time complexity can make the difference between being able to use forward/backwards if in going from $O(T)$ to $O(\log T)$ we go from, say, *not* being able to fit in main memory to *being* able to fit in memory. Note, however, the additional time cost of $\log T$ can still be quite large. If say $T = 1024$, and $b = 2$, then we pay an extra time cost of a factor of 10 (e.g., 1 day to 10 days)!

We next analyze the algorithm in a bit more detail.

For a segment of length T , this algorithm never keeps simultaneously in memory more than about M expanded frames, where

$$M = T/b^k + k * (b - 1) \quad (8.368)$$

where k is the *smallest* integer value such that

$$T/b^k \leq \ell \quad (8.369)$$

and where ℓ is the linear segment threshold, i.e., the segment length threshold below which linear forward-backward inference is performed.

The way we arrive at these equations is as follows. We know that we keep a linear segment of length ℓ in memory, so we divide T into b^k chunks, and once the chunks are smaller than or equal to ℓ , we can stop dividing. We make k as small as possible in order to divide the length T segment into as few as possible chunks, thereby making the final linear chunks as large as possible, while still meeting the constraint that the chunk is $\leq \ell$. We can solve for k exactly, as $k = \lceil \log_b(T/\ell) \rceil$.

The quantity $k * (b - 1)$ is arrived at as follows: at the top level, we need $b - 1$ islands (since we don't store the right most one). At the next level, we store only islands at any given time for one of the length T/b segments, and within that segment we store, again at most $b - 1$ islands. This repeats, and since there are k levels (the depth is k), there is an additional memory cost of $k(b - 1)$. Figure 8.57 shows the case where $b = 2$ at a depth of $k = 5$.

Since k is such that $T/b^k \leq \ell$, we we keep in memory no more than about

$$M \leq \ell + k * (b - 1) \quad (8.370)$$

segments, so our memory storage requirements is $O(M)$. Note also that $k = \log_b(T/\ell)$. As an example, if $\ell = 3$, and $b = 3$, then we keep simultaneously in memory no more than about $3 + 2 \log_3(T/3)$ frames, which has only logarithmic growth in T . This is where we get the $O(\log_b(T))$ memory. In general, the memory that we need is

$$\ell + k(b - 1) = \ell + \log_b(T/\ell)(b - 1) \quad (8.371)$$

$$= \ell + b \log_b(T/\ell) - \log_b(T/\ell) \quad (8.372)$$

$$= O(\ell + b \log_b(T)) \quad (8.373)$$

frames.

As mentioned above, this memory savings doesn't come for free, as we pay a cost in time complexity. Specifically, we do multiple redundant forward α messages but we only do a single sweep of backwards β messages. That is, in the island algorithm we never perform redundant backwards messages, only redundant foreword messages. In the case of DGMs, unfortunately, it is typically the forward messages that are the more time costly of the two as we will explain in future chapters).

Thus, we perform analysis on the number of forward messages we must perform. Given a length T segment, we need to do $R(T)$ forward messages in total. We must at least do T forward messages since there is a sweep from $t = 1$ to $t = T$ as described above, but we also need to do $(b - 1)$ forward sweeps through each of the top-level subsegments of length T/b , and the number of forward messages needed for each subsegment is $R(T/b)$. Hence, we have the following recurrence relationship:

$$R(T) = T + (b - 1) * R(T/b). \quad (8.374)$$

To solve it, we can simplify by bounding $R(T)$ as follows

$$R(T) < T + b * R(T/b) \quad (8.375)$$

and then expanding out $R(T)$, we get

$$R(T) < \underbrace{T + T + \dots + T}_{d=\log_b(T) \text{ terms}} = O(T \log_b T) \quad (8.376)$$

This means that there are $\log_b(T)$ terms in the sum on the right. Note that each forward message (except for the islands themselves) must be computed $d = \log_b(T)$ times, which is what leads to the additional computational penalty factor of d . In general, we see that we must perform $O(T \log_b T)$ forward messages.

In actuality, there are k terms in the sum, where k is defined in Equation (8.369), since we stop when we've reached a threshold of ℓ .

From the above, we see we can decrease running time and increase space requirements either by increasing b (from 2 on up) or alternatively by increasing ℓ (from 2 on up). Note that we increase memory usage by increasing b since

$$O(b * \log_b(T)) = O((b / \ln(b)) \ln(T)) \quad (8.377)$$

so memory usage is growing as $b / \ln(b)$. Note also that one should set b and ℓ such that everything just fits in available main memory (i.e., so memory usage does not spill over and swap disk). In general, it is probably not worth it to set these so that it takes significantly less than the available main memory, as that would slow things down more than necessary.

How to set the island algorithm parameters ℓ and b ? There are in fact several points that are of interest. First, note that

$$\operatorname{argmin}_{b \in \{2,3,4,\dots\}} b / \ln(b) = 3 \quad (8.378)$$

so $b = 3$ is a good default value for b . Then one can set ℓ based on how much memory your machine has. In general, one would want to set ℓ to be as large as possible without causing the algorithm either to run out of memory or start swapping.

Another interesting strategy is to take $b = \sqrt{T}$, so in this case we use a different b for segments of different lengths. Then we get memory complexity of $O(b \times \log_b(T)) = O(\sqrt{T} \times 2) = O(\sqrt{T})$ memory while the compute becomes $O(T \log_b(T)) = O(2 \times T) = O(T)$ compute. In this case, therefore, we see that the algorithm takes at most twice as long but uses $2/\sqrt{T}$ as much memory. We say “at most” twice as long since computational redundancy exists only for the forward messages — as mentioned above, the backward messages (and any computation done once each forward and backward message is complete, such as computing a posterior or computing the Viterbi or k -best assignment) is done the same number of times (not to mention that the forward messages often dominate computation). Asymptotically, this means that there is an algorithm that is still linear in T while memory usage is only square-root in T .

More generally, for any $b = T^{1/d}$ (so that $d = \log_b T$), the algorithm has $O(T^{1/d})$ memory and $O(dT)$ compute. This means that, at least from the computational complexity point of view, the $O(T)$ point in the space-memory tradeoff curve is not optimal as we can reduce memory usage without (asymptotically in T) increasing time usage!! Note that in this case, the parameters of island would be ℓ, d (rather than ℓ, b) since in this case one would need the base b of the logarithm to change for different T while d would stay fixed for different segment lengths.

We note that GMTK's implementation of the island algorithm fro DGMs allows the user to specify either the parameter pair (ℓ, b) or the pair (ℓ, d) . See § for more information.

8.6.2 Island and k -best

Another useful property of the island algorithm is that the message definitions do not change. Therefore, under the simple generalization of the k -best algorithm that uses one forward-pass and one backwards pass. Normally, the k -best algorithm has time $O(N^2 k T)$ and memory $O(NTk)$. The island algorithm changes this so that we do the same underlying message steps but have time $O(N^2 k T \log_b(T))$ and space $O(k N b \log_b(T))$. If we take the $b = \sqrt{T}$ option mentioned above, then we've got an algorithm for k -best that does $O(N^2 k T)$ time and $O(k N \sqrt{T})$ space. And if k is no more than about \sqrt{T} , then we can produce a $O(N^2 k T)$ time, $O(NT)$ space algorithm for k -best (i.e., space independent of k).

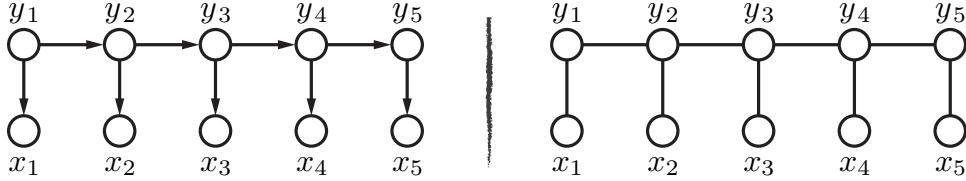


Figure 8.58: A: a Bayesian network view of an HMM. B: An undirected graphical model view of an HMM. The two views are identical (i.e., they convey the same family of models.)

8.7 What HMMs can do

HMMs are quite powerful. There are no independence assumptions in an HMM, meaning all variables are dependent on each other, either directly or indirectly. There are of course conditional independence properties in an HMM which is what makes using them computationally tractable. The reader is encouraged to read “what HMMs can do” [54] as in many cases, an HMM can do much more than what one might originally think.

8.8 Conditional Random Fields (CRFs)

Since about 2001, the Conditional Random Fields (CRFs) has been challenging the HMM as the preferred model for many sequential processing tasks. Indeed, owing to the CRFs inherently conditional nature, and it’s simple parameterization as a conditional log-linear model (which has allowed extremely simple and scalable training algorithms), the ease of use of discrete feature functions, the CRF is a good model to consider when one wishes to perform sequential decision making tasks. In this section, we describe CRFs and how they are different than (but really the same as) HMMs. In doing so, we also give a description of the label bias and the observation bias problems, and describe when they in fact are not problems.

8.8.1 The class of HMM models

As a reminder to the reader, an HMM is a joint distribution over $2T$ random variables $X_{1:T}$ and $Q_{1:T}$ that factorizes as follows:

$$p(x, q) = p(x_{1:T}, q_{1:T}) = p(x_1|q_1)p(q_1) \prod_{t=2}^T p(x_t|q_t)p(q_t|q_{t-1}) = \prod_t p(x_t|q_t)p(q_t|q_{t-1}) \quad (8.379)$$

What defines the HMM is this factorization property, and the underlying local conditional distributions $p(x_t|q_t)$ and $p(q_t|q_{t-1})$ (we assume that $p(q_1|q_0) = p(q_1)$).

Note that the above definition of an HMM is a “Bayesian network” definition, i.e., it uses locally normalized conditional distributions, so if we were to draw a graphical model for an HMM according to this definition, we would use a Bayesian network (which is of course directed), and would use Figure 8.58-A.

An HMM can equally correctly be defined as an Markov random field (MRF). I.e., a different (equivalent) definition of an HMM is that it is a joint distribution over $2T$ random variables $X_{1:T}$ and $Q_{1:T}$ that factorizes as follows:

$$p(x, q) = p(x_{1:T}, q_{1:T}) = \frac{1}{Z} \prod_{t=1}^T g(x_t, q_t) \prod_{t=2}^T h(q_t, q_{t-1}) = \frac{1}{Z} \prod_t g(x_t, q_t)h(q_t, q_{t-1}) \quad (8.380)$$

where

$$Z = \sum_{x_{1:T}, q_{1:T}} \prod_t g(x_t, q_t) h(q_t, q_{t-1}) \quad (8.381)$$

and where $g(\cdot, \cdot)$ and $h(\cdot, \cdot)$ are arbitrary non-negative functions over their two arguments (we assume for notational convenience that the function $h(q_1, q_0) = 1$ since this function doesn't really exist as there is no state q_0). In this definition of an HMM, the specification of $g()$ and $h()$ fully determines the HMM. An undirected graphical model view of an HMM is shown in Figure 8.58-B.

Note that these two definitions of an HMM do not at all mention how the individual factors ($p(x_t|q_t)$ and $p(q_t|q_{t-1})$ in the BN definition, or $g()$ and $h()$ in the MRF version) come to be. They could be formed by maximum likelihood training on some data, they could be formed by discriminative training on some data, or they even could be specified by hand – in either case the result is an HMM.

The *family* of models that constitute an HMM are those models that respect either of the above two definitions. Which of these definitions a given model respects doesn't matter since they are equivalent, as we next show.

8.8.2 Directed and Undirected HMMs are equivalent

In this section, we discuss how the family of HMM models described by a Bayesian network graph is the same as the family of models described by a Markov random field's undirected HMM graph. The two graphs are shown in Figure 8.10.

To show that the two families of models are equivalent (and hence the two definitions of HMMs are equivalent), we can show that starting from any instance in one family, we can construct a unique instance in the second family, and then starting with that instance in the second family and convert back to the first family, we end up where we started without any change.

A Boltzmann chain is an undirected graphical model view of an HMM.

Definition 131. A *Boltzmann-chain* (henceforth BC) is a joint distribution over $2T$ random variables $X_{1:T}$ and $Q_{1:T}$ that factorizes as follows:

$$p(q, x) = p(q_{1:T}, x_{1:T}) = \frac{1}{Z} \prod_{t=1}^T g(x_t, q_t) \prod_{t=2}^T h(q_t, q_{t-1}) \quad (8.382)$$

$$= \frac{1}{Z} \prod_t h(q_t, q_{t-1}) g(x_t, q_t) \quad (8.383)$$

where

$$Z = \sum_{x_{1:T}, q_{1:T}} \prod_t h(q_t, q_{t-1}) g(x_t, q_t) \quad (8.384)$$

and where $h(\cdot, \cdot)$ and $g(\cdot, \cdot)$ are non-negative functions of their respective arguments such that Z is finite.

Theorem 132. Let $\mathcal{F}_1 = \{p : p(x, q) = \prod_t p(q_t|q_{t-1})p(x_t|q_t)\}$ be the family of HMM distributions, and $\mathcal{F}_2 = \{p : p(x, q) = p(x_{1:T}, q_{1:T}) = \frac{1}{Z} \prod_t g(x_t, q_t)h(q_t, q_{t-1})\}$ be the family of “Boltzmann chains”, where g and h are arbitrary non-negative functions over the reals. Then $\mathcal{F}_1 = \mathcal{F}_2$.

We start with the easy case. We start with a Bayesian network representation of an HMM. To construct an MRF representation of an HMM, we set the MRF factors as follows:

$$h(q_t, q_{t-1}) \leftarrow p(q_t|q_{t-1}) \quad (8.385)$$

for $t > 1$ and for $t = 1$ we do

$$h(q_2, q_1) \leftarrow p(q_2|q_1)p(q_1). \quad (8.386)$$

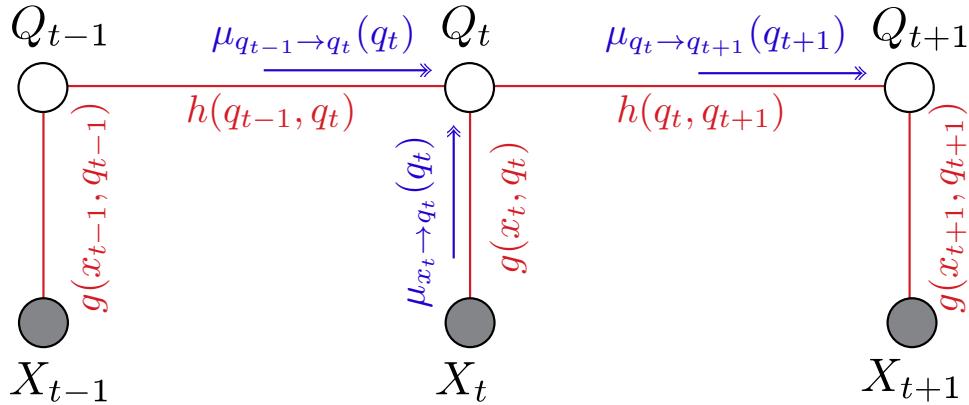


Figure 8.59: A belief-propagation message in an HMM.

Next, we set for all t

$$g(x_t, q_t) \leftarrow p(x_t | q_t). \quad (8.387)$$

This is valid since h and g can be any non-negative functions. With this setting of the g and h functions, we see that the resulting Z is such that $Z = 1$. Our HMM becomes:

$$p(x, q) = p(x_{1:T}, q_{1:T}) = \prod_t g(x_t, q_t)h(q_t, q_{t-1}) \quad (8.388)$$

To get back the original directed form, we just form the conditionals. I.e., we would form:

$$p(q_t | q_{t-1}) = \frac{p(q_t, q_{t-1})}{p(q_{t-1})} = \frac{\sum_{x_{1:T}} \sum_{\text{all } x \text{ and all } q \text{ but } q_t, q_{t-1}} \prod_t g(x_t, q_t)h(q_t, q_{t-1})}{\sum_{x_{1:T}} \sum_{\text{all } x \text{ and all } q \text{ but } q_{t-1}} \prod_t g(x_t, q_t)h(q_t, q_{t-1})} \quad (8.389)$$

$$= \frac{\sum_{x_{1:T}} \sum_{\text{all } x \text{ and all } q \text{ but } q_t, q_{t-1}} \prod_t p(x_t | q_t) p(q_t | q_{t-1})}{\sum_{x_{1:T}} \sum_{\text{all } x \text{ and all } q \text{ but } q_{t-1}} \prod_t p(x_t | q_t) p(q_t | q_{t-1})} \quad (8.390)$$

$$(8.391)$$

This returns the original Bayesian network HMM's factor $p(q_t | q_{t-1})$. The same goes for $p(q_1)$ and $p(x_t | q_t)$, so we have lost nothing going to a Markov random field and back to Bayesian network.

The next case is the harder of the two. We start with an instance that factors w.r.t. a Markov random field HMM, convert it to an instance that factors w.r.t. a Bayesian network HMM, and then convert it back an instance that factors w.r.t. a MRF. We will see that the final instance is identical to the original one, so nothing is lost in the conversion.

We start with an MRF instance:

$$p(x, q) = p(x_{1:T}, q_{1:T}) = \frac{1}{Z} \prod_t g(x_t, q_t)h(q_t, q_{t-1}) \quad (8.392)$$

We borrow a bit of notation from belief-propagation. I.e., in particular, we will use messages of the form

$$\mu_{q_t \rightarrow q_{t+1}}(q_{t+1}) \quad (8.393)$$

which is the standard belief-propagation (BP) message from node Q_t to node Q_{t+1} in an undirected graphical model with the condition that the message-passing protocol (MPP) is obeyed (i.e., all messages other than the one from Q_{t+1} have already arrived into Q_t before the message is sent out to Q_{t+1}). This notion

gives us a convenient way of depicting a cascade of marginalizations over the HMM in a particular direction. For example, this message is defined in an MRF-HMM as:

$$\mu_{q_t \rightarrow q_{t+1}}(q_{t+1}) = \sum_{q_t} \mu_{x_t \rightarrow q_t}(q_t) \mu_{q_{t-1} \rightarrow q_t}(q_t) h(q_t, q_{t+1}) \quad (8.394)$$

which is also shown in Figure 8.59. This message is in fact one of the forward recursions (e.g., like the α -recursion) part of the forward-propagation (alpha-computation), but using this notation allows us to talk about both forward and backward recursion messages in a notationally unified way. See §8.4.11 for discussion on these recursions. Thus, we refer to the backward messages as

$$\mu_{q_{t+1} \rightarrow q_t}(q_t). \quad (8.395)$$

We note that each message depends on other messages. The base case (for which there no longer is such a dependency) is at the far left end $t = 1$ and far right ends $t = T$ of the model.

Now that we have BP messages available to us, we can convert the MRF-HMM to marginals over smaller sets of variables, combine them all together, and we'll see that we get back the original undirected formulation. Specifically, we start with the MRF-HMM's joint distribution $p(x, q)$. From this joint distribution, we form all the necessary conditionals for the BN-HMM (i.e., $p(q_1)$, $p(q_t | q_{t-1})$, and $p(x_t | q_t)$ for all t), multiply them all together, and we get back the original MRF-HMM.

We start with $p(q_1)$.

$$p(q_1) = \frac{1}{Z} \mu_{x_1 \rightarrow q_1}(q_1) \mu_{q_2 \rightarrow q_1}(q_1) \quad (8.396)$$

I.e., we send a message from X_1 to Q_1 and the message from Q_2 to Q_1 to get this marginal. Recall, we are assuming that MPP is obeyed so that $\mu_{q_2 \rightarrow q_1}(q_1)$ can't be formed until q_2 has received messages from its right neighbors, and so on. Hence, $\mu_{q_2 \rightarrow q_1}(q_1)$ contains information from time points to the right of $t = 2$.

Next, we construct the Markov chain conditionals:

$$p(q_t | q_{t-1}) = \frac{p(q_t, q_{t-1})}{p(q_{t-1})} \quad (8.397)$$

$$= \frac{\frac{1}{Z} \mu_{q_{t-2} \rightarrow q_{t-1}}(q_{t-1}) \mu_{x_{t-1} \rightarrow q_{t-1}}(q_{t-1}) h(q_{t-1}, q_t) \mu_{x_t \rightarrow q_t}(q_t) \mu_{q_{t+1} \rightarrow q_t}(q_t)}{\frac{1}{Z} \mu_{q_{t-2} \rightarrow q_{t-1}}(q_{t-1}) \mu_{x_{t-1} \rightarrow q_{t-1}}(q_{t-1}) \mu_{q_t \rightarrow q_{t-1}}(q_{t-1})} \quad (8.398)$$

$$= \frac{h(q_{t-1}, q_t) \mu_{x_t \rightarrow q_t}(q_t) \mu_{q_{t+1} \rightarrow q_t}(q_t)}{\mu_{q_t \rightarrow q_{t-1}}(q_{t-1})} \quad (8.399)$$

Last, we construct the observation distributions:

$$p(x_t | q_t) = \frac{p(x_t, q_t)}{p(q_t)} \quad (8.400)$$

$$= \frac{\frac{1}{Z} \mu_{q_{t-1} \rightarrow q_t}(q_t) g(x_t, q_t) \mu_{q_{t+1} \rightarrow q_t}(q_t)}{\frac{1}{Z} \mu_{q_{t-1} \rightarrow q_t}(q_t) \mu_{x_t \rightarrow q_t}(q_t) \mu_{q_{t+1} \rightarrow q_t}(q_t)} \quad (8.401)$$

$$= \frac{g(x_t, q_t)}{\mu_{x_t \rightarrow q_t}(q_t)} \quad (8.402)$$

Clearly, each of the above factors is locally normalized, and when we multiply them all together, we will get an HMM, but which one? We will multiply them using the form:

$$p(x, q) = p(x_1 | q_1) p(q_1) \prod_{t=2}^T p(q_t | q_{t-1}) p(x_t | q_t) \quad (8.403)$$

The question is, what HMM will we end up with? Lets try:

First note that using these definitions of $p(q_t|q_{t-1})$, we get for time $2 \dots T$:

$$\prod_{t=2}^T p(q_t|q_{t-1}) = \frac{1}{\mu_{q_2 \rightarrow q_1}(q_1)} \prod_{t=2}^T h(q_{t-1}, q_t) \mu_{x_t \rightarrow q_t}(q_t) \quad (8.404)$$

which holds because $\mu_{q_{T+1} \rightarrow q_T}(q_T) = 1$ (i.e., this message really doesn't exist and is notated for convenience).

Next, we multiply in the observation distributions for time $2 \dots T$.

$$\left(\prod_{t=2}^T p(q_t|q_{t-1}) p(x_t|q_t) \right) = \left(\prod_{t=2}^T p(q_t|q_{t-1}) \right) \left(\prod_{t=2}^T p(x_t|q_t) \right) \quad (8.405)$$

$$= \left(\frac{1}{\mu_{q_2 \rightarrow q_1}(q_1)} \prod_{t=2}^T h(q_{t-1}, q_t) \mu_{x_t \rightarrow q_t}(q_t) \right) \left(\prod_{t=2}^T \frac{g(x_t, q_t)}{\mu_{x_t \rightarrow q_t}(q_t)} \right) \quad (8.406)$$

$$= \frac{1}{\mu_{q_2 \rightarrow q_1}(q_1)} \prod_{t=2}^T h(q_{t-1}, q_t) g(x_t, q_t) \quad (8.407)$$

We also have that:

$$p(q_1)p(x_1|q_1) = \frac{1}{Z} g(x_1, q_1) \mu_{q_2 \rightarrow q_1}(q_1) \quad (8.408)$$

Putting together Equation (8.407) with Equation (8.408), we have that:

$$p(x_1|q_1)p(q_1) \prod_{t=2}^T p(q_t|q_{t-1}) p(x_t|q_t) = \frac{1}{Z} g(x_1, q_1) \prod_{t=2}^T h(q_{t-1}, q_t) g(x_t, q_t) \quad (8.409)$$

Hence, we have exactly recovered the MRF representation of the HMM by forming the BN of the HMM. Therefore, the class of models represented by a BN description of an HMM is identical to the class of models representable by a MRF HMM.

8.8.3 When might one wish to use directed vs. undirected models?

Before we move on, we attempt in this section to address the question of which description should we prefer, a Bayesian network or an Markov random field description of HMMs.

If we are only interested in a mathematical explanation of the models, or if we are interested only in understanding the underlying complexity associated with computing, say, any of the standard forward-backward quantities associated with an HMM (cf. §8.4.11), then there is no reason to prefer one over the other. As mentioned above, they are identical.

There are, however, various potential practical differences between the two (some of these differences are identical to the practical differences between general Bayesian networks and Markov random fields). The main difference has to do with the issue of locally normalized factors vs. non-locally-normalized factors.

Consider a vector of random variables X and a factor $f(x)$ which in an MRF does not need to be normalized. It is common to use log-linear factors of the form $h(x) = e^{-\lambda^T f(x)}$ where $\lambda^T f(x)$ is an energy function, and $f(x) \in \mathbb{R}^M$ is a (potentially very long) length- M vector of features (i.e., $f(x) = [f_1(x) f_2(x) \dots f_M(x)]$) and $\lambda \in \mathbb{R}^M$ is an equal length vector of parameter values. It is possible in this case to choose a large ($M \gg 1$) and a diverse set of feature functions with relative impunity since regularized training algorithms (such as ℓ_2) can often pick and choose which of those features are important — certain regularization approaches (such as the use of an ℓ_1 regularizer, or equivalently utilizing a Laplacian prior

[187, 261, 433, 4, 194, 287]) prefer sparse solutions so that many of values of λ end up being zero. Moreover, such training procedures seem relatively insensitive to redundancy within the feature vectors and therefore make the model designer's job easier (i.e., one can add to $f()$ a large assortment of features without much concern about if they are redundant with respect to each other, if the ones that are not needed end up having a corresponding zero-valued parameter, and if it is not computationally infeasible to do the learning to determine which parameters should end up being zero).

With undirected models, it is also quite easy to represent constraints as in a SAT or CSP system [108]. For example, we can easily add a factor $\phi(x, y)$ that insists that only events where $|x - y| = c$ are valid, where c is some constant. Alternatively, one can add equality constraints, where $\phi(x, y) = \mathbf{1}(x = y)$. Equivalently, one of the features added to $h()$ above might take value ∞ for some of the random variable values and this has the effect of inducing a zero probability in $f(x)$.

On the other hand, there are cases where having directed, locally normalized, factors can be quite useful in practice. For example, directed models are quite useful for expressing sequencers, where the hidden Markov chain $p(q_t|q_{t-1})$ must go through a particular sequence of states, and where we wish to express locally what the probability of going from one state to the next state is. Knowing locally the transition probabilities allows us to compare transitions out of different states. For example, a scientist might wish to know how the absolute probability of going from state 3 to state 4 compares to the absolute probability of going from state 5 to state 4 — if the transition behavior is encoded with an unnormalized factor function $h(q_t, q_{t-1})$ along with parameters λ , these probabilities will depend on what the rest of the model is doing and can only be known by performing inference, effectively recovering the $p(q_t|q_{t-1})$ as we did above in Equation (8.389).

Moreover, visualizing the state transition matrix as a directed graph (quite distinct from a graphical model) is a popular way to depict not just Markov chains but also the underlying hidden Markov chain in an HMM. More generally, stochastic finite-state automata (SFSA), where the manipulation of state-transition graphs via a variety of operations [307, 308] are part of the design process in sequence modeling, are a popular form of sequence model (both in the speech and language world, and in the bioinformatics literature). We saw many examples of this starting in §8.3.

Normalized factors are also useful for expressing counters and length distributions in dynamic Bayesian networks. I.e., one factor might provide a probability distribution over a length $p(l)$ which initiates a counter chain that counts down to zero which causes another length hypothesis to be generated. We will see many examples of this in Chapter ??.

Another important advantage of directed factors is that each factor can be trained separately and then merged into the same model after training. For example, language models $p(w_t|w_{t-1}, w_{t-2})$ are often trained on a large body of text including not only speech corpus data but also web-collected data. Such models are often “backoff-based” [36] and require all variables during training to be observed. These models are then combined together in a single model *after* training has occurred. That is, each factor in a Bayesian network might be trained on different data sets, since there might not be one joint data set that contains joint data for all variables. To make sure this is clear, let's suppose we are interested in training $p(a, b, c, d) = p(a|b, c)p(b|c, d)p(c|d)p(d)$. We might have a set of four data sets $\mathcal{D}_1 = \{(a_i, b_i, c_i)\}_{i=1}^{K_1}$, $\mathcal{D}_2 = \{(b_i, c_i, d_i)\}_{i=1}^{K_2}$, $\mathcal{D}_3 = \{(c_i, d_i)\}_{i=1}^{K_3}$, and $\mathcal{D}_4 = \{(d_i)\}_{i=1}^{K_4}$, where samples from each data set are different and were the sizes might also be quite different. With a Bayesian network, factor $p(a|b, c)$ can be trained using D_1 , factor $p(b|c, d)$ can be trained using D_2 , and so on. Once these factors are trained, we can use them simultaneously in our Bayesian network model for $p(a, b, c, d)$. This is not possible with a Markov random field since the local non-normalized factors are balanced at the global level only after joint training. For example, suppose we wished to construct a MRF for $p(a, b, c, d) = \frac{1}{Z} \exp(\lambda_1 \phi(a, b, c) + \lambda_2 \phi(b, c, d) + \lambda_3 \phi(c, d) + \lambda_4 \phi(d))$. It would not be possible to use the \mathcal{D}_i data sets to separately train λ_i in this case.

It can also be argued that directed models are useful for encoding high level knowledge from a domain

expert. In some cases, perhaps due to the pseudo-causal interpretation of BN factors, it might be easy to elicit cause-and-effect-like knowledge from an expert in determining the factors in a BN. This might, moreover, make specifying $P(Q_t|Q_{t_1})$ easier. The expert knowledge could also be used to specify priors over the model when training it in a Bayesian context. In fact, directed models are often useful when one wishes to better understand and represent the underlying causality behind a process [338]. While BNs are not a perfect representation of causal phenomena, they are perhaps an easier framework to utilize when causal reasoning is the goal.

Because of the strong practical benefits of both directed and undirected models for articulatory modeling, we conclude that both representations are useful in practical settings.

8.8.4 The class of CRF models

A linear-chain CRF (henceforth CRF) is a conditional distribution over $2T$ random variables $X_{1:T}$ and $Q_{1:T}$ that factorizes as follows:

$$p(q|x) = p(q_{1:T}|x_{1:T}) = \frac{1}{Z(x)} \prod_{t=1}^T g(x_t, q_t) \prod_{t=2}^T h(q_t, q_{t-1}) = \frac{1}{Z(x)} \prod_t h(q_t, q_{t-1})g(x_t, q_t) \quad (8.410)$$

where

$$Z(x) = \sum_{q_{1:T}} \prod_t h(q_t, q_{t-1})g(x_t, q_t) \quad (8.411)$$

where $h(\cdot, \cdot)$ and $g(\cdot, \cdot)$ are arbitrary finite non-negative functions of their respective arguments (like in the MRF HMM above, we assume for notational convenience that $h(q_1, q_0) = 1$). It is these two functions that fully define the CRF. I.e., if we specify these two functions in this way, multiply them all together, and then conditionally normalize them with $Z(x)$, we have a (linear-chain) CRF. In practice, the functions h and g are formed by a combination of hand-specifying a set of feature functions, and then forming linear weights (in a log-linear model) using some form of conditional training (conditional maximum likelihood, max-margin, etc.). Alternatively, we could specify g and h by hand, normalize according to $Z(x)$, and we'd have a CRF.

Linear chain CRFs are also sometimes defined with three-way interacting factors, of the form:

$$p(q_{1:T}|x_{1:T}) = \frac{1}{Z(x)} \prod_{t=1}^T g(x_t, q_t, q_{t+1}) \quad (8.412)$$

where $g(x_T, q_T, q_{T+1}) = g(x_T, q_T)$ is a function of only two variables since q_{T+1} doesn't exist. Note also $Z(x)$ here can be computed just as efficiently using forward-backward due to these factorization assumptions as in the previous case, since the maximum clique size among the hidden variables is no larger in this case than in the previous case. In fact, it is fairly easily shown that, in the HMM case, models where the observation is a function of the transition are identical to models where the observation is a function of the state [206]. We discuss this further in §8.4.4.7.1.

Note that there is not a graphical model that accurately describes a complete CRF. A graphical model (either undirected or directed) gives the factorization of a family of *joint* probability distributions [258], where the factorization is precisely defined by graph-theoretic properties of the graph. A CRF is only a conditional distribution. In other words, a graph can be used so that, conditioned on $x_{1:T}$, the variables $q_{1:T}$ factor with respect to this graph. It is therefore not technically correct to show a graphical model for a CRF that includes both $q_{1:T}$ and $x_{1:T}$. Figure 8.60 shows a few imperfect attempts. One can, however, just give a Markov chain MRF showing the factorization of $q_{1:T}$ implicitly assumed to be conditioned on $x_{1:T}$, as done in Figure 8.60-(d), but this is still not without problems. In general, in a CRF, we can not say anything about

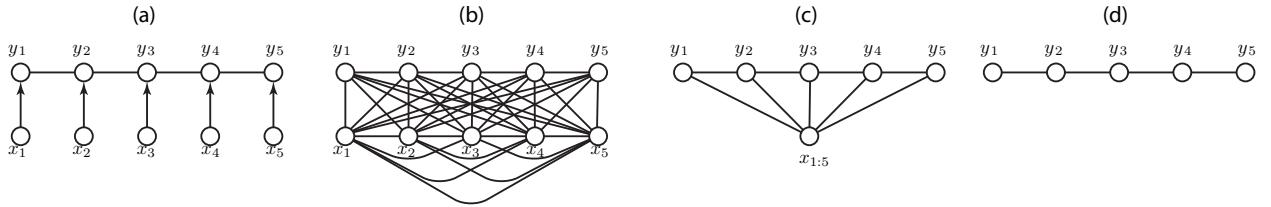


Figure 8.60: Multiple imperfect ways to use a graphical model to describe a CRF. A CRF is a conditional model only, and models $p(y_{1:T}|x_{1:T})$, so no assumptions are being made about $p(x_{1:T})$ in this conditional model. A graphical model, on the other hand, represents a family of joint probability distributions over the random variables corresponding to vertices in the graph. In (a), the CRF is shown as a partially directed and partially undirected model, but there are multiple possible semantics of multi-type edge graphical models, including ancestral graphs, chain graphs, Bayesian networks with soft constraints, and so on, all of which indicate a joint distribution and hence include requirements on $p(x_{1:T})$. In (b) the intention is to say that there are no assumptions made on the distribution $p(x_{1:T})$, but again this indicates a joint rather than a conditional distribution. In (c) we have a similar situation to (b), but where all the observations are bundled into one graph vertex representing $x_{1:T}$. Perhaps (d) is the best, but still imperfect, case since it indicates that a factorization exists over the variables $y_{1:T}$ (indicating that fast dynamic programming can be used), but here still it shows a joint distribution over only $y_{1:T}$ rather than the conditional distribution.

$x_{1:T}$ except for nothing. None of the graphs in Figure 8.60 say nothing about $x_{1:T}$, hence they are imperfect at describing the properties of a CRF.

8.8.5 Generative vs. Discriminative models for classification

When performing pattern classification, we have the option of using either a generative model or a discriminative model. A discriminative model is inherently discriminative in that its only goal is to be able to help make a decision about a class. An example of a discriminative model is one that is a conditional probability of the form $p(y|x)$ or a discriminant function $g(y, x)$. A generative model is one that involves a joint distribution over both variables $p(x, y)$ that one may “generate” samples of x by factoring the model as $p(x|y)p(y)$. A generative model may also be used for discrimination as, fixing observed variables \bar{x} , we can find the y that maximizes $p(\bar{x}, y)$. A generative model may be trained either generatively (that tries to get the model to best explain the data possibly at the expense of discriminative accuracy), or discriminatively (that tries to get the model to make decisions with the lowest error possibly at the expense of generative accuracy). There are cases where these two goals are the same, for example, in the case where we have unlimited training data and where the model family we are training with matches the true data generative process — such cases are not realistic for practical problems where we do not know the true state of nature.

The key issue is we wish to ask in this section is: does a discriminatively trained generative model have any inherent mathematical disadvantages to a conditional model? In other words, is there some inherent mathematical penalty in using a generative model to discriminate compared to using a discriminative model?

A discriminatively trained generative model is not one that necessarily would generate well. But nor would we care if it generates well if our goal is classification. The question is, what do we mean by “generates well”. Let’s take a given true generative model $p(x|q)$, i.e., the distribution for x for a given class q . A good generative model would represent all the nuance for a given class. That is, suppose that there are two objects x_1 and x_2 that are unquestionably of type q . Suppose also that x_1 is a much more frequent example of an object of type q than is x_2 , which means that $p(x_1|q) \gg p(x_2|q)$. If $p'(x|q)$ is an approximate generative model (say estimated from data), we would hope at least for these relationships to be respected, i.e., $p'(x_1|q) \gg p'(x_2|q)$. For a generative model to generate well, we would want any within- q likelihoods to be properly represented.

On the other hand, if we care only that the generative model discriminates well, we do not need the likelihoods of these intra-class objects to be well represented as long as the rank of the likelihoods of the inter-class objects are well represented. For example, supposing that x_3 is an instance of an object of type $r \neq q$, we are happy with our model p' as long as $p'(x_1|q) > p'(x_1|r)$ and $p'(x_2|q) > p'(x_2|r)$ even if $p'(x_1|q) \ll p'(x_2|q)$. I.e., we can completely ignore the within- q likelihoods as long as they are higher for those true q objects than they are for an alternative class, say r .

The question, then, becomes: is there some inherent disadvantage to using a generative model to discriminate, even if it is structured to be discriminative and also trained to be discriminative? The question can be re-phrased in the following way: do we pay to have a normalization over the observations (so that $\int f(x, q) dx = c_x$ where c_x is a constant) more than we pay to have a normalization over the classes (so that $\int f(x, q) dq = c_q$ where c_q is a constant)?

Stated yet another way, lets say that we have any arbitrary non-negative function $f(x, q)$ over both the features x and the class q . The ability of this function to generate well is identical to how well the true distribution $p(x, q)$ is modeled by a joint distribution $p'(x, q)$ constructed as follows:

$$p'(x, q) = p'(x|q)p'(q) = \left(\frac{f(x, q)}{\sum_x f(x, q)} \right) \left(\frac{\sum_x f(x, q)}{\sum_{x,q} f(x, q)} \right) \quad (8.413)$$

The ability of this function to classify well is identical to how well the following conditional distribution classifies:

$$p'(q|x) = \frac{f(x, q)}{\sum_q f(x, q)} \quad \text{and} \quad q^* = \operatorname{argmax}_q p'(q|x) \quad (8.414)$$

where q^* is the classification decision. To classify well, we do not even need $p'(q|x)$ to be close to the true conditional distribution $p(q|x)$ rather we just need $\operatorname{argmax}_q p'(q|x) = \operatorname{argmax}_q p(q|x)$ for all x .

In other words, for a given fixed $f(x, q)$, the corresponding generative model and the corresponding discriminative model have exactly the same classification capabilities (in particular, we could use either the $p(x, q)$ or $p(q|x)$ above and we'd get the same classification error rate since we started with the same $f(x, q)$).

To get into the real power of the model, we need to look at the insides of $f(x, q)$ which includes its factorization properties (one of the most important insides) and the nature of each of these factors.

Recall, an HMM for $p(x, q)$ is specified as:

$$p(x, q) = \prod_t p(x_t|q_t)p(q_t|q_{t-1}) \quad (8.415)$$

with arbitrary locally normalized factors $p(x_t|q_t)$ and $p(q_t|q_{t-1})$ which are shared for all t , and a (linear chain) CRF is:

$$p(q|x) = \frac{1}{Z(x)} \prod_t h(q_t, q_{t-1})g(x_t, q_t) \quad (8.416)$$

for arbitrary non-negative functions $g(\cdot, \cdot)$ and $h(\cdot, \cdot)$.

For an HMM, the corresponding $f(x, q)$ has exactly the same factorization properties than does a CRF. I.e.,

$$f(x, q) = \prod_t f_1(x_t, q_t)f_2(q_t, q_{t-1}) \quad (8.417)$$

I.e., there exists a set of parameter-tied functions f_1 and f_2 such that $f(x, q)$ can be written as a product of these factors. This is what is meant by an HMM and a CRF having the same factorization. If, by contrast, one model used factors involving more than two variables (e.g., one of the models was 2nd order), the 2nd order would be more powerful than the 1st order model, assuming that the same set of random variables

are used variables (i.e., clustering together a number of successive q 's into a single vector variable would essentially be faking a higher-order model). This means that if one of either an HMM or a CRF had a factorization that simplified to:

$$f(x, q) = \prod_t f_1(x_t, q_t) f_3(q_t, q_{t-1}, q_{t-2}) \quad (8.418)$$

then this would have inherently more capability than the one that allowed only two time steps of q variables (a model like the above might be useful to represent tri-gram language models).

An even simpler way of saying this is that a function $f()$ of three variables that must be formed by $f(a, b, c) = f_1(a, b)f_2(b, c)$ is inherently less capable than a function $f(a, b, c)$ that is not restricted by such a factorization property. For example, $f(a, b, c) = abc + ab + bc + ac$ cannot be represented by $f_1(a, b)f_2(b, c)$ for any $f_1(\cdot)$ and $f_2(\cdot)$, nor can it be represented by $f_1(a, b)f_2(b, c)f_3(a, c)$ for any $f_1(\cdot)$, $f_2(\cdot)$, and $f_3(\cdot)$.

Next, we make this argument somewhat more formal by showing that it is possible to transform between an arbitrary HMM and a CRF and back without there being any loss and without there being any increase in computation.

8.8.6 HMMs are identical to CRFs

Now that the title of this section has gotten your attention, what we mean to say is that the class of conditional distributions formed from an HMM is identical to the class of conditional distributions of a CRF, assuming the same set (and type) of random variables (i.e., the $2T$ variables $X_{1:T}$ and $Q_{1:T}$).

From the previous section, we would at least intuitively surmise that the space of conditional distributions that may be represented by a CRF is identical to the space of conditional distributions represented by an HMM. This means that if we conditionally (discriminatively) train an HMM, or if we conditionally train a CRF, then with rich enough representation for each of the individual factors, there is opportunity in each case to reach the same point. To decide between an HMM and a CRF, then, it may be better to consider the discussion in Section 8.8.3.

First, when we use an HMM as a conditional distribution, what this means is that we start with the standard HMM factors (in the BN case, this means start with $p(q_1)$, $p(x_t|q_t)$ and $p(q_t|q_{t-1})$) and then form the *conditional HMM distribution* defined as follows:

$$p_{\text{hmm}}(q|x) = \frac{\prod_t p(x_t|q_t)p(q_t|q_{t-1})}{\sum_q \prod_t p(x_t|q_t)p(q_t|q_{t-1})} \quad (8.419)$$

When we discriminatively train an HMM using conditional maximum likelihood (which is the same as MMI/MMIE training [10, 66, 140, 139, 141, 252, 445]), then what we are doing is setting the HMM parameters so that this conditional HMM distribution $p(q|x)$ is maximized on some data. There are other discriminative training approaches can be applied to HMM training as well, such as max-margin approaches.

We show that the set of conditional HMM distributions are the same as the set of CRFs in the same way as we showed that a MRF and BN HMM are identical. I.e., we start with a conditional HMM, and show the CRF can represent it perfectly (the easy case). Next, we show that starting from a CRF, we can form a conditional HMM that perfectly represents it (the harder case). We will conclude that, since there is a one-to-one correspondence between HMM instances and CRF instances, the key difference between them is how the parameters are derived: with the HMM the parameters can be derived via either a generative or a discriminative training objective, while the CRF's parameters are typically derived using only a discriminative training objective.

Theorem 133. Let $\mathcal{F}_1 = \left\{ p : p(q|x) = \frac{1}{Z(x)} \prod_t h(q_t, q_{t-1}) g(x_t, q_t) \right\}$ be the family of conditional CRF distributions, where g and h are arbitrary non-negative functions over the reals, and $\mathcal{F}_2 = \left\{ p : p(q|x) = \frac{\prod_t p(x_t|q_t) p(q_t|q_{t-1})}{\sum_q \prod_t p(x_t|q_t) p(q_t|q_{t-1})} \right\}$ be the family of conditional HMM distributions. Then $\mathcal{F}_1 = \mathcal{F}_2$.

Proof. We'll do the easy case first. We start with an HMM and consider its conditional distribution:

$$p(q|x) = \frac{\prod_t p(x_t|q_t) p(q_t|q_{t-1})}{\sum_q \prod_t p(x_t|q_t) p(q_t|q_{t-1})} \quad (8.420)$$

Consider now a CRF.

$$p(q|x) = \frac{1}{Z(x)} \prod_t h(q_t, q_{t-1}) g(x_t, q_t) \quad (8.421)$$

since there are no restrictions on h and g other than non-negativity, we can just set $h(q_t, q_{t-1}) \leftarrow p(q_t|q_{t-1})$ and $g(x_t, q_t) \leftarrow p(x_t|q_t)$ from the HMM. Clearly, then, the CRF's $Z(x)$ would be that $Z(x) = \sum_q \prod_t p(x_t|q_t) p(q_t|q_{t-1})$ and we see that the CRF perfectly represents the conditional HMM distribution.

Next, we need to show that given an arbitrary CRF, an HMM can be formed that perfectly represents the CRF when the HMM is used to form a conditional HMM distribution. This actually isn't so hard since we did most of the work in Section 8.8.2.

We start with an arbitrary CRF which is given by the factors $h(q_t, q_{t-1})$ and $g(x_t, q_t)$ (note that we always must have these factors for the CRF since it is with these factors that make it possible to compute $Z(x)$ efficiently). Define arbitrary fully factorizable positive independent measure over $x_{1:T}$ (an i.i.d. distribution suffices)

$$p(x_{1:t}) = \prod_{t=1}^t p_t(x_t) \quad (8.422)$$

where $p_t(x_t) > 0$ for all t and all x_t . Form the normalized relationship;

$$p(q|x)p(x) = \frac{1}{Z} \prod_t h(q_t, q_{t-1}) \underbrace{g(x_t, q_t)p_t(x_t)}_{g'(x_t, q_t)} \quad (8.423)$$

where $Z = 1$ and $g'(x_t, q_t) = g(x_t, q_t)p_t(x_t)$. On the right hand side of Equation (8.423), we have formed a joint distribution from the CRF factors h and g - we stress that this is a joint distribution $p_{\text{JCRF}}(q, x)$ over q and x , not one that is necessarily good or representative of anything physical, rather it is a validly mathematically normalized joint distribution in the sense that $\int p_{\text{JCRF}}(q, x) d\mu(q) d\mu(x) = 1$ for the appropriate measures. Moreover, it is a joint distribution that factorizes with respect to the Markov random field HMM graph Figure 8.58-B. But once we have such a distribution, we know from Section 8.8.2 that it is possible to set the HMM factors $p(x_t|q_t)$ and $p(q_t|q_{t-1})$ so that the Bayesian network HMM is exactly the same as this joint distribution. I.e., we can set $p(x_t|q_t)$ and $p(q_t|q_{t-1})$ so that the following is true:

$$p(q|x)p(x) = \frac{1}{Z} \prod_t h(q_t, q_{t-1}) g'(x_t, q_t) = \prod_t p(x_t|q_t) p(q_t|q_{t-1}) \quad (8.424)$$

Now if we take this resulting HMM and form the conditional HMM distribution (Equation (8.419)), what do

we get back? Starting from the conditional HMM distribution and using these factor assignments, we get:

$$\frac{\prod_t p(x_t|q_t)p(q_t|q_{t-1})}{\sum_q \prod_t p(x_t|q_t)p(q_t|q_{t-1})} = \frac{p(q|x)p(x)}{\sum_q p(q|x)p(x)} \quad (8.425)$$

$$= \frac{p(q|x)}{\left(\sum_q p(q|x)\right)} \quad (8.426)$$

$$= p(q|x) \quad (8.427)$$

In other words, the conditional HMM distribution formed from the HMM we created is exactly the same as the original CRF. \square

Therefore, HMMs and CRFs are identical in the sense that they offer the same opportunity to produce a given classifier.

If we form a conditional HMM distribution from an HMM that is discriminatively trained, we can identically classify/decode using either the conditional HMM form $p(q|x)$ or the joint form $p(q, x)$ using, say, Viterbi decoding, since the normalization factor doesn't change the classification boundaries (only the training does). Therefore, there is no mathematical reason to prefer CRFs over HMMs.

8.8.7 Observations dependent on pairs of state variables

In this section, we discuss the issue of Mealy vs. Moore Machines and the HMM/CRF question.

Equation (8.412) shows a CRF that uses factors involving two rather than a single state variable. More generally, a linear-chain CRF can be defined as a conditional distribution over $2T$ random variables $X_{1:T}$ and $Q_{1:T}$ that factorizes as follows:

$$p(q|x) = p(q_{1:T}|x_{1:T}) = \frac{1}{Z(x)} g(x_1, q_1) \prod_{t=2}^T h(q_t, q_{t-1})g(x_t, q_t, q_{t-1}) = \frac{1}{Z(x)} \prod_t h(q_t, q_{t-1})g(x_t, q_t, q_{t-1}) \quad (8.428)$$

where

$$Z(x) = \sum_{q_{1:T}} \prod_t h(q_t, q_{t-1})g(x_t, q_t, q_{t-1}) \quad (8.429)$$

where $h(\cdot, \cdot)$ and $g(\cdot, \cdot, \cdot)$ are arbitrary non-negative functions of their respective arguments (like in the MRF HMM above, we assume for notational convenience that $h(q_1, q_0) = 1$ and $g(x_1, q_1, q_0) = g(x_1, q_1)$).

This means that the observation can interact directly with two successive rather than only one state variable. HMMs, however, can also be written so that the observation is dependent on the state transition rather than on just an individual state. In fact, SFSA descriptions of HMMs [307, 308] often use this form of HMM rather than the single state conditioned description. Two of the early HMM researchers, proponents, expositors, and HMM practitioners in the field of speech recognition, moreover, have adopted these different (but mathematically equivalent) HMM descriptions, namely Fred Jelinek [205] who preferred to say that the observation depended on the transition, and Lawrence Rabiner who preferred to describe the observation depending only on the state [348]. We discussed this distinction at length and how they were equivalent in §8.4.4.6 and again in in §8.4.10.

An HMM where the observation is directly dependent on both the current and the next state, and hence the state transition, is given in Figure 8.61. This can be contrasted with Figure 8.45, where the observation at time t is directly dependent on the current and previous state, a model that is obviously equivalent. The

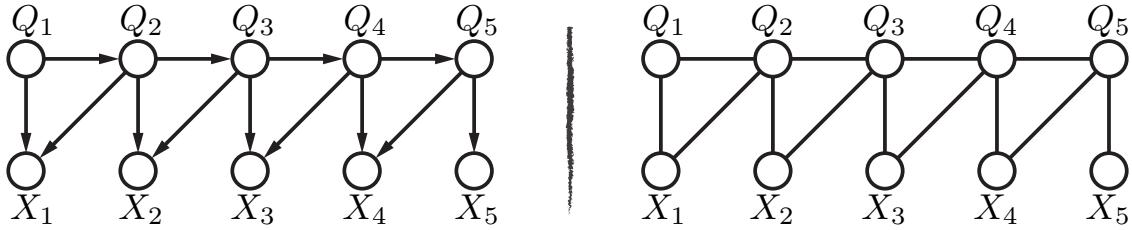


Figure 8.61: A: A Bayesian network view of an HMM where the observation X_t is dependent on the state transition between states Q_t and Q_{t+1} . We note that the computational complexity if this model is identical to the one shown in Figure 8.58-A. B: An undirected graphical model view of the same model.

BN version of Figure 8.61 corresponds to the equation:

$$p(x_{1:T}, q_{1:T}) = p(q_1)p(x_T|q_T) \prod_{t=1}^{T-1} p(x_t|q_t, q_{t+1})p(q_{t+1}|q_t) = \prod_t p(x_t|q_t, q_{t+1})p(q_{t+1}|q_t) \quad (8.430)$$

For the same reasons given in Section 8.8.6, such an HMM is identical to the CRF in Equation (8.412). More discussion on the difference between Moore and Mealy machines is given in §8.4.4.6 and §8.4.10.

8.8.8 The observations and temporal integration

CRFs have an added property which is that, since they are only representing a conditional distribution, the factors may involve observations from the entire sequence rather than from only the current time. That is, contrasted with Equation (8.428), we may define a CRF as:

$$p(q_{1:T}|x_{1:T}) = \frac{1}{Z(x)} \prod_t h(q_t, q_{t-1})g(x_{1:T}, q_t) \quad (8.431)$$

where

$$Z(x) = \sum_{q_{1:T}} \prod_t h(q_t, q_{t-1})g(x_{1:T}, q_t) \quad (8.432)$$

Note that $g(x_{1:T}, q_t)$ may be a function of any of the observed variables. Since the random variables $X_{1:T}$ are always observed, they are constant in the above equations. Therefore, using global factors involving variables from all time such as $g(x_{1:T}, q_t)$ does not add to the computational complexity of computing $Z(x)$ or any of the standard forward-backward quantities. In practice, we never use an HMM with all observations over all t . Rather, the factor will take the form $g(x_{t-M:t+M}, q_t)$ meaning that the time t hidden variable Q_t is involved in a factor that uses observations within a $2M + 1$ -wide window centered at t .

It might seem like an HMM, being a generative model, does not have the luxury to use such large spans of observations in its factors. However, as mentioned above, when an HMM is meant to be used as a conditional distribution (such as when it is trained discriminatively), it need not be a good generative model, and in fact can even be a poor generative model. For example, consider the HMM formed as follows:

$$p(x, q) = p(x_{1:T}, q_{1:T}) = p(x_1|q_1)p(q_1) \prod_{t=2}^T p(x_t|q_t)p(q_t|q_{t-1}) = \prod_t p(x_t|q_t)p(q_t|q_{t-1}) \quad (8.433)$$

Suppose that there is an underlying observation sequence (z_1, z_2, \dots, z_T) . We define a sequence function of the original sequence as follows: $x_t = (z_{t-M}, z_{t-M+1}, \dots, z_t, z_{t+1}, \dots, z_{t+M})^\top$ for each t . This means that a window the z feature stream of width $2M + 1$ is used for each frame in an HMM. For convenience, let

us define $x_t^i \triangleq z_{t+i}$. Hence, in any particular sample of the observed variables, we would have $x_t^i = x_{t+1}^{i-1}$, and more generally $x_t^i = x_{t-k}^{i+k}$. We have the relation:

$$p(x_1, x_2, \dots, x_T) = p\left(\begin{bmatrix} z_{1-M} \\ z_{2-M} \\ \vdots \\ z_{1+M} \end{bmatrix}, \begin{bmatrix} z_{2-M} \\ z_{3-M} \\ \vdots \\ z_{2+M} \end{bmatrix}, \dots, \begin{bmatrix} z_{T-M} \\ z_{T-M+1} \\ \vdots \\ z_{T+M} \end{bmatrix}\right) \quad (8.434)$$

Now, modeling this sequence of vectors as an HMM, in which each time step would correspond to a distinct random variable, would look something more like:

$$p(X_1 = x_1, X_2 = x_2, \dots, X_T = x_T) \quad (8.435)$$

$$= p\left(\begin{bmatrix} Z_{1-M} \\ Z_{2-M} \\ \vdots \\ Z_{1+M} \end{bmatrix} = \begin{bmatrix} z_{1-M} \\ z_{2-M} \\ \vdots \\ z_{1+M} \end{bmatrix}, \begin{bmatrix} Z_{2-M} \\ Z_{3-M} \\ \vdots \\ Z_{2+M} \end{bmatrix} = \begin{bmatrix} z_{2-M} \\ z_{3-M} \\ \vdots \\ z_{2+M} \end{bmatrix}, \dots, \begin{bmatrix} Z_{T-M} \\ Z_{T-M+1} \\ \vdots \\ Z_{T+M} \end{bmatrix} = \begin{bmatrix} z_{T-M} \\ z_{T-M+1} \\ \vdots \\ z_{T+M} \end{bmatrix}\right) \quad (8.436)$$

The true generative distribution, if we were able to obtain it, would be such that any impossible event would be given zero probability. For example, under the true distribution, if there was an event where $x_t^i \neq x_{t-k}^{i+k}$, then this would be given zero probability.

If an HMM was given a training set where, for all training samples, $x_t^i = x_{t-k}^{i+k}$, since the HMM is not the true generative distribution, it might (and in fact would) not give zero probability to the event $x_t^i \neq x_{t-k}^{i+k}$. In fact, depending on the training method, it could give this event a reasonably high probability. If, moreover, a discriminative training method was used for the HMM, the fact that it does not give an event that did not occur in the training set a zero probability is irrelevant to the objective of training which is, in such case, ensuring that the conditional HMM distribution is accurate when used for classification.

To put this into perspective, we said above that a CRF can have a factor $g(x_{t-M:t+M}, q_t)$ for time t that involves observations for more than time t . What the discussion here means is that there is nothing stopping us from using a “factor” in an HMM of the form $p(x_{t-M:t+M}|q_t)$ as long as we do not expect it to accurately generate samples in this form. That is, if we were to discriminatively train the HMM, then we are not concerned with the generative accuracy of the HMM in the first place. If we generatively train an HMM and use factors such as $p(x_{t-M:t+M}|q_t)$, then we should not expect the HMM to generate well but it might (and in fact often does) have other uses (cf. §8.8.9).

It may be aesthetically and mathematically unsettling to use a generative distribution for tasks other than generation of samples. It may be even more unsettling use it when we know that, if it were ever used for a generative task, it would give non-zero probability to impossible events. Why moreover would we want to use generatively trained HMMs that involve factors such as $p(x_{t-M:t+M}|q_t)$? Why not use a CRF directly? There are of course many reasons for using CRFs, since we are using a model formed and optimized for a particular task, namely discrimination of objects as represented by a set of numeric features $x_{1:T}$. There are a number of reasons, however, for using generative models as mentioned in §, even when one is performing classification. Secondly, as we see in §8.8.9, they can still work quite well in practice and can be easier to train than discriminative models. Third, when discriminatively training a model, it is often good to initialize the parameters to their generatively trained values. More discussion on these points is given in §.

8.8.9 HMM delta features, and why they help speech recognition performance

In this section, we give an example of a case where the issue described in §8.8.8, namely using more one frame of observations for x_t in each a locally normalized HMM factor $p(x_t|q_t)$, is extremely useful in

practice.

State-of-the-art speech recognition systems augment HMM feature vectors X_t with approximations to their first and second order time-derivatives (called delta- and delta-delta- features [137, 161, 162, 163], or just “dynamic” features). Most often, estimates of the derivative are obtained using linear regression [347], namely:

$$\dot{x}_t = \frac{\sum_{k=-K}^K kx_t}{\sum_{k=-K}^K k^2}$$

where K in this case is the number of points used to fit the regression. This can be viewed as a regression because

$$\dot{x}_t = \sum_{k=-K}^K a_k x_{t-k} + \epsilon$$

where a_k are defined accordingly, and ϵ can be seen as a Gaussian error term. A new feature vector is then produced that consists of x_t and \dot{x}_t appended together.

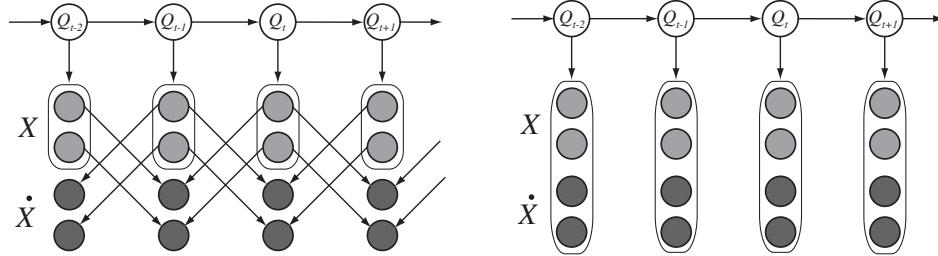


Figure 8.62: A GM-based explanation of why delta features work in HMM-based ASR systems. The left figure gives a GM that shows the generative process of HMMs with delta features. The right figure shows how delta features are typically used in an HMM system, where the information between \dot{X}_t and Q_t is greatly increased relative to the left figure.

It is elucidating to expand the joint distribution of the features and the deltas, namely $p(x_{1:T}, \dot{x}_{1:T}) = \sum_{q_{1:T}} p(x_{1:T}, \dot{x}_{1:T}|q_{1:T})p(q_{1:T})$. The state conditioned joint distribution within the sum can be expanded as:

$$p(x_{1:T}, \dot{x}_{1:T}|q_{1:T}) = p(\dot{x}_{1:T}|x_{1:T}, q_{1:T})p(x_{1:T}|q_{1:T}).$$

The conditional distribution $p(x_{1:T}|q_{1:T})$ can be expanded as is normal for an HMM [347, 30], but

$$p(\dot{x}_{1:T}|x_{1:T}, q_{1:T}) = \prod_t p(\dot{x}_t|\text{parents}(\dot{x}_t)).$$

This last equation follows because, observing the process to generate delta features, \dot{X}_t is independent of everything else given its parents. The parents of \dot{X}_t are a subset of $X_{1:T}$ and they do not include the hidden variables Q_t . This leads to the GM on the left in Figure 8.62, a generative model for HMMs augmented with delta features. Note that the edges between the feature stream X_t , and the delta feature stream \dot{X}_t correspond to deterministic linear implementations. In this view, delta-features appear to be similar to fixed-dependency auto-regressive HMMs, where each child feature has additional parents both from the past and from the future. In this figure, however, there are no edges between \dot{X}_t and Q_t , because $\dot{X}_t \perp\!\!\!\perp Q_t | \text{parents}(\dot{X}_t)$. This means that $\text{parents}(\dot{X}_t)$ contain all the information about \dot{X}_t , and Q_t is irrelevant. If there was an edge

between Q_t and \dot{X}_t , and if such a model was learnt based on data, then the associated parameters for this interaction would be learnt as irrelevant (at best) or noise (at worst).

It is often asked why delta features help ASR performance as much as they do. The left of Figure 8.62, the true generative model, does not depict the class of model typically used with delta features. A goal of speech recognition is for the features to contain as much information as possible about the underlying word sequence as represented via the vector $Q_{1:T}$. The true generative model on the left in Figure 8.62 shows, however, that there is zero information between the \dot{X}_t and Q_t . When the edges between \dot{X}_t and its parents (\dot{X}_t) are removed, the mutual information [97] between \dot{X}_t and Q_t can only *increase* (from zero to something greater) relative to the true generative model. The right of Figure 8.62 shows the standard model used with deltas, where it is not the case that $\dot{X}_t \perp\!\!\!\perp Q_t$. Since in the right model, it is the case that more information about \dot{X}_t and Q_t exist, it might be said that this model has a structure that is inherently more discriminative.

Interestingly, the above analysis demonstrates that additional conditional independence assumptions (i.e., fewer edges) in a model can increase the amount of mutual information that exists between random variables. When edges are added between the delta features and the generative parents X_t , the delta features become less useful since there is less (or zero) mutual information between them and Q_t .

Therefore, the very conditional independence assumptions that are commonly seen as a flaw of the HMM provide a benefit when using delta features. More strongly put, the *incorrect* statistical independence properties made by the HMM model on the right of Figure 8.62 (relative to truth, as shown by the generative model on the left) are the very thing that enable delta features to decrease recognition error. The standard HMM model with delta features seem to be an instance of a model with an inherently discriminative structure [45, 39].

In general, can the removal of edges or additional processing lead to an overall increase in the information between the entire random vectors $X_{1:T}$ and $Q_{1:T}$? The data processing inequality [97] says it can not. In the above, each feature vector (\dot{X}_t, X_t) will have more information about the temporally local hidden variable Q_t — this can sometimes lead to better word error scores. This same analysis can be used to better understand other feature processing strategies derived from multiple frames of speech, such as PCA or LDA preprocessing over multiple windows [181] and other non-linear generalizations [148, 238, 193].

8.8.10 The observations and parametric form

In HMMs, the observation distributions $p(x_t|q_t)$ are often Gaussians or Gaussian mixtures. In CRFs, the observation factor $g(x_t, q_t)$ typically arises from a log-linear model. That is, we have that $g(x_t, q_t) = \lambda^\top a(x_t, q_t)$ where $a(x_t, q_t)$ is a vector of time-homogeneous feature functions, and where λ is a vector of parameters. The feature functions are time-homogeneous in that they don't change as a function of time, nor do the parameters λ — the only thing that changes is the observed vector x_t and the state q_t . We wish to discuss if there is any benefit to one vs. the other.

Firstly, a single multivariate Gaussian can easily be modeled by a log-linear factor. Both are members of the exponential family. The Gaussian has as sufficient statistics the vector (x_1, x_2, \dots, x_N) along with $(x_1 x_1, x_1 x_2, \dots, x_1 x_N, x_2 x_2, x_2 x_3, \dots, x_2 x_N, \dots, x_N x_N)$ so if we define $a(x_t, q_t)$ as a length $N(N + 1)/2 + N$ vector of these features, then any set of parameters for a Gaussian can be encoded with an appropriate value of λ . This in fact is the difference between the usual mean/variance parameterization of a Gaussian and its canonical parameterization as seen via its membership in the exponential family. The log-linear representation is not unrestricted though. Indeed, in order for the factor to integrate to a non-infinite quantity, we must have that $\lambda_\ell < 0$ for all ℓ corresponding to any of the features $(x_i)^2$. Parameters so restricted are called the natural parameters of the exponential model.

On the other hand, a log-linear model can use any set of features it wants, not just ones based on polynomial combinations of the elements of x_t . Indeed, it can use any non-linear function of elements of

x_t as well as q_t to produce a rich set of features, all of which are parameterized by the associated vector λ . Indeed, this flexibility is one of the great benefits of log-linear models — no matter how complex a set of features are used for $a()$, the parameterization is still log-linear. In general, there is not any theoretical limit to the complexity of a distribution of the form

$$g(x_t, q_t) = \exp(\lambda a(x_t, q_t)) \quad (8.437)$$

where $a(\cdot)$ is a function that returns a vector of features over x_t and q_t . I.e., $a()$ can be any set of complicated non-linear aspects of these features.

Hence, researchers and specialists in a given applications domain (such as NLP, speech, or biology) can design by hand features that are appropriate for the domain, and the parameters λ can be learned by the same method. This method has yielded great success, and the features that are designed usually are based on having intimate knowledge of the domain. On the other hand, many people who use CRFs for speech use a mixture of Gaussians as the $a()$ function that has been trained using maximum-likelihood from an HMM (i.e., generative training). This is sort of ironic as to use a CRF it is necessary to use an HMM — at the very least, this approach counts as an admission that obtaining the features $a()$ is difficult in practice, especially when little is known about the domain.

Moreover, getting back to the case where x_t is a continuous vector and q_t is a state, is this inherently better than a factor of the form $p(x_t|q_t)$? If we consider a mixtures of Gaussians, there is no theoretical limit to the complexity of distributions to Gaussian mixtures.

To summarizes, when comparing an HMM and a CRF w.r.t. the factors involving the observations, model might have an advantage over the other if we are in the case where the two models have specific fixed “implementations” of these factors. If, on the other hand, the implementation is such that it uses a family of models that has no inherent representational capacity limit (or if the representational capacity has no bound, as is sometimes the case with Bayesian non-parametric models), then there is no inherent advantage of one other the other.

8.8.11 CRFs, Convexity, and Training

The log probability score of a standard log-linear model is convex in the parameters and a CRF, when it is (as is typical) formed in the following way

$$p(q_{1:T}|x_{1:T}) = \frac{1}{Z(x)} \exp(\lambda a(q_{1:T}, x_{1:T})) \quad (8.438)$$

where a is a vector of feature functions and λ is a vector of parameters, is also log convex. Hence, one might argue that it is inherently easier to train a CRF than an HMM. Given the right features, the convexity of the training process means that there is inherently more opportunity to reach good points in a CRF’s parameter space than with a discriminatively trained HMM which is typically non-convex. Before concluding that the CRF wins this round, one must observe that we still need to find the right features with a CRF. As mentioned earlier, generatively trained Gaussian mixture or Gaussian HMM scores are often used as CRF features, so the entire process of obtaining features and then discriminative optimization in a CRF is not jointly convex. It may be that a non-convex optimization procedure that simultaneously finds the features and also learns the log-linear parameters of an HMM is superior, but this brings us back to the (non-convex) process of discriminatively trained HMMs.

It should also be mentioned that hierarchical CRFs, which involve a sequence of hidden variables like a hierarchical HMM, are no longer convex. It is often the case that HMMs are really flattened hierarchical models. So, when comparing Hierarchical CRFs and HMMs, the potential convexity benefit of CRFS vanishes.

8.8.12 The label-bias problem

It might be asked, if HMMs and CRFs are identical, what does this say about the advantages that CRFs have as related to label-bias (and observation-bias) problem? Indeed, the label-bias problem was one of the key motivations for CRFs [256] — the CRF was offered as a discriminative model that did not suffer from problems that certain other discriminative sequence models had (as we will discuss below). In this and the following sections, we introduce the label-bias and the observation-bias problems, when they arise, when they are a problem, and (surprisingly) a number of cases where it might seem at first like they cause problems with the model but on a second look we see that the problems are actually not so bad at all.

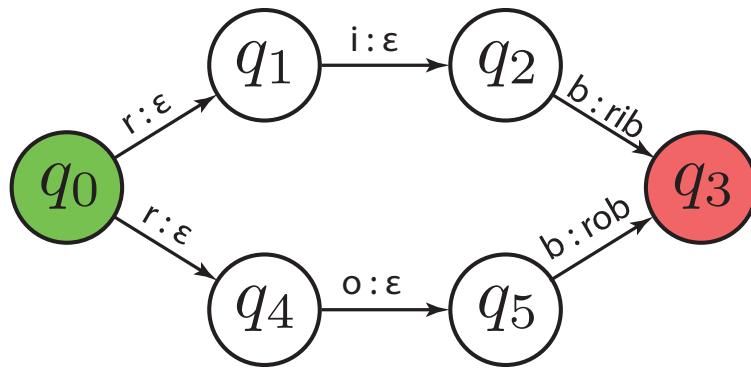


Figure 8.63: An example that was used to demonstrate the label-bias issue under a CMM. Here, the notation $o : \ell$ corresponds to observation-label pairs, and the symbol “ ε ” represents the null output label. Hence, this shows a classic Mealy style FSA but in this case used to describe the behavior of a model that exhibits label-bias, as described in the text.

The label-bias issue was first introduced in the context of an stochastic finite-state automata example, as shown in Figure 8.63. The argument went something like this.

1. The HMM is not a discriminative model.
2. The conditional Markov model (or CMM, defined below and in Figure 8.64) is a discriminative conditional model and hence should be better for classification purposes.
3. The CMM has problems, namely label-bias and observation-bias.
4. Hence, the CMM is not a good model.
5. The CRF does not have label-bias and observation-bias issues, but is still a conditional model, and hence it is perhaps best for sequential discriminative tasks.

8.8.12.1 Conditional Markov Models (CMMs)

Firstly, what is a CMM? A CMM is a class of models that is distinct from both the CRF and the HMMs in that the CMM has both different factorization properties and different normalization requirements. The factorization and normalization requirements of an CMM lead to some conditional independence statements that can cause problems. Technically speaking, the issue is with the model’s v-structures. Recall, a v-structure in a Bayesian network is a variable that has multiple incoming edges. In a Bayesian network, the family of probability distributions is not changed when you change the direction of edges as long as the v-structures in the model are not changed. The reason is that the set of independence statements is not changed if you change the directions of the arrows without changing the set of v-structures. A CMM has a different

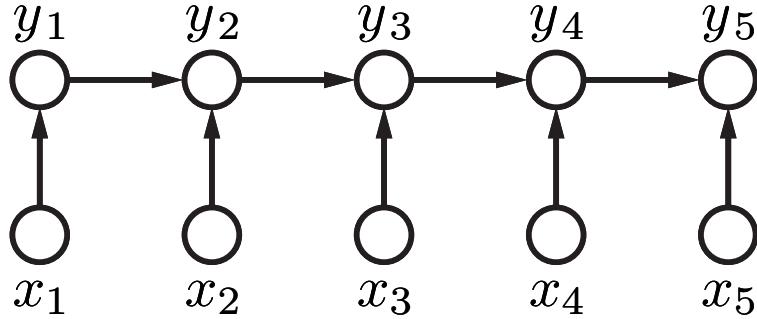


Figure 8.64: An Bayesian network description of the class of conditional Markov models (CMMs), also sometimes called a MEMM (maximum entropy Markov model).

set of v-structures than an HMM. In fact, an HMM has no v-structures at all, which can be seen either by noting that all variables in a BN description of an HMM have only one incoming edge (equivalently, we see that the class of HMM models can be described either via a BN or a Markov random field as shown in § 8.8.2, and we note that the Markov random field has no v-structures). Also, a CRF is not a directed model so it also has no v-structures.

A CMM family can easily be described using a Bayesian network, as shown in Figure 8.64. Note that the CMM has sometimes been called the “maximum entropy Markov models” (or MEMMs [297]) which is a misnomer since within the definition of these models there is nothing inherently “maximum entropy” about them. In [297], one of the conditional factors $p(q_t|q_{t-1}, x_t)$ was implemented with a local conditional maximum entropy model which is the reason for the name, but there are other ways to implement the factor $p(q_t|q_{t-1}, x_t)$ besides a local maximum entropy model. Hence, we prefer to use the term CMM. In the sequel, we will define the label- and observation-bias problems, we will show that these can exist in the CMM, but that they exist in neither HMMs nor CRFs.

Lets now go over CMMs as shown in Figure 8.64 in a bit more detail. An CMM is the class of all joint distribution over q and x that may be factorized as follows:

$$p(q, x) = p(q_1|x_1)p(x_1) \prod_t p(q_t|q_{t-1}, x_t)p(x_t). \quad (8.439)$$

This factorization constraint should be compared with Figure 8.64. Note, this is family of *joint* probability distributions, unlike CRFS which are a class of conditional distributions. The factorization assumptions here, however, are quite different than that of an HMM which is also a joint distribution family. In particular, we see now that we’ve got a v-structure at Q_t for every $t > 2$ (i.e., Q_t has two parents, Q_{t-1} and X_t). Due to these v-structures, this model makes the assumptions that all the observations are independent, that is $X_A \perp\!\!\!\perp X_B$ for all index sets $A \cap B = \emptyset$. Since the CMM is typically used for prediction, where we fix $\bar{x}_{1:T}$ and predict using $\text{argmax}_{q_{1:T}} p(q_{1:T}, \bar{x}_{1:T}) = \text{argmax}_{q_{1:T}} p(q_{1:T}|\bar{x}_{1:T})$, the actual distribution $p(\bar{x}_{1:T})$ does not change the result. Hence we assume the simplest distribution on $p(\bar{x}_{1:T})$ which is the fully factorized one $p(\bar{x}_{1:T}) = \prod_t p(\bar{x}_t)$.

A generalization of CMMs can be made by using factors of the form $p(q_t|q_{t-1}, q_{t-2}, \dots, q_{t-k}, x_{t-M:t+M})$ rather than $p(q_t|q_{t-1}, x_t)$. This has been called a *recurrent sliding window* model in the literature [124]. The model is shown in Figure 8.65. Note that officially, the way a recurrent sliding window model was proposed was to first make predictions for $Q_{t-1}, Q_{t-2}, \dots, Q_{t-k}$, clamped to values $\bar{q}_{t-1}, \bar{q}_{t-2}, \dots, \bar{q}_{t-k}$, and then make the prediction for time t using: $p(q_t|\bar{q}_{t-1}, \bar{q}_{t-2}, \dots, \bar{q}_{t-k}, \bar{x}_{t-M:t+M})$. This would not be the standard forward algorithm for these models since the forward procedure would be the equivalent of jointly considering all hypothesizes over the

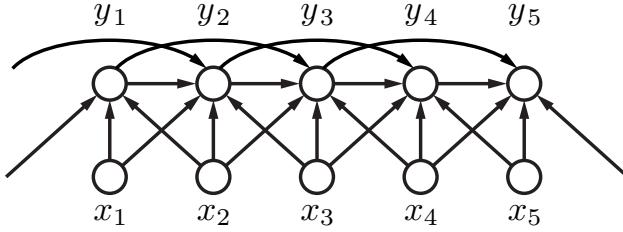


Figure 8.65: A Bayesian network description of the factorization properties underlying a recurrent sliding window model, with $k = 2$ and $M = 1$.

entire hidden sequence. The successive prediction and clamping of the recurrent sliding window procedure, however, has significantly less computational cost.

The label bias and observation bias problems as described below occur for the recurrent sliding window model as well, but it is simplest to discuss the case when $k = 1$ and $M = 0$ (i.e., a CMM) as we do next.

8.8.12.2 Label Bias and CMMs

Lets begin by describing label-bias in a CMM as it was originally described using Figure 8.63. The goal of the model is to classify the observation sequence to be one of two words “rib” or “rob” which differ in only one character in the middle. We have a set of observations that have the ability to strongly represent any of the letters of each of the two words, so much so that lets also use the set of letters $X_t \in \{r', i', o', b'\}$ as our observation alphabet set.

When a CMM is formed as described in Figure 8.63, it should have non-zero probability for advancing either to state q_1 or q_4 when we see the observation ‘r’. That is, we should have both that $p(q_1|q_0, r') > 0$ and also $p(q_4|q_0, r') > 0$. This is shown in Figure 8.63 via the arcs between these states. Next, once we are in state q_1 we deterministically advance to state q_2 , and this means that $p(q_2|q_1, *) = 1$ meaning that the probability is 1 regardless of what the observation is at that point. Similarly, the model states that $p(q_5|q_4, *) = 1$, again giving the same probability oblivious to the current observation. Lastly, the model states that both $p(q_3|q_2, b') = p(q_3|q_5, b') = 1$. So, the model then hypothesizes either “rib” or “rob” for output as desired, depending on if the transition $q_2 \rightarrow q_3$ (“rib”) or the transition $q_5 \rightarrow q_3$ (“rob”) is taken. The choice of which of these two transitions is taken depends only on which of the initial two transitions $q_0 \rightarrow q_1$ and $q_0 \rightarrow q_4$ is taken, these having probability $p(q_1|q_0, r')$ vs. $p(q_4|q_0, r')$ respectively. The choice of final transition, however, does not depend on the key distinguishing observation (“i” or “o”). That is, the model outputs one of the two hypotheses irrespective of the key bit of information needed to make the correct choice. The model has a strong “bias” in favor of the labels at the expense of the observations, hence it is called “label bias.”

Why does this happen? The culprit boils down to the CMMs factorization properties (in particular, its v-structures) which might seem innocuous at first. They actually means, however, that there are some conditional independence statements that can be extremely undesirable for a sequence model and that can have some real detrimental consequences in practical applications. In particular, due to the v-structures, this model says that X_t is independent of Q_{t-1} . This means that the current observation may not have influence on the previous state variable. Even worse, this model says that, for all $t = 2 \dots T$, the set of random variables $\{X_t, X_{t+1}, \dots, X_T\}$ is independent of the set of variables $\{Q_1, X_1, Q_2, X_2, \dots, Q_{t-1}, X_{t-1}\}$. This property of “future observations being unable to influence anything in the past”, (or worded perhaps more temporally, “past variables unable to predict any future observations”) is the root cause of what is referred to as label-bias and observation-bias.

It is important to note that neither the HMM nor the CRF makes these independence assumptions. More

on this below, but this is actually worth considering in the context of early CRF papers [256], where CRFs are compared with the CMM but discriminatively trained HMMs (as is common for speech recognition) were not considered. The HMM never had the label-bias problem to begin with and more specifically, neither did the conditional HMM distribution of Equation (8.419).

The first thing to note is that, perhaps surprisingly, the CMM independence assumptions do not necessarily have an adverse effect during decoding. For example, consider a three time-frame CMM. While it is the case that $X_3 \perp\!\!\!\perp \{Q_1, Q_2\} | \{X_1, X_2\}$ giving $p(q_1, q_2 | x_1, x_2, x_3) = p(q_1, q_2 | x_1, x_2)$ (so x_3 would never effect the coding of q_1, q_2 in this form), it is not the case that $X_3 \perp\!\!\!\perp \{Q_1, Q_2\} | \{X_1, X_2, Q_3\}$ – knowing Q_3 allows X_3 to have an influence on earlier states.

During decoding, we perform the following computation in a CMM:

$$\operatorname{argmax}_{q_1, q_2, q_3} p(q_1, q_2, q_3, \bar{x}_1, \bar{x}_2, \bar{x}_3) = \operatorname{argmax}_{q_1, q_2, q_3} p(q_1, q_2, q_3 | \bar{x}_1, \bar{x}_2, \bar{x}_3). \quad (8.440)$$

The question is, which of the above independence statements apply during decoding? We can view this in several ways. First, consider the backwards inference (we note that forward and backward inference produce identical results):

$$\max_{q_1, q_2, q_3} p(q_1, q_2, q_3 | \bar{x}_1, \bar{x}_2, \bar{x}_3) = \max_{q_1, q_2, q_3} p(q_1 | \bar{x}_1) p(q_2 | q_1, \bar{x}_2) p(q_3 | q_2, \bar{x}_3) \quad (8.441)$$

$$= \max_{q_1} p(q_1 | \bar{x}_1) \left(\max_{q_2} p(q_2 | q_1, \bar{x}_2) \left(\max_{q_3} p(q_3 | q_2, \bar{x}_3) \right) \right) \quad (8.442)$$

$$= \max_{q_1} p(q_1 | \bar{x}_1) \left(\max_{q_2} p(q_2 | q_1, \bar{x}_2) \phi_{Q_3}(q_2, \bar{x}_3) \right) \quad (8.443)$$

$$= \max_{q_1} p(q_1 | \bar{x}_1) \phi_{Q_2}(q_1, \bar{x}_2, \bar{x}_3) \quad (8.444)$$

Looking at this last maximization over q_1 , we see that the maximization over q_1 involves a function that in fact *does* utilize the observation x_3 . This seems counter intuitive, since the aforementioned conditional independence assumptions state that current states are not influenced by future observations. How can joint-decoding render dependent variables that were previously independent? To see how this can occur, note that we have not marginalized out (via summation) Q_3 as in $p(q_1, q_2 | x_1, x_2, x_3) = \sum_{q_3} p(q_1, q_2, q_3 | x_1, x_2, x_3)$, but rather we are finding the maximum assignment to all of Q_1, Q_2, Q_3 jointly and simultaneously. This decoding process effectively “conditions” (in a fashion) on the variables that we are maximizing over, inducing a dependence between X_3 and earlier in the network.

How does this happen? In a normal V-structure in a Bayesian network, say where $X \rightarrow Z \leftarrow Y$, there is an independence property stating that $X \perp\!\!\!\perp Y | Z$. If Z becomes observed, then X and Y are no longer independent, i.e., it is not the case that $X \perp\!\!\!\perp Y | Z$. Let X be observed to value \bar{x} . Note we are not setting the value Z to be fixed, rather setting X to be fixed and maximizing over Y and Z . That is, we are interested in computing $\max_{y,z} p(z | \bar{x}, y) p(\bar{x}) p(y)$. The question is, does X in this case influence the decision on Y ? Independence would mean that the score of values of Y does not change for any value of X (which would be the case if Z was integrated away), but we see above that this is not the case. I.e., the factor $p(z | \bar{x}, y)$ still exists in the maximization and can provide different values for pairs of z and y . For example, there might be certain values of z that give zero probability for certain x, y pairs, and this factor would then preclude any of these pairs from occurring with the corresponding z . Algorithmically, we have:

```

1  $s^* = 0$  ;
2 for  $(y, z) \in \mathcal{D}_Y \times \mathcal{D}_Z$  do
3    $s_{y,z} = p(z|\bar{x}, y)p(\bar{x}, y)$  ;
4   if  $s_{y,z} > s^*$  then
5     Store new running maximum score and assignment;
6      $s^* = s_{y,z}$  ;
7    $(y^*, z^*) = (y, z)$  ;

```

When we compute the score $s_{y,z}$ in line 3 of the algorithm, both Y and Z are effectively observed which induces a dependence between variables X and Y due to the V-structure. Therefore, when performing Viterbi decoding in the CMM, the conditional independence assumptions do not appear to necessarily cause a problem.

The following is an example:

- X, Y, Z all binary. $p(x), p(y)$ uniform. We observe $\bar{x} = 0$.

- Consider the following table of values for $p(z|x, y)$:

x	y	$p(z = 0 x, y)$	$p(z = 1 x, y)$
0	0	0.4	0.6
0	1	0.7	0.3
1	0	0.7	0.3
1	1	0.4	0.6

- goal: $\max_{y,z} p(z|\bar{x}, y)p(\bar{x})p(y)$.
- So if $\bar{x} = 0$, we'll choose $(y, z) = (1, 0)$
- So if $\bar{x} = 1$, we'll choose $(y, z) = (0, 0)$
- Thus, x can influence the choice of y .

So the above table means that there is not necessarily a problem in a CMM, at least if the probability tables are set to certain values. On the other hand, there are indeed table values that do lead to problems and undesirable properties with a CMM.

How, then, does label-bias occur and how does it produce problems? In many sequence models (such as part-of-speech tagging in NLP), a given state has only a few possible following states. Often it is the case that a state should have only one possible following state and this occurs with probability one, which was the issue depicted in Figure 8.63.

Label-bias can happen in an CMM since there is a form of competition between how strongly a given state variable can influence its successor and how strongly the successor observation can influence the successor state. This competition occurs in the factor $p(q_t|q_{t-1}, x_t)$, and there are several ways to see how this happens.

First, consider the maximization in Equation (8.442), but now suppose that for all values of q_2 there is only one possible value of q_3 that may occur with non-zero probability. In such case, we see that $p(q_3|q_2, \bar{x}_3) = \delta_{q_3=f(q_2)}$ where $f(\cdot)$ is a deterministic function of only q_2 , which means that the observation \bar{x}_3 has no chance to influence the state and will have no effect. In the equations, the function $\phi_{Q_3}(q_2, \bar{x}_3)$ is really only a function of q_2 , so the various values of q_2 are not effected by different values of \bar{x}_3 . This is the label bias problem.

Note that this problem does not happen in the HMM (or CRF) case since the factorization is different. In the HMM case, for example, we see that even if q_2 completely determines q_3 , that doesn't restrict x_3 's

influence on q_3 and there may be a reverse influence from x_3 indirectly via q_3 on scores for possible values for q_2 . This can be seen by considering decoding under an HMM and its factorization properties:

$$\max_{q_1, q_2, q_3} p(q_1, q_2, q_3, \bar{x}_1, \bar{x}_2, \bar{x}_3) = \max_{q_1, q_2, q_3} p(x_1|q_1)p(q_1)p(x_2|q_2)p(q_2|q_1)p(x_3|q_3)p(q_3|q_2) \quad (8.445)$$

$$= \max_{q_1} p(x_1|q_1)p(q_1) \max_{q_2} \left(p(x_2|q_2)p(q_2|q_1) \max_{q_3} (p(x_3|q_3)p(q_3|q_2)) \right) \quad (8.446)$$

$$= \max_{q_1} p(x_1|q_1)p(q_1) \max_{q_2} (p(x_2|q_2)p(q_2|q_1)\phi_{Q_3}(q_2, \bar{x}_3)) \quad (8.447)$$

Even if $p(q_3|q_2) = \delta_{q_3=f(q_2)}$, it is still possible in an HMM for the \bar{x}_3 to effect previous states — the function $\phi_{Q_3}(q_2, \bar{x}_3)$ can be a true function of both q_2 and \bar{x}_3 , meaning changing the values of \bar{x}_3 may change the scores for different values of q_2 . That is, when $p(q_3|q_2)$ is deterministic, we have that $\phi_{Q_3}(q_2, \bar{x}_3) = p(\bar{x}_3|f(q_2))p(f(q_2)|q_2)$ which indeed change values for different values of \bar{x}_3 .

8.8.12.3 Information Theoretic Description of Label Bias in CMMs

We can look at the label-bias problem more generally for the CMM using information theoretic quantities. We next show that in the CMM case, the conditional entropy $H(Q_t|Q_{t-1})$ places a bound on how much information, in the model, the observations may have on the labels. Consider the joint mutual information function:

$$I(Q_{1:T}; X_{1:T}) \quad (8.448)$$

which indicates how much information the observations may have about the labels – clearly, if this quantity is low, relative to a model¹¹, the observations are not informative about the labels. Consider the following sequence of steps:

$$I(Q_{1:T}; X_{1:T}) \stackrel{(a)}{=} \sum_t I(Q_t; X_{1:T}|Q_{1:t-1}) \quad (8.449)$$

$$\stackrel{(b)}{=} \sum_t I(Q_t; X_t|Q_{1:t-1}) \quad (8.450)$$

$$\stackrel{(c)}{=} \sum_t I(Q_t; X_t|Q_{t-1}) \quad (8.451)$$

$$\stackrel{(d)}{=} \sum_t H(Q_t|Q_{t-1}) - H(Q_t|Q_{t-1}, X_t) \quad (8.452)$$

$$\stackrel{(e)}{\leq} \sum_t H(Q_t|Q_{t-1}) \quad (8.453)$$

where (a) follows by the chain rule of mutual information, (b) follows since $Q_t \perp\!\!\!\perp \{X_{t+1:T}, X_{1:t-1}\}|Q_{1:t-1}$ in a CMM, (c) follows since both $Q_t \perp\!\!\!\perp Q_{1:t-2}|\{Q_{t-1}, X_t\}$ and $Q_t \perp\!\!\!\perp Q_{1:t-2}|\{Q_{t-1}\}$ in a CMM, (d) follows by the definition of conditional MI, and (e) follows since conditional entropy is always positive. Therefore, we see that the overall mutual information between the observation sequence and label sequence is bounded above by the sum of the conditional entropy, and if $H(Q_t|Q_{t-1})$ is small (or zero), then this limits how

¹¹We note that when we compute information theoretic quantities “relative to a model”, we assume that the model is defined by a graphical model (in this case a Bayesian network), and we assume that among all models that factor with respect to that graphical model, we take the one that is closest to the truth, which means that all conditional mutual information values are exact except where the model is forced to make conditional independence statements.

much information the observations will have (in a CMM) about the labels. In the extreme case, where $H(Q_t|Q_{t-1}) = 0$, the only uncertainty is $H(Q_1)$ which means that only the first observation will influence the labels. If we go further, and say $H(Q_1) = 0$, then the sequence is entirely determined, and none of the observations influence the labels.

We note moreover that since $I(Q_{1:T}; X_{1:T}) = \sum_t I(Q_t; X_t|Q_{t-1})$, the only time that X_t can influence the labels for a given t is in one of the terms of the sum. Therefore, if there is a value t such that $H(Q_t|Q_{t-1}) = 0$, then that means the one opportunity X_t has to make a difference in the prediction is lost. That is, $H(Q_t|Q_{t-1}) = 0$ implies that X_t has zero influence in the model for this particular t . Another way of saying this is to note, from the above equations, that $I(Q_t; X_t|Q_{t-1}) \leq H(Q_t|Q_{t-1})$ so if $H(Q_t|Q_{t-1}) = 0$ then $I(Q_t; X_t|Q_{t-1}) = 0$ and X_t has no chance to influence the labels. The original rib/rob example of Figure 8.63 showed an example of this. In that example $H(Q_1|Q_0) = 1$ (two possible states q_1 or q_4 with equal chance of being in each). Then $H(Q_2|Q_1) = 0$ which means that X_2 has no influence on the decision.

In general, for a CMM to avoid the label bias problem, it must be the case that $H(Q_t|Q_{t-1})$ in the model is not too small. This can result if the model is trained appropriately [230], and/or if the data is such that it doesn't require a small conditional entropy. Intuitively, when we have a factor $p(q_t|q_{t-1}, x_t)$ in a CMM, if it is the case that $H(Q_t|Q_{t-1})$ must be small, this factor is required to mostly ignore x_t since Q_t is already entirely determined by Q_{t-1} . This effect cascades along the chain so that none of the observations have an influence. More on this in §8.8.12.5.

8.8.12.4 The label-bias problem and HMMs

It is important to realize that the HMM does not have a label-bias problem, as we saw in Equation (8.447). An HMM makes no unconditional independence statements (only conditional ones), so nothing is (unconditionally) independent of anything else in an HMM. More relevantly, a CMM makes different independence assumptions as does an HMM. In an HMM, even if $H(Q_t|Q_{t-1})$ is small, all observations may still influence the label sequence since the observation distributions are separate. In other words, unlike a CMM, an HMM has both $p(q_t|q_{t-1})$, a factor that can easily reflect a low entropy $H(Q_t|Q_{t-1})$, and a factor $p(x_t|q_t)$ that is not necessarily influenced by the low-entropy state conditional. A strong influence between the state and observation can be retained in an HMM.

We have:

Lemma 134 (label bias theorem). *In a CMM, we have that $I(Q_{1:T}; X_{1:T}) \geq \sum_t H(Q_t|Q_{t-1})$. Hence, if there is strong Markov chain deterministic, the observations do not influence the states. Moreover, for any t such that $H(Q_t|Q_{t-1}) = 0$, X_t has no influence.*

Note, this is not true with an HMM or a CRF.

Practically speaking, in many tasks one does want to have small $H(Q_t|Q_{t-1})$. This is particularly true in NLP, from which the original CRF paper [256] was drawn.

So, to summarize: it is possible to produce CMMs that do not have the label-bias problem. It depends on how the parameters are set, which when using objective-function-based optimization, depends on the objective function. Standard ML estimation might not be best in this case, and alternative objective functions might function better [230]. We discuss this further in §8.8.12.5.

8.8.12.5 The label-bias problem and hybrid Deep MLPs/Markov chains

We note that hybrid systems, as described in §5.7.2, where we use the output of a (potentially deep) neural network as virtual evidence scores as input to a Markov chain, also does not suffer from the label bias problem, and this true is even though one might think of the arrows pointing up from the observation to the

hidden state. In a hybrid system, as already mentioned, the best graphical model to describe it is with the observed child mechanism and with a time-inhomogeneous CPT for the observations and this, therefore, factorizes just like the HMM which does not suffer from the label-bias problem.

8.8.13 What about the observation-bias problem?

An effect that can be equally detrimental to label-bias can occur in a CMM when the observations have too strong an influence on the labels. This has been called *observation bias* [248] in the literature.

In many tasks, for a given x_t there is only a small possible set of valid values for q_t , mean that $H(Q_t|X_t)$ is small or zero. If $H(Q_t|X_t) = 0$, then at the current time t , there is sufficient information in X_t alone to determine Q_t . If $H(Q_t|X_t) = 0$ for all time t , then there is no reason to use a dynamic model in the first place and it would be better to utilize multiple static models. Therefore, we would expect that if $H(Q_t|X_t) = 0$ for all t , then the model would also insist that, conditioned on the observations, the states variables at different times are independent.

$$I(Q_{t-1}; Q_t|X_t) = H(Q_t|X_t) - H(Q_t|Q_{t-1}, X_t) \quad (8.454)$$

$$= -H(Q_t|Q_{t-1}, X_t) \quad (8.455)$$

$$= 0 \quad (8.456)$$

which follows since mutual information and discrete random-variable entropy is always positive, the only assignment satisfying the equalities is that $H(Q_t|Q_{t-1}, X_t) = 0$. In fact, letting A and B be any sets of time points, it turns out that $Q_A \perp\!\!\!\perp Q_B|X_{1:T}$, meaning that all state variables are mutually independent given the observations. To see this.

$$I(Q_A; Q_B|X_{1:T}) = H(Q_A|X_{1:T}) - H(Q_A|Q_B, X_{1:T}) \quad (8.457)$$

$$= \sum_i H(Q_{a_i}|Q_{a_{1:i-1}}, X_{1:T}) - \sum_i H(Q_{a_i}|Q_{a_{1:i-1}}, Q_B, X_{1:T}) \quad (8.458)$$

$$\stackrel{(a)}{\leq} \sum_i H(Q_{a_i}|X_{a_i}) \quad (8.459)$$

where (a) follows since entropy is positive and conditioning reduces entropy.

Therefore, we see that as $H(Q_t|X_t)$ gets small, we limit the information that the state variable may have about other variables, and if $H(Q_t|X_t) = 0$, then the state variables may not contain information about each other. But so far, this is true in general, irrespective of any modeling assumptions being active. What happens in a CMM that causes observation bias?

Before answering that, we consider $H(Q_t|X_t) = 0$ for all t to be too strict, as it is unlikely that this would occur for all points t (if it was true, there would be no reason to use a sequential model, as mentioned above). It is perhaps more reasonable to assume that $H(Q_t|X_t) = 0$ for a particular set of time points, and lets assume this set is of size one for now, say t_0 . If this is the case in a CMM, then this severs the connection between the future and past relative to t_0 . In other words, if $H(Q_{t_0}|X_{t_0}) = 0$, then $I(Q_A; Q_B|X_{t_0}) = 0$ when A (resp. B) consists of time points strictly after (resp. before) t_0 .

To see this, consider the following expansion:

$$I(Q_A; Q_B | X_{t_0}) = H(Q_A | X_{t_0}) - H(Q_A | Q_B, X_{t_0}) \quad (8.460)$$

$$= H(Q_A | X_{t_0}) + 0 - H(Q_A | Q_B, X_{t_0}) - 0 \quad (8.461)$$

$$= H(Q_A | X_{t_0}) + H(Q_{t_0} | X_{t_0}) - H(Q_A | Q_B, X_{t_0}) - H(Q_{t_0} | X_{t_0}) \quad (8.462)$$

$$\stackrel{(a)}{=} H(Q_A | X_{t_0}) + H(Q_{t_0} | X_{t_0}, Q_A) - H(Q_A | Q_B, X_{t_0}) - H(Q_{t_0} | X_{t_0}, Q_B, Q_A) \quad (8.463)$$

$$\stackrel{(b)}{=} H(Q_A, Q_{t_0} | X_{t_0}) - H(Q_A, Q_{t_0} | X_{t_0}, Q_B) \quad (8.464)$$

$$\stackrel{(c)}{=} H(Q_{t_0} | X_{t_0}) + H(Q_A | X_{t_0}, Q_{t_0}) - H(Q_{t_0} | X_{t_0}, Q_B) - H(Q_A | X_{t_0}, Q_B, Q_{t_0}) \quad (8.465)$$

$$\stackrel{(d)}{=} 0 + H(Q_A | X_{t_0}, Q_{t_0}) - 0 - H(Q_A | X_{t_0}, Q_B, Q_{t_0}) \quad (8.466)$$

$$= I(Q_A; Q_B | X_{t_0}, Q_{t_0}) = 0 \quad (8.467)$$

where (a) and (d) follow since conditioning reduces entropy, the non-negativity of entropy, and since $H(Q_{t_0} | X_{t_0})$ is already zero; (b) and (c) follow by the chain rule of entropy. The final equality follows since $\{X_{t_0}, Q_{t_0}\}$ d-separates Q_A from Q_B . Therefore, what happens before t_0 can not influence after t_0 if there is a time point at which the state is determined by the observation.

It is important to note that in the above analysis, we did not make any assumptions that were particular to a CMM. This means that if $H(Q_{t_0} | X_{t_0})$ is too small, then any sequential model, a CMM, HMM, or CRF, will essentially be severed in two at t_0 . Intuitively, this should make sense as if $H(Q_{t_0} | X_{t_0}) = 0$, then Q_{t_0} becomes effectively observed in the model, and since it is a separator it renders independent that which comes before and after it. In order to overcome this problem, therefore, we must make Q_{t_0} a non-separator, something that can be done by using a hire-order Markov chain. For example, in a 2nd-order HMM, the state transition distributions are of the form $p(q_t | q_{t-1}, q_{t-2})$. Even if q_{t-1} is effectively observed, q_{t-2} can influence the distribution on q_t and keep the model from being partitioned into two independent sequences.

Lemma 135 (observation bias theorem). *Suppose $H(Q_{t_0} | X_{t_0}) = 0$ for some t_0 and let A be a set of time indices before and B be indices after t_0 . Then $I(Q_A; Q_B | X_{t_0}) = 0$.*

When the state is determined from the observations at a time, it cuts the model by rendering the left and right half of the model independent conditioned on X_t .

We can generalize this somewhat using any dynamic graphical model via the notion of a separator. In any dynamic graphical model, there is some notion of present that renders the past and future independent. If it is the case that the portion of the model constituting the present is determined given the observations, in the model, (i.e., if $H(\text{present} | X_{t_0}) = 0$) then this will split the model into two independent chunks, the past and future.

But what about the “observation bias” issue? Observation bias can occur when the forced choice at a given time might not influence choices at other times. For example, if at time t_0 , $H(Q_{t_0} | X_{t_0}) = 0$ then $p(q_{t_0} | q_{t_0-1}, x_{t_0})$ will yield the same distribution over q_{t_0} for all values of q_{t_0-1} rendering q_{t_0-1} irrelevant. This means that the decision on q_{t_0-1} would need to be based only on x_{t_0-1} and possibly other values of q_t for $t < t_0 - 1$. The observation bias problem occurs in CMMs when all of the following occur:

1. $H(Q_{t_0} | X_{t_0}) = 0$, and let \bar{q}_{t_0} indicate the value compatible with \bar{x}_{t_0} ,
2. $H(Q_{t_0-1} | X_{t_0-1})$ is also small, and let \bar{q}_{t_0-1} indicate the value compatible with \bar{x}_{t_0-1} , and
3. $\{\bar{q}_{t_0-1}, \bar{q}_{t_0}\}$ is a very rare (low probability) event, according to the model.

The problem arises since the decision to choose \bar{q}_{t_0-1} and \bar{q}_{t_0} is based on $H(Q_{t_0}|X_{t_0})$ and $H(Q_{t_0-1}|X_{t_0-1})$ being small, but due to the model there is nothing to counterweight that with the rarity of the event $\bar{q}_{t_0-1}, \bar{q}_{t_0}$. Assuming that $T = t_0 = 2$, the Viterbi optimization becomes

$$\operatorname{argmax}_{q_1, q_2} p(q_1|\bar{x}_1)p(q_2|q_1, \bar{x}_2) = \operatorname{argmax}_{q_1} p(q_1|\bar{x}_1) \operatorname{argmax}_{q_2} p(q_2|\cdot, \bar{x}_2) \quad (8.468)$$

which follows since q_1 is irrelevant in the second factor. Hence, the rare event $\{\bar{q}_{t_0-1}, \bar{q}_{t_0}\}$ might become the Viterbi path without there being a chance for the prior marginal probability of that event to express itself. Note that observation bias is not a necessary property of CMMs, but one that occurs when conditional entropies $H(Q_{t_0}|X_{t_0})$, according to the model, are near zero.

Note that observation bias problem does not occur with HMMs since the HMM has a separate factor for $p(q_t|q_{t-1})$ which can be used to down-weight any rare events. For example, in the optimization within the HMM, we have

$$\operatorname{argmax}_{q_1, q_2} p(\bar{x}_1|q_1)p(q_2|q_1)p(\bar{x}_2|q_2) \quad (8.469)$$

and if a particular pair q_0, q_1 was rare, then the factor $p(q_2|q_1)$ can always discourage it (or even preclude it if $p(q_2|q_1) = 0$) even when the factors $p(\bar{x}_1|q_1)$ and $p(\bar{x}_2|q_2)$ attempt to guarantee it.

What happens to the Viterbi score in an HMM when the only states q_1, q_2 that give observations \bar{x}_1, \bar{x}_2 a non-zero score in $p(\bar{x}_1|q_1)p(\bar{x}_2|q_2)$ are ones such that $p(q_2|q_1) = 0$? This means the Viterbi score, and in fact all scores, are zero. Indeed, an HMM is a joint distribution $p(\bar{x}_1, \bar{x}_2, q_1, q_2)$ and it could very well be that $\sum_{q_1, q_2} p(\bar{x}_1, \bar{x}_2, q_1, q_2) = p(\bar{x}_1, \bar{x}_2) = 0$.

Exercise 136. Let $p_{\text{HMM}}(q_{1:T}, x_{1:T})$ be an HMM joint distribution over $Q_{1:T}, X_{1:T}$, meaning that it obeys the HMM factorization requirements. Let $p_{\text{CMM}}(q_{1:T}, x_{1:T})$ be a CMM joint distribution. Derive an HMM distribution $p_{\text{HMM}}(q_{1:T}, x_{1:T})$ such that when we find the CMM distribution $p_{\text{CMM}}(q_{1:T}, x_{1:T})$ that minimizes the KL-divergence $D(p_{\text{HMM}} \| p_{\text{CMM}})$, where p_{HMM} is fixed, we see significant label bias. That is, find p_{HMM} so that

$$p_{\text{CMM}}^* \in \operatorname{argmin}_{p \in \mathcal{F}(G, R)} D(p_{\text{HMM}} \| p) \quad (8.470)$$

has label bias, where G is the Bayesian network for CMMs (e.g., Figure 8.64).

Next, derive an HMM distribution $p_{\text{HMM}}(q_{1:T}, x_{1:T})$ such that when we find the CMM distribution $p_{\text{CMM}}(q_{1:T}, x_{1:T})$ that minimizes the KL-divergence $D(p_{\text{HMM}} \| p_{\text{CMM}})$, we see significant observation bias.

8.8.14 How to overcome label/observation bias.

	Label Bias	Observation Bias
state	$H(Q_t Q_{t-1}) \approx 0$	$\exists(q_t, q_{t-1}) \text{ s.t. } p(q_t q_{t-1}) \approx 0$
observation		$H(Q_t X_t) \approx 0, H(Q_{t-1} X_{t-1}) \approx 0$

Table 8.3: Summary of the conditions for label and observation bias in a CMM. The table gives the conditions for either label or observation bias in terms of what could cause these conditions. The conditions refer to properties of set of random variables that, if true, would make it such that the CMM attempting to obey these properties could have the biases.

As mentioned, label- and observation- bias do not occur with either HMMs or CRFs, so one way to overcome them is simply to not use an CMM. It is perhaps interesting to note that label bias might occur

when $p(q_t|q_{t-1}) \approx 1$ while observation bias might occur when $p(q_t|q_{t-1}) \approx 0$, and these properties being true of the pair can cause problems since the only factor involving the pair in a CMM is $p(q_t|q_{t-1}, x_t)$. We summarize the conditions in Table 8.3. We saw that label/observation bias is not a necessary condition of CMMs, but rather is a consequence of certain settings of an CMM's parameter values. Therefore, one way to avoid the issue is to utilize different parameter values, ones that avoid the conditions given in Table 8.3.

A typical way of setting parameters is via optimizing a training objective. If the objective function that is optimized is maximum likelihood, then it may not be possible to avoid label bias. For example, let $C()$ be a generic count function as computed by the training data. I.e., $C(A = a, B = b)$ is the number of times that random variable A is value a and jointly B is value b in the training data. Suppose the training data is such that every time state q_0 occurs, it is followed by state q_1 . This means that $C(Q_t = q_0, Q_{t+1} = q_1, X_{t+1} = x) = C(Q_t = q_0)$ so the maximum likelihood estimate becomes:

$$P(Q_{t+1} = q_1 | Q_t = q_0, X_{t+1} = x) = \frac{C(q_0, q_1, x)}{C(q_0, x)} = 1 \quad (8.471)$$

so this means that regardless of the observations, the CPT in this case is always 1 for these values.

How can this be avoided? One possibility is to utilize smoothed or regularized estimates of the CPTs, such as Dirichlet prior probabilities in a Bayesian approach, or to utilize CPTs that employ the type of backoff-based smoothing commonly found in language models [36].

An alternative strategy is to train a CMM by optimizing a different objective function — this is the approach advocated and evaluated in [230], where in fact they find that the label bias problem does not occur with CMMs when the objective function does not require that the conditional state distribution be completely accurate in a maximum likelihood sense.

On the other hand, for many applications label-bias would be unavoidable if one were to use a CMM. For example, consider applications where segments are relatively long (in terms of the average number of frames per segment), segments correspond to state values, and where there are relatively few transitions between these states. In such case, a distribution that reflects the training data would have $p(q_t|q_{t-1}) \approx 1$ and label bias would occur. A particularly egregious example of this is when a certain subsegment occurs only at the end of a segment. Once the model hypothesizes entering the final segment, it would ignore all observations that might indicate evidence contrary to being in that final segment, but a CMM would be unable to recover from this. A CMM can then exhibit “one-way door” behavior, once you are in you are unable to get out. This is another example of future observations being unable to influence previous states that we saw in §8.8.12.1. An particularly severe example of this is described in [94], where the goal is to identify signature lines in email messages which typically occur only at the end of the message.

8.8.15 Are CRFs better than HMMs? Are HMMs better than CRFs?

CRFs are inherently trained discriminatively, while HMMs may be trained either generatively or discriminatively. Given that CRFs and HMMs define the same class of conditional distributions (as described in §8.8.6, it would seem that since HMMs may be treated both generatively and discriminatively, HMMs are better.

On the other hand, many empirical results in the literature have demonstrated that CRFs outperform HMMs on a number of different sequential classification tasks. CRFs, being within the class of log-linear models, are relatively easy to use and are easy to train. It is easy to produce a long list of log-linear features that can be used in such models and trained by the same convex method. Also, many of the comparisons between CRFs and HMMs have not considered discriminative training procedures for the HMMs and have only trained the HMMs generatively. CRFs, moreover, make it easy to involve long windows of observations in each factor. HMMs, however and as we have argued in §8.8.8, also have this ability if one does not wish for them to be generatively accurate. It is this authors opinion, therefore, that since HMMs and CRFs define

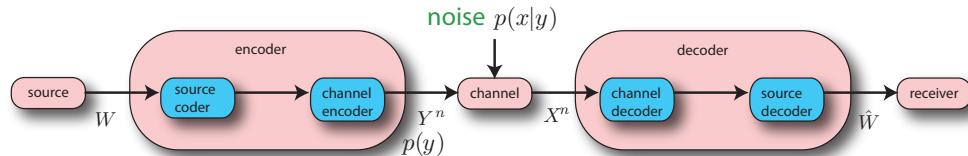


Figure 8.66: Source/channel model of communications, which is an inspiration for generative models in pattern classification, machine learning, and speech recognition.

inherently the same class of conditional distribution, there is no inherent advantage to one over the other when wishing to model a conditional distribution. It is acknowledged, however, that using either CRFs or HMMs might be easier or better in practice due to the availability of, familiarity with, or computationally efficiency of a toolkit for one or the other.

HMMs have the ability to be treated generatively, however, and if there are valid reasons to use generative models, then HMMs maintain that benefit. In the next section, we discuss a number of reasons why one might wish to use generative models.

8.8.16 Why ever use generative models?

Hidden Markov models and Dynamic Bayesian networks, as we will see, are generative models in that they represent families of joint probability distributions over a set of random variables corresponding to vertices in a graph.

Firstly, what is a generative model? Traditionally, generative models came from the area of communications and information theory [97]. Consider Figure 8.66. There is some message source that generates messages encoded by a random integer W . These messages are encoded into a string of symbols, $Y_{1:T}$, which represents T instances of random variable Y , each of which has distribution $p(y)$ but more generally there is some joint distribution $p(Y_{1:n} = y_{1:n})$. These variables are sent through a noisy channel, where the noise is random and governed by distribution $p(x|y)$. The received message takes the form of T random variables $X_{1:t}$, and these are then decoded to come up with an estimate \hat{W} of the original message. note that “decoder” is the inverse system of the “encoder” and it attempts to recover the original source signal or some “subpart” of the original source signal. This model is based on the idea that communication is sending information from one place and/or time to another place and/or time over a medium that might have errors. The model has in fact become pervasive, and originated in 1948, in a classic paper by Claude E. Shannon entitled “The Mathematical Theory of Communications”, the paper that single-handedly created this field of information theory. Shannon’s work grew out of solving problems of electrical communication during WWII, but information theory applies to many other fields as well including pattern recognition.

Pattern recognition is the process of taking a signal X and deciding the best explanation Y for this X from among some set of possible explanations. In the 1950s and 1960s, it became clear that Shannon’s model of communication could also be applied to pattern recognition. One classic example is that of speech recognition, where we have:

- Source: human thought, speakers brain
- Encoder: Human Vocal Tract, Sequences of Articulatory Gestures,
- Channel: air, sound pressure waves, general acoustic environment where speech might be spoken.
- Noise: background noise which could include other speakers (the notorious cocktail party effect, where humans can understand speech even in very noisy environments, like at a cocktail party), or other environmental sounds like fan noise, cars, trains, wind noise, etc.

- Decoder: human auditory system acts as the decoder, as its job is to take the acoustic signal and transform it back into a message:
- Receiver: a human that receives the message which might be considered human thought, or some representation in the listeners brain.

Now when we produce a model for this process, we produce a source distribution $p(y)$ and a channel distribution $p(x|y)$ and model the joint in this fashion: $p(x, y) = p(y)p(x|y)$ and this is the classic source-channel description of generative models. We call it a “generative model” since it models the process by which messages (as received by a receiver) are generated from inception (via $p(y)$) and modification (via $p(x|y)$).

Note that there are some important differences in communications theory and pattern recognition. In communications theory, we generally know the encoder. I.e., we know the process by which we take an original message W and transform it into channel symbols Y . I.e. the code in this case is human-made. In pattern recognition, however, we do not know this process. I.e., in speech recognition we do not know the representation of the original message (human thought, or whatever it may be if it even exists) nor do we know the process by which it is transformed into Y (e.g., acoustic speech). This in some sense makes pattern recognition a more challenging problem. Hence, much of the challenge in pattern recognition (e.g., object recognition in a sound source, an image source, etc.) can be seen as the decoder aspects of the communication model (we don’t know for certain the code).

A perfect example example of a generative model is an HMM, where we model $p(y) = p(y_{1:T}) = \prod_{t=1}^T p(y_t|y_{t-1})$ as a first-order Markov chain, and we model $p(x|y) = p(x_{1:T}|y_{1:T}) = \prod_{t=1}^T p(x_t|y_t)$ as product of individual factors, as we saw in §8.4.9. Another classic example of a generative model would be a naïve Bayes classifier where y is a single integer, and $p(x|y)$ factorizes over all elements of x .

Generative models can be used for classification, in the same way that the source-channel communications model is used for communication. I.e., given an unknown object \bar{x} (a received message), find the object (original source message) y that best explains \bar{x} . This means, given \bar{x} , find the y^* that maximizes

$$y^* \in \operatorname{argmax}_y p(y)p(\bar{x}|y) \quad (8.472)$$

If $p(y)$ and $p(x|y)$ are accurate, then this is a valid way to do pattern classification in the sense that this process is equivalent to:

$$y^* \in \operatorname{argmax}_y p(y)p(\bar{x}|y) = \operatorname{argmax}_y p(y, \bar{x}) = \operatorname{argmax}_y p(y|\bar{x}) \quad (8.473)$$

which means that y^* minimizes the error probability, where the error probability is given as $1 - p(y^*|\bar{x})$.

Unfortunately in practice, we do not know the true generative process, and so $p(y)$ and $p(x|y)$ are far from perfect, and so the above procedure is not guaranteed to minimize error. If we have a procedure that based on finite data, tries to find the $p(y)$ and $p(x|y)$ that maximizes likelihood, then we will have an inaccurate decision making process.

Now before going further, we should make the distinction between a generative model and objectives for training that generative model. A generative model can be trained using a variety of methods. In the simple case above, lets say that the generative model $p_\theta(x, y) = p_{\theta_y}(y)p_{\theta_{x|y}}(x|y)$ consists of parameters $\theta = (\theta_y, \theta_{x|y})$ for training each factor. A generative training procedure would include a method like maximum likelihood (ML), where given a training set $\mathcal{D} = \{(x^{(i)}, y^{(i)})\}_i$, one would perform the following optimization:

$$\theta_g^* \in \operatorname{argmax}_\theta \prod_i p_\theta(x^{(i)}, y^{(i)}). \quad (8.474)$$

The ML parameter estimation method can be shown to minimize the learnt distribution with empirical distribution as reflected in the training data, and it is also asymptotically consistent. There are many ways to optimize the parameters besides ML. Simple modifications might include a regularizer, but that would not make for a discriminative parameter estimation procedure.

There are a wealth of discriminative parameter training methods that can be used to train generative models, and we do not discuss them here. A discriminative parameter training method for generative models is one that adjusts the models parameters to optimize an objective that measures some quality of the model that reflects how well it might do in a classification tasks. For example, a simple one would be conditional maximum likelihood and would take the form:

$$\theta_d^* \in \operatorname{argmax}_{\theta} \prod_i p_{\theta}(y^{(i)} | x^{(i)}). \quad (8.475)$$

where $p_{\theta}(y^{(i)} | x^{(i)}) = \frac{p_{\theta}(y^{(i)}, x^{(i)})}{p_{\theta}(x^{(i)})} = \frac{p_{\theta}(y^{(i)}, x^{(i)})}{\sum_y p_{\theta}(y, x^{(i)})}$. Therefore, while the parameters θ govern the generative model $p(x, y)$, they are not being adjusted to be generatively accurate, rather they are being adjusted so that when one forms the conditional distribution $p_{\theta}(y^{(i)} | x^{(i)})$, they will be accurate. We should compare this form with the conditional HMM distribution discussed in §8.8.6. So while the generative model's parameters are being adjusted to produce a conditional distribution that is accurate, its generative accuracy may suffer. That is, while $p_{\theta_d^*}(y|x)$ might produce a low error, the resulting generative joint distribution $p_{\theta_d^*}(x, y)$ might differ quite significantly from either $p_{\theta_g^*}(x, y)$ or $p(x, y)$. When the goal is classification, this of course doesn't matter. Indeed, as mentioned in §8.8.9, HMMs with delta features will necessarily not be generatively accurate but can (and usually do) discriminate better.

A discriminative model on the other hand is one that inherently produces a score that can be used solely for the purposes of classification. For example, we might use a multi-layered perceptron (MLP) or a deep neural network [25] to produce a distribution $p_{\theta_{y|x}}(y|x)$ that is not formed by a ratio as before. That is, there is no way in the model to go from $p_{\theta_{y|x}}(y|x)$ back to the generative model since there is nothing that has even been adjusted to produce a joint distribution. In some cases, in fact, the conditional form of a generative distribution and the conditional distribution are the same, i.e., a ratio of naïve Bayes Gaussians can lead to a logistic model [326]. In general, however, (e.g., for a neural network) there is no known ratio of generative models that lead to the corresponding discriminative form. In fact, a discriminative model need not be a probability distribution at all, and instead could be a process that takes a given \bar{x} and y and produces a score, $s(\bar{x}, y)$, what are sometimes called discriminant functions. It is often the case that it is still possible to learn them, in that an optimization procedure can be set up that allows any parameters to be adjusted so that

$$\operatorname{argmax}_y s(\bar{x}, y) \quad (8.476)$$

produces accurate results.

So, if discriminative models inherently only model a process of mapping from x to y , why should we use a generative model at all? There are, indeed, a number of reasons still for using generative models as we outline in the following:

8.8.16.0.1 It is conceptually easy to think of problems generatively: The source-channel model of communication described above is conceptually easy, as it allows one to produce a model with parts that have correspondents in the real world. In many cases, there can be easy to produce hierarchical decomposition of objects, where objects are composed of their constituent parts, and this works recursively down to the level of the observations x . Hence, it is possible to incorporate high-level domain knowledge about the task. For example, in speech, when our task is to recognize sentences (so that $y \in D_Y$ might be a sequence of words), we know that sentences consist of words, words consist of syllables and phones, phones themselves have

constituent parts that may be learned, and then these parts may have realizations in term of the observations. This hierarchical decomposition of speech recognition is still used today, in the most recent state-of-the-art systems. Other domains, too, may take advantage of high-level knowledge to form a generative process. For example, in tandem mass spectrometry (MS/MS) based peptide and protein identification, a sub-area in computational biology, one sometimes uses a generative process to model the way in which peptides are fragmented via multiple successive processes into resultant spectra.

8.8.16.0.2 Unsupervised learning, clustering, data interpretation, inducing hidden causes in data:

Unsupervised learning is the process of learning a mapping from X to Y without ever seeing any observed values for Y . Clustering is the process by which different samples from X are grouped into clusters. Data interpretation is the goal of these procedures. Inducing hidden causes in data is deducing in some way the underlying (perhaps causal [339]) explanations for the observed data that we see. All of these problems are quite similar to each other. The idea is that there is some observed data represented by X we wish to explain this data with some hidden cases Y , and we wish to learn the best set of hidden causes that describe the data. One example of this is independent component analysis, where Y is a vector of hidden variables that are marginally independent but conditionally dependent (conditioned on X). We encode our hypotheses about the nature of and interactions within the underlying cause of what we observe X and then we can use a learning procedure (such as the EM algorithm) to learn a good set of parameters, subject to these hypotheses. This is shown in Figure 8.67. A sequential example of this the factorial HMM shown in Figure 8.70a.

8.8.16.0.3 The decoding problem is easy: In general, the optimal decoding procedure is maximum likelihood decoding, where we perform $\max_y p(\bar{x}, y)$. In many cases, the model involves hidden variables in the joint distribution of the form $p(x, h, y)$ and decoding should hence become $\max_y \sum_h p(x, h, y)$. As this can be intractable as we saw in §(), a surrogate decoding algorithm is Viterbi decoding: $\max_{y,h} p(x, h, y)$ and we use whatever y is returned. The Viterbi algorithm is quite successful in many pattern recognition tasks, is relatively easy to code, is not computationally intractable in many cases, and works well. The Viterbi algorithm's origin was based on the (generative) source-channel model of communications. Of course, the Viterbi algorithm can be used for discriminative models as well, so it could be argued that this is not an advantage to generative models but a feature of both.

8.8.16.0.4 Generative decomposability: . Often generative models are decomposable, so that parts of the model can be trained separately. Moreover, when they are trained using a generative criterion (i.e., if the optimization criterion decomposes along with the probability model) then this can make training very fast, and easy to parallelize. For example, if our generative model is of the form $p(x, h, y) = p(x|h)p(h|y)p(y)$, and there is no variational dependence between the parameters of $p(x|h)$ and $p(h|y)$ then each of these factors can be trained entirely separately. In ASR, for example, we can separately train the language model, the pronunciation model, and the low-level acoustic model all at the same time.

This can be particularly useful when we have different size training sets for different components. For example, we might two training data sets $\mathcal{D}_1 = \{(x^{(i)}, h^{(i)})\}_{i=1}^{N_1}$ and $\mathcal{D}_2 = \{(h^{(i)}, y^{(i)})\}_{i=1}^{N_2}$, where $N_2 \gg N_1$. With a generative model, we can use \mathcal{D}_1 to train $p(x|h)$ and use \mathcal{D}_2 to train $p(h|y)$, something not easy to do if we were to learn $p(y|x)$ directly. Examples where this comes up is in speech recognition and statistical machine translation, where a much larger text corpus can be used to train the language model

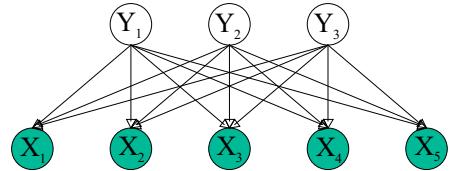


Figure 8.67: Independent component analysis (ICA) — the Y vector is independent and explains the data as represented by the X vector.

separately from the rest of the model. The key thing is that with generative models, we might have different amounts of data used to train each of the different factors, and this is possible to do with generative models, since each factor is self contained (i.e., each factor is locally normalized).

We note that when parameters are tied, this can change as the parameters are no longer variationally independent. I.e., it might be that $p_{\theta_x|h}(x|h)$ and $p_{\theta_h|y}(h|y)$ use parameters that depend on some higher-level parameter, e.g., we could have that both $\theta_{x|h}(\theta)$ and $\theta_{h|y}(\theta)$ are functions of a higher level parameter θ which is the goal of the optimization. Alternatively, in a Bayesian context, both $\theta_{x|h}$ and $\theta_{h|y}$ might be random variables and be both dependent on some other (hyper) random variable θ . While this provides much flexibility to generative models, it can complicate the training process.

Most discriminative models, however, do not decompose in any form. Parameters must, in such case, be trained jointly leading to more complex training processes.

8.8.16.0.5 Parameter sharing/tying is easy with generative models: Due to the hierarchical nature of generative models, it makes it easy to share parameters. For example, in $p(x, h, y) = p(x|h)p(h|y)p(y)$ there might be multiple values of h that are constrained to produce the same distribution $p(x|h)$. As an example, for word models in ASR, the models for /cat/ and /bat/ can share model parameters for the middle phone /a/. Sharing can reduce parameter estimation variance for the sake of bias (since more effective training data is available). Moreover, since the model is generative, it is easy to use high-level domain-knowledge to determine the strategy for sharing. GMTK has many flexible options for parameter sharing (cf. §), and also includes a program `gmtkTie` to try to make sharing decisions automatically in some circumstances (cf. §).

8.8.16.0.6 Generative models might require less training data to do well: There have been both theoretical and practical arguments supporting the case that generative models require less training data to perform well than do discriminative models [326]. Speech recognition history has supported this over the years, as early attempts at discriminative training of HMM systems did not yield significant improvements, perhaps due to the lack of a sufficient amount or quality of training data. On the other hand, there are other empirical results demonstrating cases where generative models do not use less training data to achieve a good results [248], so it could be that more research is needed on this front.

8.8.16.0.7 Rarity of events: Generative models $p(x|y)p(y)$ represent $p(x)$ implicitly, and if x is very rare $p(x)$ is very small which might help us produce confidence values about the classification decision. Using just $p(c|x)$ as in a discriminative model, we will be unknowable about how rare x is and could blindly classify x with confidence, no matter how rare or unlikely x is. A generative model, hence, could be used to provide better confidence estimates on any decision being made based on how rare or unlikely x is.

8.8.16.0.8 Dealing with missing information: Ordinarily, we say that x is observed and denote this as \bar{x} . In cases where part of x is missing, then it can be important to still be able to make a decision. Lets say that x is a vector partitioned into two parts $x_{1:T} = (x_O, x_H)$ where H is an index set where the values are unknown. In many cases, a generative model can easily handle missing information. For example, given $p(x, y)$, it is sometimes possible to construct $\sum_{x_H} p(x, y) = p(x_O, y)$ and make a decision based on $\text{argmax}_y p(\bar{x}_O, y)$ — HMMs with missing features are a good example of a case when this is possible. With a discriminative model, there is no representation of $p(x)$ and so it is not, in general, possible to construct $p(y|x_O)$. Note that this advantage is retained even if the generative model is trained or structured discriminatively (see below), since it will stay a generative model.

8.8.16.0.9 Generative models may be trained discriminatively: As mentioned above, generative models need not be "generative", they can still be trained using a discriminative parameter optimization criterion as mentioned above. There are many criterion, each with its own advantages and challenges. Although decomposability of the generative model is often lost in this case, it means we can still formulate the model using high-level domain knowledge but then train the parameters discriminatively. What is left after training might not accurately model the generative process any longer, but might classify well.

8.8.16.0.10 Generative models can be structured discriminatively: The naïve Bayes model is most often wrong from a generative perspective but often performs well in classification. As argued in S8.8.9, HMMs have this property as well, but they still perform well for classification. Such model are said to be structurally discriminative generative models [40, 470], since their inherent structure allows them to classify well. In fact, producing models that are so structured might be a goal of an optimization procedure itself [319, 341].

8.8.16.0.11 Class skew/imbalance: Discriminative models are often quite sensitive to class data skew. This happens when the true generative distribution priors are imbalanced, where, for example, there might be $y_1, y_2 \in D_Y$ such that $p(y_1) \ll p(y_2)$. Since many discriminative models represent $p(y|x)$, they will tend to learn the priors and in some cases ignore the data (i.e., a very rare y_1 might never be classified, for a given x). Examples of models that have this problem include neural networks/MLPs. Even support vector machines have this problem

Generative models, on the other hand, learn $p(x|y)$, and hence resulting the scores are less sensitive to this problem. That is, as long as there are some samples of x for a given class y then the generative model has a representation for it. In fact, different regularization strategies can be used, in generative models, for different values of y . For cases where $p(y) \approx 0$ we might want to strongly regularize $p(x|y)$ (or tie its some of its parameters with classes that are close but more frequent) and when y is frequently represented in training data, we may not need to regularize very much.

Of course, there are ways to deal with this with discriminative models. Some common strategies to reduce class imbalance issues are:

1. Divide by the prior to produce scaled likelihoods. I.e., we form i.e., $p(y|x)/p(y) = p(x|y)/p(x)$ giving scaled likelihoods which can then be used as a scoring function to decide on y for a given \bar{x} .
2. Reducing training data set of classes so that the training priors are uniform. That is, if a given y_1 is abundant in training data but y_2 is not, we might sub-sample from those training data items of label y_1 . The problem with this approach, of course, is that it reduces the size of the data set and hence can lead to less information to learn from. It is possible to do this multiple times, to learn multiple different models, perhaps randomly, and combine them in various ways.
3. It can be possible to learn threshold on a cross-validation held-out set. For example, with binary classification, rather than deciding y if $p(y|x) > 0.5$, a different threshold could be learned.
4. Mutation of existing data: use high level knowledge to generate new samples of the class that needs more samples, or mutate existing samples, even one could generate random data.
- 5.
6. Of course, we can use a generative model that produces $p(x|y)$ and use this as the score since it is not sensitive to prior skew.

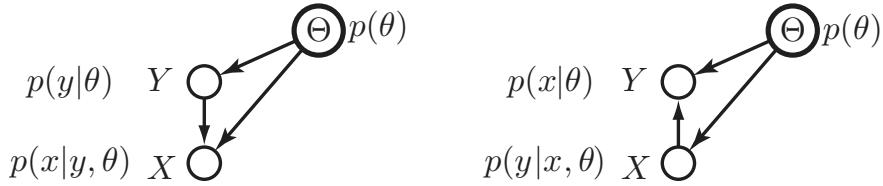


Figure 8.68: Bayesian parameter dependence between marginal $p(x)$ and conditional $p(y|x)$ distribution.

8.8.16.0.12 Dependence between the parameters of $p(y)$ and $p(x|y)$: In [192], the notion of *variational dependence* between parameters was described. They argued that for certain Bayesian networks (or factorization or constraint models), it may be that there is a parameter dependence between θ_x (the parameters of the model $p(x)$) and θ_m (the parameter of the joint marginal $p(y, x)$). What this means is that there is a function f with $\theta_x = f(\theta_m)$, and the Jacobian of $f()$ is non-singular for many values of θ_m meaning that the transformation is “invertible.” Hence, knowing θ_x may restrict the set of possible θ_m which may also restrict the conditional $p(y|x)$. Now, this is not guaranteed to always be true, but in some cases it might be true. When there is variational parameter dependence, we can obtain θ_m from θ_x , and then we can obtain $\theta_{y|x}$ from θ_m , so there may also be a dependence between $\theta_{y|x}$ and θ_x . In such case, estimating the joint distribution $p(c, x)$ may benefit the parameter estimation procedure for conditional model $p(y|x)$. Of course, once we have the parameters for $p(y|x)$ we need not retain any of the irrelevant parameters in $p(x)$, but the point is that there is utility in estimating first a generative model for the sake of better estimating the conditional model.

We may generalize this to a Bayesian estimation procedure, as shown in Figure 8.68. On the left, we see that both factors $p(y)$ and $p(x|y)$ now depend on a random variable θ that is common to both factors. Assuming that θ is hidden during training, it can be beneficial to both factors to train them jointly since information obtained from training, say $p(y|\theta)$ will help to train $p(\theta)$ and hence will benefit $p(x|y, \theta)$, and vice versa. A similar situation holds in the right Bayesian context, where the factors are now $p(y|x, \theta)$ and $p(x)$.

In any event, when ever there is some form of parameter dependence between the conditional distribution $p(y|x)$ and the joint distribution $p(x, y)$ it can be beneficial to learn a generative model even if the final goal is the conditional $p(y|x)$ for classification purposes.

8.8.16.0.13 Brown’s argument: An argument was made in favor of generative models in [67]. Rather than paraphrase it, we quote it directly, and then relate it to other points made in this section. Note, the argument is made in the context of statistical machine translation, where the goal is to translate from French strings f to English strings e , and one can do this generatively by modeling $p(e, f) = p(e)p(f|e)$ as in the source-channel model of communication mentioned above, or alternatively discriminatively via the model $p(e|f)$. Clearly, e takes the role of y (the hidden variables) in our discussion, and f takes the role of x (the observed variables) and our goal is, given an instance of f , find the best e . Here is the quote from [67], starting at page 265:

... We arrive, then, at the Fundamental Equation of Machine Translation:

$$\hat{e} = \operatorname{argmax}_e \Pr(e)\Pr(f|e) \quad (8.477)$$

... Equation (8.477) summarizes the three computational challenges presented by the practice of statistical translation: estimating the *language model probability*, $\Pr(e)$; estimating the *translation model probability*, $\Pr(f|e)$; and devising an effective and efficient suboptimal search for

the English string that maximizes their product. We call these the language modeling problem, the translation modeling problem, and the search problem.

... Why do we estimate $\Pr(e)$ and $\Pr(f|e)$ rather than estimate $\Pr(e|f)$ directly? We are really interested in this latter probability. Wouldn't we reduce our problems from three to two by this direct approach? If we can estimate $\Pr(f|e)$ adequately, why can't we just turn the whole process around to estimate $\Pr(e|f)$? To understand this, imagine that we divide French and English strings into those that are well-formed and those that are ill-formed. This is not a precise notion. We have in mind that strings like *Il va à la bibliothèque*, or *I live in a house*, or even *Colorless green ideas sleep furiously* are well-formed, but that strings like *à la va Il bibliothèque* or a *I in live house* are not. When we translate a French string into English, we can think of ourselves as springing from a well-formed French string into the sea of well-formed English strings with the hope of landing on a good one. It is important, therefore, that our model for $\Pr(e|f)$ concentrate its probability as much as possible on well-formed English strings. But it is not important that our model for $\Pr(f|e)$ concentrate its probability on well-formed French strings. If we were to reduce the probability of all well-formed French strings by the same factor, spreading the probability thus liberated over ill-formed French strings, there would be no effect on our translations: the argument that maximizes some function $f(x)$ also maximizes $cf(x)$ for any positive constant c . As we shall see below, our translation models are prodigal, spraying probability all over the place, most of it on ill-formed French strings. In fact, as we discuss in Section 4.5, two of our models waste much of their probability on things that are not strings at all, having, for example, several different second words but no first word. If we were to turn one of these models around to model $\Pr(e|f)$ directly, the result would be a model with so little probability concentrated on well-formed English strings as to confound any scheme to discover one.

The two factors in Equation (8.477) cooperate. The translation model probability is large for English strings, whether well- or ill-formed, that have the necessary words in them in roughly the right places to explain the French. The language model probability is large for well-formed English strings regardless of their connection to the French. Together, they produce a large probability for well-formed English strings that account well for the French. We cannot achieve this simply by reversing our translation models.

There are two main points in this argument: 1) I.e. the model $p(e|f)$ must do two things simultaneously, a) it must produce probability only on those e that are well formed, and b) it must implement a good translation between a given f and e . When we break the model apart, however, $p(e)$ need only concentrate on scoring high well-formed English strings, while $p(f|e)$ need not do that, and since f is always observed when used in a classification model, it need not have highest probability on well formed French strings. 2) when we use a generative model for classification purposes, the generative model need not be generatively accurate. That is, $p(f|e)$ need not generate good French strings if we were to sample from it since, when we use it, f is always observed. This point was already made above, but we included the quotation from [67] to indicate this point in quite a different context.

8.8.16.0.14 Initial Parameters for Discriminative Models: Lastly, as we have mentioned, since generative models are in some sense easier to train than discriminative models, the parameters resulting from generative models and generative training can be used in discriminative training. This is typically the case when discriminatively training HMM systems [445]. In the area of many-layered neural networks (such as deep models [25]), it wasn't until recently where generative pre-training methods were used to initialize the parameters before they started working. There seems to be a general trend, therefore, that when building a

large discriminative system to start with a generative model, or at least use the parameters from a generative model as an initialization for training a discriminative model.

8.8.16.0.15 Engineering/real world advantages: When a system is large and complicated, it can be much easier to engineer a working generatively trained model than it is to get a large discriminative model working. For rapid system development on huge engineering problems, generative models (since they will work sooner) may be preferred. In general, it can be quite difficult to get a discriminative training procedure to work. Evidence of this is that in the field of speech recognition, it was many years before people were able to fruitfully produce discriminatively trained HMM systems.

8.8.17 Why use discriminative models?

We do not wish for the reader to go away with the impression that we are anti-discriminative models. Indeed, discriminative models and training methods are critical in a number of important applications. Indeed, for classification purposes, it is usually the case that discriminative models, or discriminative training of generative models, performs better, principally because such models are formed specifically with their goal in mind, which is classification.

Indeed, there are a number of reasons, besides resultant classification accuracy, to use discriminative models. In [428], the argument is made that one should train only the simplest thing and not anything more complicated.¹² For example, consider training a linear separator for binary classification where $D_Y = \{0, 1\}$. One strategy would be to learn two Gaussians $p(x|Y = 0)$ and $p(x|Y = 1)$ each of which would have a mean vector and (shared) covariance matrix. If the dimensionality is n , this would lead to $O(n^2)$ parameters to be learned. On the other hand, learning the linear separator directly would involve learning only $O(n)$ parameters, a perhaps much simpler and easier task. Vapnik's argument asks: why spend energy training a model where most of the effort and result of training is not used? Why deal with something more complicated than necessary when the goal is only to produce a classification system? In many cases, as argued in §8.8.16, it is not more complicated than necessary (e.g., variational dependence) but in many other cases it is.

An additional argument in favor of discriminative models, that is a variant of the above, is that it is not necessary to produce $p(y|x)$. Clearly, given $p(y|x)$ it is possible to make minimum error predictions by using $\operatorname{argmax}_y p(y|x)$. However, we don't need $p(y|x)$ to do this, and instead we can use a discriminant function $s(\bar{x}, y)$ such that

$$\operatorname{argmax}_y s(\bar{x}, y) = \operatorname{argmax}_y p(y|x) \quad \forall x \tag{8.478}$$

In fact, it could be that for optimizations other than finding the maximum y , $s(\bar{x}, y)$ and $p(y|x)$ produce entirely different results. For example, $s(\bar{x}, y) \geq 0$ always, if we were to estimate $\hat{p}(y|\bar{x}) \leftarrow s(\bar{x}, y) / \sum_{y'} s(\bar{x}, y)$, it might be that $\hat{p}(y|\bar{x})$ and $p(y|x)$ are different everywhere except at the maximum y . In general, there are three aspects of $p(y|x)$ that we might want:

1. We might want just the maximum, $\operatorname{argmax}_y p(y|x)$
2. We might want just the ranking. That is, we want *just* an ordering y^1, y^2, \dots, y^N such that $|D_Y| = N$ and

$$p(y^1|\bar{x}) \geq p(y^2|\bar{x}) \geq \dots \geq p(y^N|\bar{x}) \tag{8.479}$$

¹²This is really an Occam's razor argument, or as stated by Einstein, "Make things as simple as possible, but not simpler."

but we do not care about the scores. The ranking can be used to produce a N-best list of hypothesis, and this can be useful in a number of machine learning applications (e.g., speech recognition and machine translation).

3. We want the actual scores of the conditional probability $p(y|\bar{x})$ for all $y \in D_Y$.

Clearly, these are ordered in increasing difficulty in that if we have 3, we also have 1, and 2, and if we have 2 we have also 1. Therefore, applications needing only the top ranked have an certain advantage over those applications that need the exact probability scores.

Discriminative models, in general, might provide any of the above. If a generative model is a true model of $p(x, y)$, then it necessarily also may produce the scores $p(y|x)$. But as we have seen, a generative model already need not “generate” perfectly well, which means that a generative model’s expression of $p(x|y)$ need not be accurate. By the same token, a generative model need not produce a perfectly accurate instance of $p(y|x) = p(y, x)/p(x)$ if the goal is only to compute the top ranked hypothesis. Still, there are reasons where the underlying mathematical optimization procedure for producing a discriminative function that is accurate only in its top ranked hypothesis, if that is the goal, is easier to formulate – kernel machines and max-margin training methods more generally are one such example.

A final argument in favor of discriminative models is that they often have theoretically nice generalization bounds. For example, max margin methods decreases the VC dimension in both binary, multiclass, [385] and structured classification tasks [420].

Whether to use generative or discriminative models, therefore, is really a question that one cannot answer without considering the specific applications domain one is in, and we will end this section perhaps unsatisfactorily by leaving it up to the reader to decide which type of model she should use. After all, this is only a GMTK manual.

8.9 Factored and Structured State Spaces: Vectorial HMM

In this section, we will begin to explore “structured” state spaces. We will see that there are certain things that an HMM alone might not not be ideally suited for. We do not yet define fully general structured prediction, which we do in §8.13 after first defining dynamic Bayesian networks (DBNs) in §8.10 and dynamic graphical models (DGMs) in §8.11.

The basic idea of a factored state space is that $p(y_t)$ is no longer a distribution over a monolithic integer random variable Y_t . Instead, Y_t is broken down into components that can be represented by a vector of length M . Hence the random variable Y_t has the equivalence relationship $Y_t \equiv (Y_t^1, Y_t^2, \dots, Y_t^M)$. This means that Y_t is a vector of length M and that the domain size satisfies the relationship:

$$D_{Y_t} = D_{Y_t^1} \times D_{Y_t^2} \times D_{Y_t^3} \cdots \times D_{Y_t^M}, \quad (8.480)$$

meaning that the domain of Y_t is really a Cartesian product of the domains of each of the constituent variables $\{Y_t^i\}_i$. This does not mean that all possible values of $y_t \equiv (y_t^1, y_t^2, \dots, y_t^M)$ have non-zero probability, but the representational capacity of the two are identical. We also say that Y_t is a flattened representation of $(Y_t^1, Y_t^2, \dots, Y_t^M)$. In fact, the process of starting with the vector of M Markov chains $(Y_t^1, Y_t^2, \dots, Y_t^M)$ and flattening it to a single Markov chain Y_t is identical to the process of WFST composition mentioned in §8.4.4.5.

Without making any further assumptions, we can form an HMM not based on the original flattened random variable Y_t but instead based on the vector of random variables $(Y_t^1, Y_t^2, \dots, Y_t^M)$. The resulting joint distribution for this HMM has the form:

$$p(x_{1:T}, y_{1:T}^{1:M}) = p(y_1^{1:M}) \prod_{t=2}^T p(x_t | y_t^{1:M}) p(y_t^{1:M} | y_{t-1}^{1:M}) \quad (8.481)$$

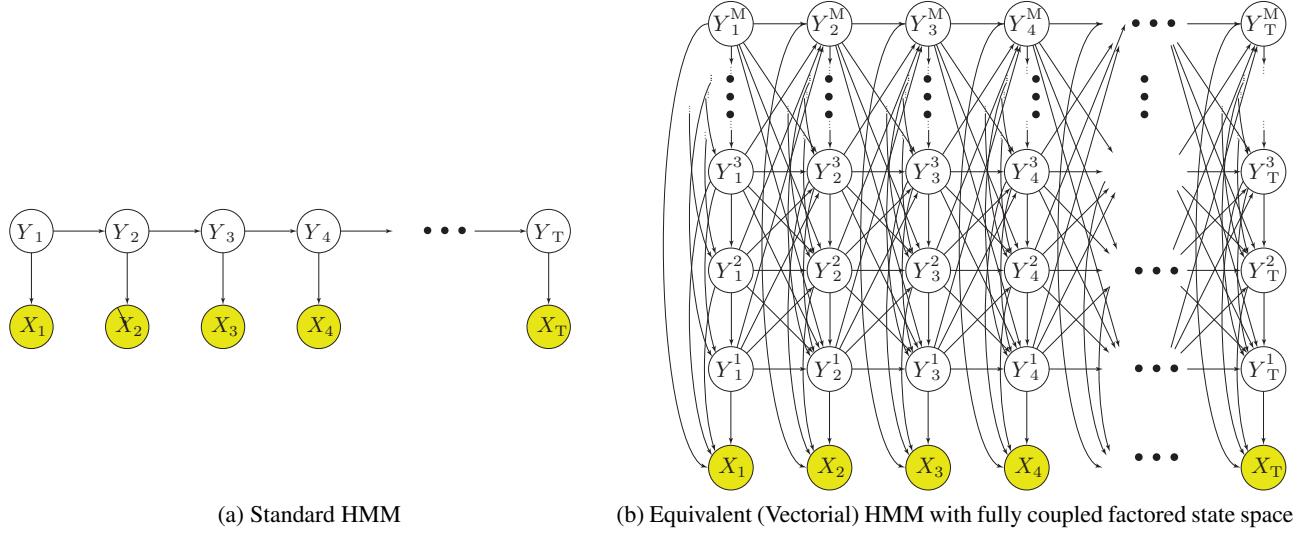


Figure 8.69: HMM with fully coupled factored state space, where $Y_t \equiv (Y_t^1, Y_t^2, \dots, Y_t^M)$.

so we see that there is still a Markov property, namely that the past and future are independent given the present but there is otherwise no assumed property or requirement of the family over how the vector at time t relates to the vector at time $t - 1$. Indeed, if we were to look at the Bayesian network for this family of distributions, we would get Figure 8.69b. Since each state is now a vector, it is reasonable to call this a vectorial HMM (although this is not standard terminology in the literature). When we compare with the original HMM (see either Figure 8.10, or repeated here again for convenience in Figure 8.69a), the factored graph looks much more complicated and potentially more costly computationally. Let $N = |\mathcal{D}_{Y_t}|$ be the number of states in the HMM and $N_i = |\mathcal{D}_{Y_t^i}|$ be the number of states in the i^{th} chain in the factorial representation. Then, the forward-backward algorithm for Equation (8.481) and Figure 8.69b has complexity $O(T \left(\prod_{i=1}^M N_i \right)^2)$. When we consider Equation (8.480), we realize that this is the same complexity as the HMM $O(TN^2)$ so nothing is gained and nothing is lost. Indeed, the graph in Figure 8.69b shows that any of the variables Y_t^i may directly interact, simultaneously with any of the other variables Y_t^j with $j \neq i$, with any of the preceding variables $Y_{t-1}^{i'}$ (for any i') or following variables $Y_{t+1}^{i''}$ (for any i''). By this, we mean that the graphical model allows factors of the form $p(y_t^i | y_t^{i+1}, y_t^{i+2}, \dots, y_t^M, y_{t-1}^{1:M})$ for $i \in \{1, \dots, M\}$.

Given that there is no computational advantage to the factorial representation, what use is there for models such as Figure 8.69b? The main use is representational, in that there are applications where it is useful to think of Y_t not as a monolithic integer, but where Y_t is composed of modular parts that might be equal in importance (e.g., each Y_t^i could represent a dimension in some M dimensional space, so if $M = 2$ or $M = 3$ the hidden variables might represent position in 2D or 3D, which can be useful, say, for robot navigation), or the parts might be inherently hierarchically related (e.g., in speech recognition, they might represent sentences, words, phones, subphones, in proteomics they might represent tertiary structure, secondary structure, and underlying structure of a protein). Many examples of this given in the following pages (and in some sense, as we will see, every DBN example we will see in this text is an example).

An important additional advantage of structured form, besides representational convenience, becomes apparent when we start making further assumptions about the relationship between successive vectors. As there are many edges in Figure 8.69b, there are many possible assumptions that can be made. Without getting into additional details about any particular application, in this section we first consider two common

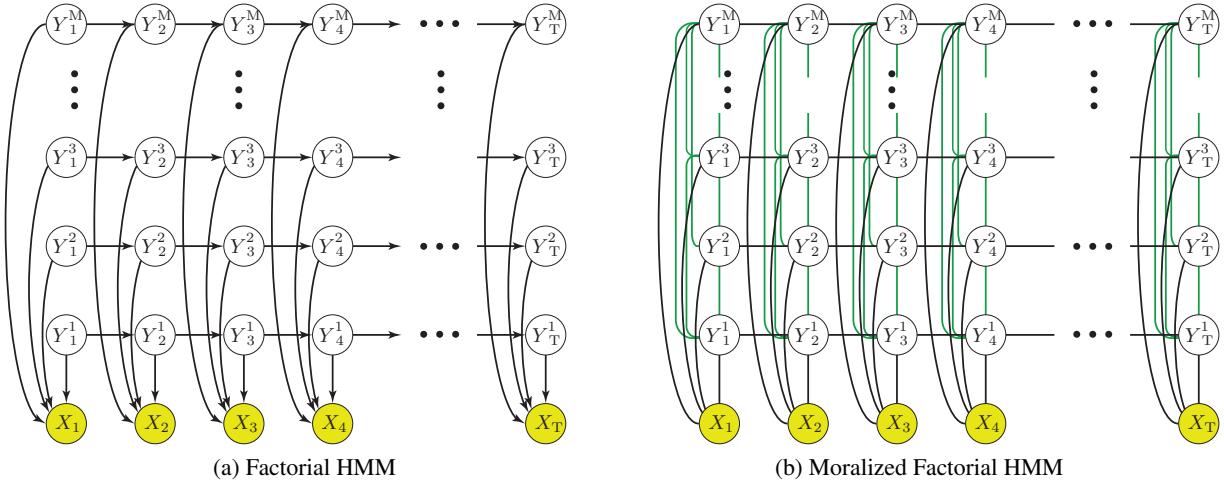


Figure 8.70: (a): A factorial HMM, with M Markov chains running in parallel all simultaneously contributing to the explanation of the observations X . In this model, the Markov chains would be independent but for the observations which, due to the v-structure (i.e., explaining away property, as described in §4.1) induces a dependence upon the chains.(b): A moralized factorial HMM, where the moral edges are in green. Note that we now have a sequence of cliques, one clique at each time frame consisting of $\{Y_t^i\}_{i=1}^M \cup \{X_t\}$, and persistent edges between successive Y_t^i s. This means that exact inference cost is at least $\Omega(N_f^M)$.

structural assumptions.

8.9.1 Factorial HMMs

The first example we consider is the *factorial HMM* (FHMM) [170], shown in Figure 8.70a. The Factorial HMM has all the edges that exist in Figure 8.69b removed except for those that are between two corresponding variables at successive time frames $Y_{t-1}^i \rightarrow Y_t^i$ (what are called *persistent edges*), and for those that lead from each state to the observation $Y_t^i \rightarrow X_t$. The equation for the factorial HMM is greatly simplified relative to

$$p_{\text{FHMM}}(x_{1:T}, y_{1:T}^{1:M}) = p(x_1|y_1^{1:M}) \prod_{m=1}^M p(y_1^m) \prod_{t=2}^T p(x_t|y_t^{1:M}) \prod_{m=1}^M p(y_t^m|y_{t-1}^m) \quad (8.482)$$

From Figure 8.70a or Equation (8.482), we see that

$$\sum_{x_{1:T}} p_{\text{FHMM}}(x_{1:T}, y_{1:T}^{1:M}) = \prod_{m=1}^M \prod_t p(y_t^m|y_{t-1}^m) \quad (8.483)$$

meaning marginalizing over the hidden variables, we are left with a set of M independent Markov chains, meaning that $Y_{1:T}^i \perp\!\!\!\perp Y_{1:T}^j$ for $i \neq j$.

An important thing to note about the factorial HMM, however, is that different elements of the vector $(Y_t^1, Y_t^2, \dots, Y_t^M)$ at a given time t are in fact not independent when X_t is observed, meaning that it is *not* the case that $Y_{1:T}^i \perp\!\!\!\perp Y_{1:T}^j | X_{1:T}$ for $i \neq j$. This is because of the v-structure $(Y_t^1, Y_t^2, \dots, Y_t^M) \rightarrow X_t$ (i.e., explaining away property, as described in §4.1, where common parents are not independent given the child random variable). This means that, although unconditionally independent, when FHMMs are used in practice, the M Markov chains can become quite coupled.

FHMMs are useful for contexts where there are multiple independent chains of events that go into explaining the observables. An early example of this was given in [429, 430], where the goal was to explain a speech signal in a noisy environment with two Markov chains, one that represented the words being spoken, and another that represented an explanation of the noise source. The assumption is that the speech and the noise are independent marginally (which isn't necessarily true in practice, due to effects like *Lombard speech* [257], where speech can change due to noisy environments), but conditioned on the acoustic environment the two can become dependent. More generally, for sources that consist of multiple independent causes but that are “sensed” by a single sensor (which could be audio, video, etc.), an FHMM seems appropriate. The application considered in [170] was one of music, where the goal was to see the degree to which it was possible to learn the temporal independent components in a set of Bach chorals.

Now, as is often the case with graphical models, when we remove edges, we place more factorization requirements on the family of models and in doing so often make possible more efficient inference methods. The FHMM is no exception as, relative to Figure 8.69b, exact inference cost does indeed improve. To see this, we consider the moralized graph in Figure 8.71a. This graph also does not show the observations X_t since, once the parents of X_t are coupled in the moralization process, including X_t does not change the state space or computational complexity.

For the sake of this discussion, let $N_f = |\mathcal{D}_{Y_t^i}|$ be the common state space of each of the hidden Markov chains in the factorial HMM. Hence, we have that $N = (N_f)^M$. The flattened HMM complexity, again, would be $O(TN^2) = O(TN_f^{2M})$. A smart exact inference procedure for a factorial HMM can be much faster than this.

We eliminate the variables slice by slice, using the elimination order $Y_1^M, Y_1^{M-1}, \dots, Y_1^1$ which then produces a subgraph that looks identical to the original one but where the first frame has been completely eliminated. This is shown in Figure 8.71a, where $M = 4$. When we eliminate Y_1^4 in Figure 8.71a the set of neighbors has size four, so the elimination clique is of size five. Next, eliminating Y_1^3 in Figure 8.71b leads again to an elimination clique of size five. The same size elimination clique arises when we eliminate Y_1^2 in Figure 8.71c, and Y_1^1 in Figure 8.71d. After the first frame is completely eliminated in the above order, we get Figure 8.71e which looks like a time shifted version of Figure 8.71a and hence if we next eliminate the variables using the elimination order $Y_2^M, Y_2^{M-1}, \dots, Y_2^1$, the same pattern of fill-in edges and elimination cliques result. After all vertices are eliminated in the graph, the reconstituted graph, showing all fill-in edges, is shown in Figure 8.71f. We see that the largest elimination clique encountered is thus of size five and so the overall complexity is $O(TN^5)$.

More generally, we see that at each time slice, we encounter M elimination cliques, each of size $M + 1$. This means that the elimination schedule above for the factorial HMM has complexity $O(TM N_f^{M+1})$. When we compare this to the flattened HMM complexity of $O(TN_f^{2M})$, we see that the factorial HMM can have significantly improved computational properties.

We claim that, for T long enough (namely, $T \geq 3$), the above elimination order cannot be improved upon when one wishes to obtain optimal exact inference. From the above, it can be seen that the treewidth of the model is at least M since we were able to triangulate the model with cliques of size $M + 1$. Indeed, the variables at any given time slice are complete and put a lower bound of $M - 1$ on the treewidth. Eliminating the first (or any) of these variables on the left produces a clique of size $M + 1$, hence guaranteeing the optimality of the above elimination order.

Theorem 137. *The factorial HMM has an optimal computational complexity of $O(TM N_f^{M+1})$.*

This means that there is no benefit to eliminating variables starting from the middle of the model (i.e., not frame 1 or T)? For example, starting from Figure 8.71a, if we were to eliminate any of the variables Y_t^i for $1 < t < T$, we'd immediately generate a clique of size six. Hence it is optimal to eliminate all the variables on the ends (respectively $t = 1$ or $t = T$) first before proceeding (respectively forwards or backwards). The other question one might ask is if there might be benefit to eliminate variables in frame $t + 1$ before finished

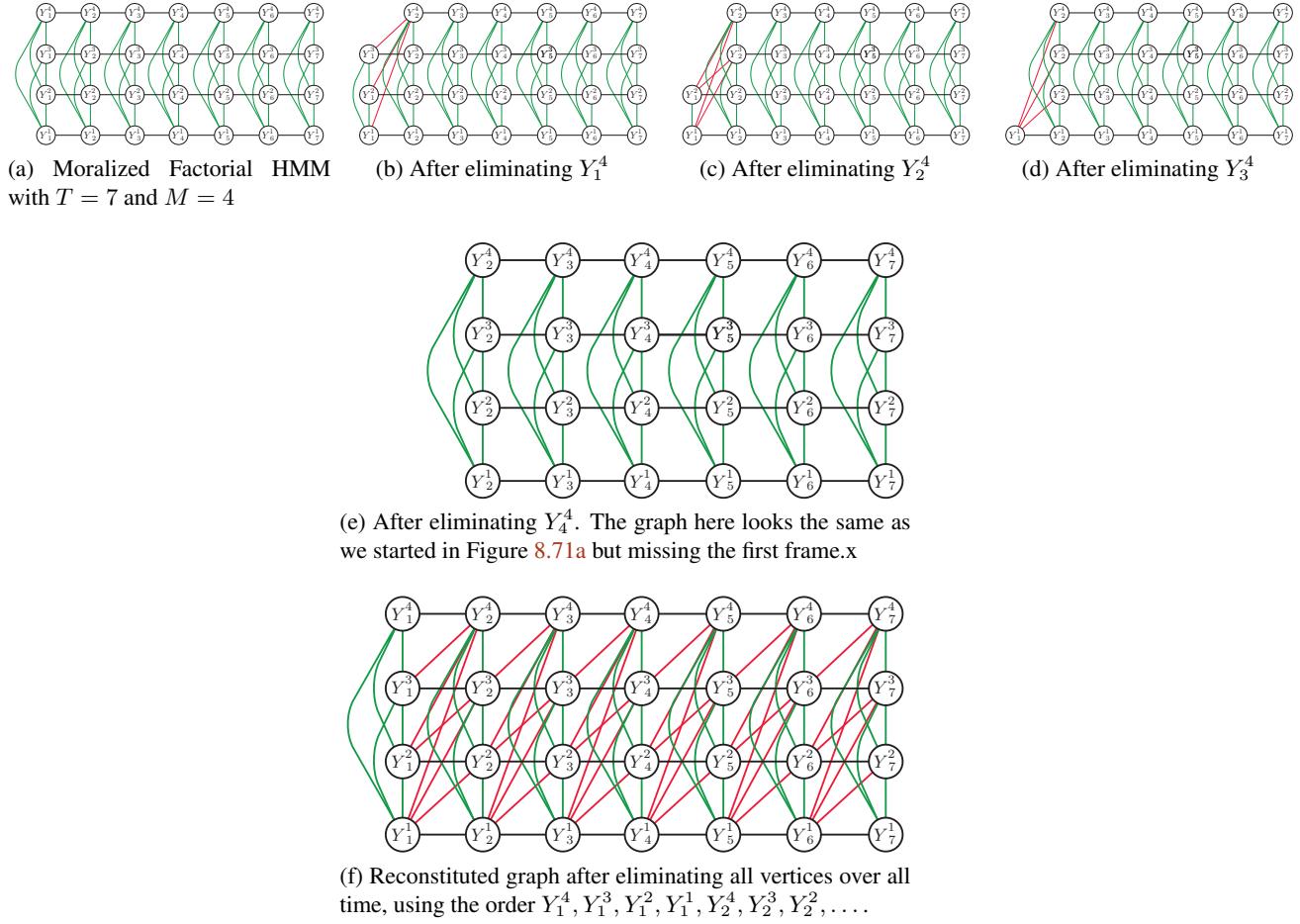


Figure 8.71: The moralized factorial HMM and various steps of the elimination process, shown without the observations X_t since that does not change the complexity of running the elimination procedure. Black edges correspond to the original edges in the graph; Green edges are due to the moralization process; and red edges are fill-in edges due to the vertex elimination process. From these figures, we can deduce that the complexity of inference in a factorial HMM is $O(TMN_f^{M+1})$ where $N_f = |\mathcal{D}_{Y_t^i}|$, $\forall i$ is the common state space of the factored Markov chains.

with frame t . Since the above elimination order is optimal, there is of course no theoretical benefit. On the other hand, the optimal elimination order (as is typical) is not unique. For example, we could eliminate the variables Y_1^i according to any permutation of $\{1, 2, \dots, M\}$ first and that would still be optimal.

Exercise 138. Show that we can eliminate the variables Y_1^i according to any permutation of $\{1, 2, \dots, M\}$ first and this is optimal.

We discuss Factorial HMMs in the context of constrained elimination again in §13.3.1.1. In some cases, we will see that constrained elimination cannot produce the optimal triangulation.

8.9.2 Hierarchical HMMs

A second common structural assumption that is made about the vector $(Y_t^1, Y_t^2, \dots, Y_t^M)$ is known as the *hierarchical HMM*. In fact, the hierarchical HMM is so common that it is often the case that when the HMM

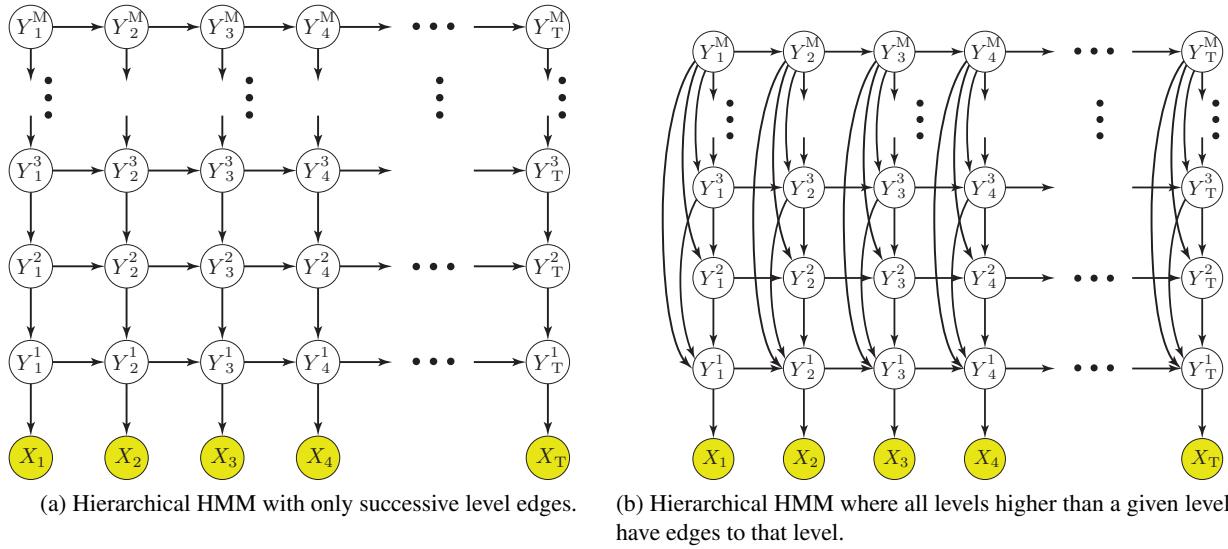


Figure 8.72: (a): A hierarchical HMM, where (b): A hierarchical HMM, where multiple Markov chains are connected successively to each other but only the lowest-level Markov chain is connected to the observations.

is used, it is really a hierarchical HMM in flattened form rather than a standard HMM. Unlike the factorial HMM, where each Y_t^i for different i has the same “level”, granularity, or importance, in the hierarchical HMM, the different $\{Y_t^i\}_i$ are organized hierarchically. That is, the different Y_t^i might be operating at different levels in a hierarchical decomposition of some physical process. They often evolve at different rates as well, in what could be called a multi-rate model. A classic example is in speech recognition, where Y_t^M might represent words, Y_t^{M-1} might represent phonemes, Y_t^{M-2} might represent subphones, and Y_t^{M-3} could represent states or some other constituent parts of subphones (e.g., typically beginning, middle, and end). But there are many examples in NLP and genomics, where the goal is for the higher level variables to represent higher-level concepts or clusters each of which are comprised of parts consisting of sequences of the immediately lower level variables.

There is no one single model that is a hierarchical HMM (HHMM) — rather, the HMM relaxes the constraints made by the fully general model (Figure 8.69b) in that higher level variables generally are connected (and may be said to directly influence) lower level variables. One such example is where there is a strict successive set of Markov chains, one at each level, where the chain at level i connects only to the chain at level $i - 1$, as shown in Figure 8.72a. In this form of model, the influence higher levels have on lower levels is only indirect via intermediary levels.

A second, more general, form of hierarchical HMM is shown in Figure 8.72b, where the chain at level i has edges to all lower levels. Note that this model can also represent a form of temporal series of decision trees, where a question is asked at each level — for $i = M \dots 2$, a question is asked at level i and then the questions asked all lower levels depend on the answer at level i . This is made possible by the fact that Y_t^i has parents at time t at all higher levels (as well as the persistent edge to Y_{t-1}^i).

At this point, we now have the tools to start defining many other models in the above fashion, including models as input-output HMMs, hidden Markov decision trees, and so on. Note that it is possible to have state spaces even more general than factored state spaces, something often called structured state spaces and we briefly touch on this in § 8.13. But before doing so, and before discussing additional possible assumptions that can be made beyond Figure 8.69b, we define useful classes of models within which these live, which this chapter is really about.

8.10 Dynamic Bayesian Networks (DBNs)

All of the models described so far in §8.9 are examples of DBNs.¹³ We've in fact already been using DBNs informally to define the generalizations of HMMs in that section. Before we go on to describe still further generalizations of HMMs in §TODO, in this section we informally discuss and then more formally define DBNs.

A dynamic Bayesian network (DBN) [107] is a template Bayesian network (cf. §8.1) that when expanded (i.e., “instantiated”) is a Bayesian network with certain edges pointing in the direction of “time” (or along some common spatial or axial dimension). The temporal edges exist between variables that represent different time points, while other edges point between variables meant to represent the same time point.

Often, an instantiated DBN must correspond to a given known number, T , of time steps. In such case, the template is expanded so that it fits the given T . In many other cases, however, T is unknown and it might not even be known if there is a given fixed T as the model can expand arbitrarily for as long as necessary — this is common in online and streaming applications. For example, in real-time speech recognition system (e.g., dictation or certain smart phone applications), the final length T is unknown at the point that the words start appearing on a screen, and this occurs before the speaker is done speaking. In other cases, the application demands that the model is able to make a prediction at every time step t , and we can't wait until some ultimate final T . Such applications include financial time series prediction (e.g., a “buy” or a “sell” decision), or in human activity recognition (e.g., we need to determine in real time what a person is doing for health or safety monitoring, and if we wait too long, an incorrect or dangerous activity would be only detected but not prevented).

When we don't know T , we start by expanding the model a bit, and then whenever it is determined that the model needs to be extended further, an additional “chunk” of model is append on at the end of the existing model. In this case, moreover, we can often remove the earlier portions of the model (and free up any resources used) so that memory requirements for the model not only do not expand unboundedly but they remain constant.

In either case (fixed T , or unbounded length streaming) there is a common core structure or chunk that is repeated as needed, and the core structure is one of the things that specified by (or determined indirectly via) the template. In GMTK, this core structure is called the “chunk” and is referred to as C .

A DBN template determines a “rolled up” structure — when the actual time length T becomes known, or whenever we wish to extend the model by an additional chunk, the template may be expanded to any desired length. To produce an unrolled network, a portion of the template structure is repeated, often one repetition is used for each time “frame” or time slice of data that is available. Sometimes, however, one chunk of unrolling corresponds to multiple time frames of data (see §8.11.1 for this case).

When T is known for each sequential sample, it is most common for these lengths go vary from sample to sample. We might say have a training data set $\mathcal{D} = \left\{ (x_{1:T_i}^{(i)}, y_{1:T_i}^{(i)}) \right\}_{i=1}^D$ consisting of D pairs of sequences, input sequences $x_{1:T_i}^{(i)}$ and output sequences $y_{1:T_i}^{(i)}$ where the i^{th} sequence has length T_i . For example, in speech recognition, T_i (the utterance or segment length) will typically be different for each training and testing utterance. It is worth pointing out here that most statistical models make some form of i.i.d. assumption, as mentioned in §8.2. I.e., we might have a set of samples $\{x_i\}_i$ where $x_i \sim p(x)$ for some distribution, and the samples are drawn i.i.d.. With dynamic models, there is still an i.i.d. assumption but the samples are sequences which might be variable length. Each sequence $x_{1:T_i}^{(i)}$ is (presumed to be) drawn independently from some variable length distribution $p(x_{1:T}, T)$ where T is itself a random variable. It is not assumed, however, that the set of individual frames within a sequence need be independent nor any of

¹³In recent years, the term DBN has also come to mean “deep belief network” [25]. It is this authors belief that deep belief networks should be called deep neural networks (DNNs) in order to reserve the minimally overloaded and much earlier defined term DBN for “dynamic bayesian networks.” We note that GMTK supports deep neural networks as input to a random variable.

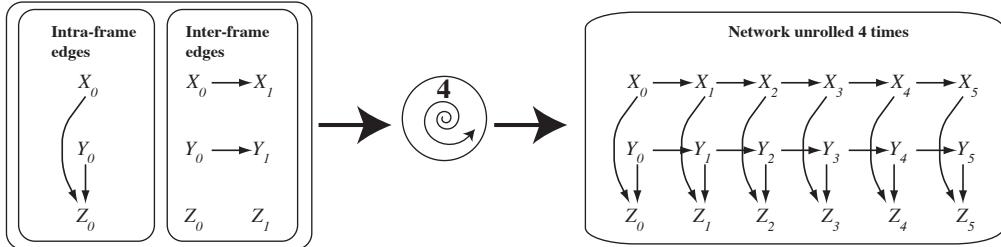


Figure 8.74: A “rolled” and unrolled network in a typical dynamic Bayesian network. On the left, we see two networks that together comprise the DBN template (cf. §8.1). The template consists of two time frames or time slices. The first part of the template shows the intra-frame edges and consists of only one time slice — these are the edges that exist between variables that live within the same slice. The other part of the template shows the inter-frame edges, i.e., the edges that go between two successive slices. On the right is the network that has been unrolled 4 (four) times. Note that we use the convention that a network can be unrolled 0 (zero) times; in this case, the resulting network is identical to what would be obtained by applying the intra-frame edges to the inter-frame network. In the figure, therefore, a network unrolled zero times would consist only of two frames. Incidentally, this model corresponds to a factorial HMM with two chains X_t and Y_t similar to Figure 8.70a with $M = 2$.

the random variables therein.

With many dynamic Bayesian networks (DBNs) [107, 312, 373, 249] a template is specified by listing a collection of N variables that are to exist at each time frame. Along with these variables, a set of intra-frame and inter-frame dependencies used to unroll the template are also given. When the template is unrolled, the set of variables are repeated as needed. When unrolling T times, this leads to a network with a total of TN variables. The intra-frame edges in the template are repeated along with the set of variables, essentially “rubber-stamping” the variables and their edges. Hence, the intra-frame edges each occur a total of T times. The inter-frame edges are also repeated as needed, and are used to connect pairs of vertices in neighboring frames. Thus, the inter-frame edges in the unrolled model each occur $T - 1$ times. This scenario is depicted in Figure 8.74 and Figure 8.73.

Hence, DBNs are simply Bayesian networks with a repeated “template” structure over time. Other than this periodicity, however, DBNs once instantiated to a particular length have exactly the same semantics as Bayesian networks. Specifically, a DBN instantiated to length T is a directed acyclic graph $\mathcal{G} = (V, E) = (\bigcup_{t=1}^T V_t, E_T \cup \bigcup_{t=1}^{T-1} E_t \cup E_t^\rightarrow)$ with vertex set V , and edge set E comprising pairs of vertices. If $uv \in E$ for $u, v \in V$, then uv is an edge of \mathcal{G} . The sets V_t are the vertices at time slice t , E_t are the *intra-slice edges* between vertices in V_t , and E_t^\rightarrow are the *inter-slice edges* between vertices in V_t and V_{t+1} .

As mentioned above, as DBN may arise only via instantiation from a rolled up template. So instantiated, a DBN has vertices that are repeated in each slice, has the same set of intra-slice edges among corresponding vertices, and the same pattern of inter-slice edges between vertices of adjacent slices. Therefore, a DBN does not have as much flexibility as is allowed by the graph \mathcal{G} and its corresponding family of distributions \mathcal{G} . An instantiated DBN instance actually lies within a subset of the family of models indicated by \mathcal{G} . There are several reasons for this. The first is that the set of random variables V_t is isomorphic to V_τ for any $\tau \neq t$ — i.e., for every vertex $v_t \in V_t$ there is a corresponding $v_\tau \in V_\tau$, and vice versa. The random variables

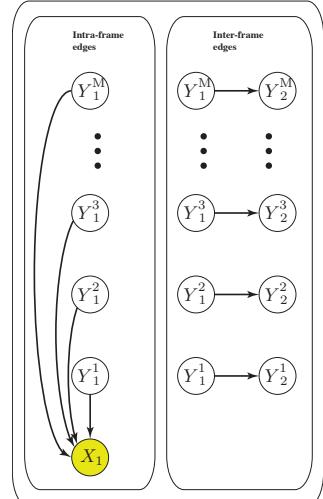


Figure 8.73: The DBN template for the factorial HMM shown in Figure 8.70a.

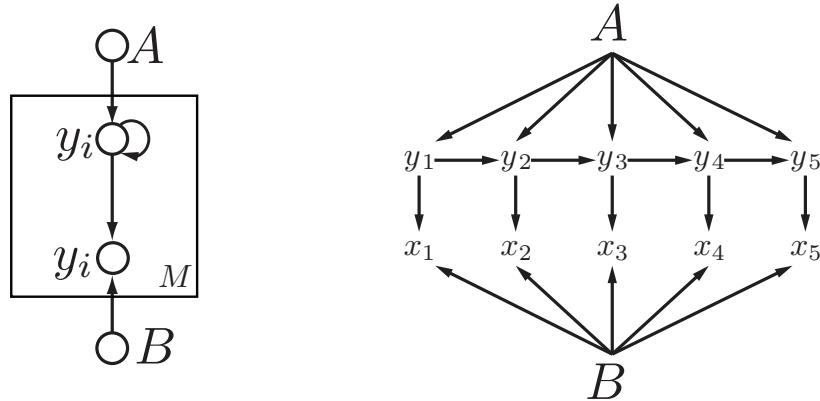


Figure 8.75: A time-homogeneous Markov chain viewed in a Bayesian fashion. Here, the values of the transition matrix M have two distributions $p(y_t|y_{t-1}, \Theta_{y_t|y_{t-1}})$ and $p(x_t|y_t, \Theta_{x_t|y_t})$. If $\Theta_{y_t|y_{t-1}}$ and $\Theta_{x_t|y_t}$ are constant random variables then this represents parameter sharing, where the parameters of each of the factors $p(y_t|y_{t-1})$ and $p(x_t|y_t)$ are identical. When $\Theta_{y_t|y_{t-1}}$ and $\Theta_{x_t|y_t}$ are random variables themselves, then we have a random HMM.

have the same “names” but different time indices. A similar isomorphism exists between E_t and E_τ , and between E_t^\rightarrow and E_τ^\rightarrow . This is shown in Figure 8.74.

A second restriction on DBN instantiations of \mathcal{G} comes in the form of a constraint on the parameters of the random variables, constituting a form of time-homogeneity property (cf. §8.3). As in any BN, the collection of edges pointing into a vertex corresponds to a conditional probability function (CPF). In a DBN, the CPF of a vertex is shared (or tied) with the CPF of all other vertices that have come from the same underlying vertex in the DBN template. In other words, given a random variable X_t having parents X_{π_t} , then $p(X_t = x|X_{\pi_t} = x_\pi) = p(X_\tau = x|X_{\pi_\tau} = x_\pi)$ for all t, τ and for all scalar values x and vector values x_π . Indeed, if one imparts a Bayesian interpretation on the model, one can draw an expanded DBN using explicit variables for parameters, as shown in Figure 8.75. In a Bayesian setting, the parameters would be true random variables (i.e., A and B would be random variables), but here we mean only to suggest that they are constant random variables (e.g., $p(A = \bar{A}) = 1$ for some \bar{A}), to represent shared parameters.

With this mechanism, as with any time-homogeneous model, it is possible to represent a DBN of unbounded length, but with only a finite description length and a finite number of parameters. Indeed, this is a property that characterizes all dynamic graphical models as mentioned below.

It is important to realize, however, that while the two sets of random variables V_t and V_τ are isomorphic, they are distinct random variables and can have very different conditional probability distributions. That is, even though corresponding random variables over different slices share the same parameters, this does **not** mean that the random variables have the same probability distributions. For example, even with an HMM, where $y_{1:T}$ is hidden and $\bar{x}_{1:T}$ is observed, even though y_t is “isomorphic” with y_τ , it is usually not the case that $p(y_t|\bar{x}_{1:T}) \neq p(y_\tau|\bar{x}_{1:T})$ for $t \neq \tau$. Thus, tied parameters do not mean tied (or even stationary, cf. §8.3) distributions.

It is well known that the hidden Markov model (HMM) is one type of DBN [398]. Even given its success and flexibility, however, the HMM is only one model within the enormous family of statistical techniques representable by DBNs. Like an HMM, a DBN makes a temporal Markov assumption, meaning that the future is independent of the past given the present. In fact, it is true that many (but not all, see Section 9.1.7) DBNs can be “flattened” into a corresponding HMM, but residing in the DBN framework has several advantages. First, in DBN form, there can be exploitable computational advantages since the DBN explicitly represents factorization properties, and factorization can be key to tractable probabilistic inference

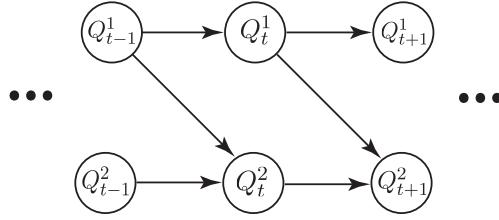


Figure 8.76: Simple two-stream Markov chain

[219]. These factorizations, however, are lost when the model is flattened. An example of the computational savings lost when flattening was shown in the case of a factorial HMM in §8.9.1.

Secondly, the factorization specified by a DBN implies that there are additional parameter constraints that the model must obey. For example, consider Figure 8.76 which shows a two-Markov chain DBN with chains (Q_t^1, Q_t^2) . A flattened HMM would have one chain $R_t \equiv (Q_t^1, Q_t^2)$ with transition probabilities set as follows:

$$p(R_t = r_t | R_{t-1} = r_{t-1}) = p(Q_t^1 = q_t^1, Q_t^2 = q_t^2 | Q_{t-1}^1 = q_{t-1}^1, Q_{t-1}^2 = q_{t-1}^2)$$

where $r_t \equiv (q_t^1, q_t^2)$ is the joint HMM state space (for example, a bijection can be formed using $r_t = |\mathcal{D}_{Q_t^2}|q_t^1 + q_t^2$). Such flattening, however, ignores the factorization constraint expressed by the graph, which can be written as:

$$\begin{aligned} & p(Q_t^1 = q_t^1, Q_t^2 = q_t^2 | Q_{t-1}^1 = q_{t-1}^1, Q_{t-1}^2 = q_{t-1}^2) \\ &= p(Q_t^1 = q_t^1 | Q_{t-1}^1 = q_{t-1}^1) p(Q_t^2 = q_t^2 | Q_{t-1}^1 = q_{t-1}^1, Q_{t-1}^2 = q_{t-1}^2) \end{aligned}$$

In other words, not all possible $p(r_t | r_{t-1})$ CPFs are allowed given the graph due to its conditional independence property — in the above, if no assumptions were made, both Q_{t-1}^1 and Q_{t-1}^2 could directly interact with Q_t^1 . This means that $p(r_t | r_{t-1})$ really constitutes a 4-dimensional tensor, but due to the factorization constraints it is a low-rank tensor.

Of course the HMM can represent a distribution designed under these constraints. But when training parameters, we must find the optimal solution within the parameter space subject to these constraints. For example, with maximum likelihood training, we could train an objective of the form $\text{argmax}_\theta \log p_\theta \prod_t p(r_t | r_{t-1})$ subject to the CPT $p(r_t | r_{t-1})$ being an appropriately low-rank tensor. It is during training (when the amount of training data might be limited) that one wants to reduce the degree of parameter freedom (via a set of constraints on the model). Factorization is one method of reduced parameter freedom, and can hence be seen as a form of parameter regularization. Since a DBN (or any graphical model) naturally expresses factorization, it is an ideal candidate to train model parameters — doing so is identical to constrained optimization.

A third advantage of DBNs is that they convey structural information about the underlying problem. Such structure might represent anything from the result of data-mining process [31] on the training data to dependencies over high-level knowledge sources, or both. In either case, information about a domain is visually and intuitively portrayed. We see many such structures in Chapters 9 through 12.

Loosely speaking, DBN probabilistic inference (a generalization of the Baum-Welch procedure for HMMs [347]) has a computational cost upper bound (i.e., it is possible to show that this is the worst case) equal to very roughly the joint state space (the number of combined variable assignments that can occur with non-zero probability) of all the variables in two time slices of the graph [34, 312, 462, 463, 468] multiplied by the total number of time slices T . Therefore, one must be careful when adding variables to a DBN that the cost does not become prohibitive. While this section does not get into the specifics of DBN inference, it should be known that this cost often strongly depends on the DBN triangulation method used [34, 13]. In

other words, adding variables will often, but not necessarily, cause a significant increase in computational cost. We discuss these issues at length in Chapter 13.

8.11 Dynamic Graphical Models

A dynamic graphical model (DGM) is any graphical model that can be expanded over time. As there are many types of graphical model (such as Bayesian networks, factor graphs, undirected Markov random fields, etc.), there are just as many types of dynamic graphical model. Any static graphical model can be extended in type if there is a notion of a template and a set of rules for expansion, either to a fixed length T , or by an additional section for online or streaming applications. In either case, such models are tailored specifically to sequence data.

In this section, we define dynamic graphical models. We concentrate really on two types of such models, namely generalized dynamic Bayesian networks, giving a generalized definition from the one given in § 8.10, and also dynamic factor graphs, where we can have arbitrary factors over a set of random variables.

In all cases, A DGM consists of a template and rules for validity and expansion, but the expansion is along one-dimension — this makes it possible to determine computational properties and develop inference procedures, for any expansion, based only on the template. Unlike plate models (cf. §8.1), the DGM expansion rules no longer require edges between only corresponding variables in successively expanded chunks (non-persistent edges over time are possible).

The template and expansion rules also mean that for any expansion, the model still has a finite-length description. That is the number of parameters in the model may be on the order of the size only of the template, rather than the the length T . This similar to a time homogeneity assumption in a Markov chain (cf. § 8.3). The number of parameters not increasing with T means that different random variables must be sharing the same parameters. That is, suppose that t_1 and t_2 are two distinct time points and X_{t_1} and X_{t_2} are two different random variables associated respectively with those time points. While the random variables are different, it may be that the parameters governing the distribution of X_{t_1} and its immediately neighboring variables in an MRF (i.e., its Markov blanket) are constrained to be the same as the parameters governing X_{t_2} and its blanket. During any model training procedure, therefore, the training algorithm becomes one of constrained optimization, i.e., we must optimize an objective function (such as the likelihood of a data set) subject to the constraint that the parameters for different random variables are identical. Parameter sharing during learning can be seen as a form of regularization [58] — for example, one can imagine a Bayesian prior distribution over the parameters that provides zero probability for parameter values except those where certain parameters are equal to each other (see Figure 8.75). Parameter sharing yields the same effect. We note that parameter sharing is used in almost all template models.

As mentioned in §8.3 Markov chains can be either time-homogeneous or time-inhomogeneous and so can DGMs. In the latter case, the parameters are a function of $0 \leq t \leq T$ so it can be the case that the number of parameters is no longer a constant with T . There are several ways of expressing this, however. In a pure time-inhomogeneous model, each time t has its own set of parameters θ_t leading to a model with an unbounded description length. For example, in an HMM with states Y_t , a time-homogeneous model might be represented as $\Pr_t(Y_t = i|Y_{t-1} = j) = f(t, i, j)$ meaning that the probabilities might change at each time step. On the other hand, if θ_t is a parameter vector at time t , then it could be that there are some underlying finite set of parameters ν that are selected from for a given t as in $\theta_t(\nu)$. The model therefore can still have t -dependent parameters but also have a finite description length based on ν , and hence would not constitute a time-homogeneous model. A construct similar to this was used in [447].

A general dynamic graphical model consists of a graph $G = (V, E)$ template, a set of rules, and an integer expansion parameter $T \geq 0$. The family associated with a DGM expanded to length T may be denoted $\mathcal{F}(G, \mathcal{M}, T)$. Any $p \in \mathcal{F}(G, \mathcal{M}, T)$ is such that it must obey all the Markov properties specified by

expanding G by T using rules \mathcal{M} , and p must be a distribution over a set of random variables whose size is some function of T . The complete family of distributions associated with a DGM considers all T , namely $\bigcup_{T>0} \mathcal{F}(G, \mathcal{M}, T)$. In general, for $T \neq T'$, $\mathcal{F}(G, ,T) \cap \mathcal{F}(G, \mathcal{M}, T') = \emptyset$.

Should DGMs be considered finite state models? Since they expand for all T , if we consider all T the family $\bigcup_{T>0} \mathcal{F}(G, \mathcal{M}, T)$ is not finite state. However, we do consider DGMs to be finite state in that they are finite-state per unit time. For example, in an HMM each of the Y_t variables may have a finite set of values. For a DGM, the hidden variables corresponding to a fixed time range may have only a finite number of values. There are also “Infinite HMMs” and non-parametric Bayesian models. These terms corresponds to the learning process, where during learning there is a prior and learning algorithms that may expand the state space unboundedly (at a cost) if necessary . Once such models are learnt, however, they are once again finite state in the sense described above.

DGMs generalize dynamic Bayesian networks (DBNs) [107], dynamic (or temporal) Markov random fields (DMRFs) (both of which themselves generalize HMMs [138] and factorial HMMs [313, 170]), Boltzmann chains [379], sequential segmental models [173, 330], dynamic conditional random fields (CRFs) [256] and segmental conditional random fields. If the template, and all expanded graphs, are Bayesian networks, then the DGM corresponds to a DBN. If the template and its expansions always produce a MRF, then we have a DMRF. If the expanded template corresponds to a conditional distribution, the DGM corresponds to a CRF or its generalizations. Any type of static graphical model, in fact, including factor graphs [254] and chain graphs [258] has a dynamic counterpart. In this article we concentrate on only DBNs and DMRFs, but the inference methods apply to any type of dynamic graphical model.

In practice, DGMs have a quite different general shape than their static cohorts. Since DGMs are sequential, the expanded graphs are typically much wider than higher since T can be arbitrarily large. For certain biological signals, the height of the DGM becomes essentially negligible. It is clear, therefore, that probabilistic inference should follow a form similar to the standard forward-backward inference algorithm in hidden Markov models (HMMs). However, unlike with an HMM, the state space is structured, and this structure offers some unique opportunities for performing fast inference not available to the HMM. Moreover, in many domains (such as speech recognition) the state space at each time point can be quite large (in the tens of millions). The best approach to inference would be able to deal with both of these extremes (T large, and large state space) and everything in between.

Moreover, as mentioned above, our claim is that we can deduce an upper bound estimate on the complexity of the model for any T just using the template and without needing to examine each such expansion. That is, our goal is to produce an inference algorithm for any $p \in \bigcup_{T>0} \mathcal{F}(G, \mathcal{M}, T)$ using just the information in $\mathcal{F}(G, R, \tau)$ for some fixed value τ . Doing so means that the cost in deducing such an inference algorithm is amortized over the use of (G, R) .

There are many inference problems in DGMs, and they correspond to the generalizations of the HMM probabilistic queries given in §8.4.11.5. For example, one might wish to perform DGM filtering, DGM smoothing, DGM fixed-lag smoothing, or DGM prediction/extrapolation.

8.11.1 Generalized Dynamic Bayesian Network Templates: Prologue, Chunk, and Epilogue

As mentioned above, DGM factors can be either all directed and acyclic meaning that we are generalizing DBNs, but can also correspond to arbitrary factors, in which case we need to specify a normalizer Z (dynamic factor graphs). In this section, we define templates for generalized DBNs and DGMs. Many of the examples will be given in terms of general DBNs (whose instantiation are all Bayesian networks), but for any directed BN factor of the form $p(a|b, c, d)$ we can instead consider this to be an arbitrary factor $\phi(a, b, c, d)$ (corresponding to a Bayesian network moralization step) that resides wherever the variable a resides at, where a, b, c, d are timed random variables. We will point out this distinction when necessary, but will for now mostly discuss things in terms of DBNs.

We generalize the notion of a DBN template so that there are now three sections: a certain number of distinct frames occur at the beginning (called the “*prologue*”) and also at the end (called the “*epilogue*”) of any expanded instantiation. We also extend the DBN template so that the section from the template that is repeatedly unrolled to instantiate a full model is not just a single frame and the inter-frame connections between the current and the previous frame — instead, this repeated section will be a “chunk” of vertices that themselves could span a fixed range of time-frames. We call this the generalized DBN template or, as we will call it in this document, just the DBN template, and the three sections are called the prologue, chunk, and epilogue.

We first describe the generalized DBN template informally, then give some examples, and then more precisely define what generalized DBN templates are allowed to be.

A generalized DBN template consists of a graph $G = (V, E) = (\mathcal{G}^p, \mathcal{G}^c, \mathcal{G}^e)$, that is partitioned into three *blocks*: the prologue block $\mathcal{G}^p = (V_p, E_p)$, the chunk block $\mathcal{G}^c = (V_c, E_c)$ (which is to be repeated in time), and the epilogue block $\mathcal{G}^e = (V_e, E_e)$ (or just the prologue, chunk, and epilogue for short). Note that each of the prologue, chunk, and epilogue are themselves graphs. Also, as we will see, in order for there to be temporal dependence between blocks, at least one of the prologue or epilogue must be non-empty — if both are non-empty, the expanded model will consist of a set of independent chunks. We call this a partition into blocks for now, as for the moment this constitutes a real partition (there is no intersection between the blocks) but we ultimately use the term “section” since after processing, the three sections will have intersection. We use a “partition into blocks” without confusion, meaning there will be no intersection in this case, but when we use the term “section” we allow for there to be some intersection between sections.

In general, any of the blocks may correspond to any number of traditional “frames”, “time slices”, or just “slices” — e.g., a chunk need not consist of only one time slice, but can be any fixed number of time slices long. This relaxation allows for more flexible specification of models and more general inference schemes as we see.

The following is an informal definition of a generalized DBN.

Definition 139 (Definition of a DBN). *There are three sections of a graph, $\mathcal{G}^p = (V_p, E_p)$ called the prologue, $\mathcal{G}^c = (V_c, E_c)$ called the chunk, and $\mathcal{G}^e = (V_e, E_e)$ called the epilogue. Each of these sections can be any number of frames long. There are an additional set of edges E^\leftrightarrow between sections \mathcal{G}^p , \mathcal{G}^c , and \mathcal{G}^e , and these edges may be between adjacent or non-adjacent sections. A DBN is a directed acyclic graph \mathcal{G} that is formed by starting with \mathcal{G}^p , repeating \mathcal{G}^c any number of times, and ending with \mathcal{G}^e . Successive sections are connected with edges from E^\leftrightarrow . Factors $p(a|\text{parents}(a))$ reside in the time frame of the child a .*

One or more of \mathcal{G}^p , \mathcal{G}^c , and \mathcal{G}^e may be empty. If \mathcal{G}^p and \mathcal{G}^e are empty, the unrolled template corresponds chunk sections that are independent from each other. If \mathcal{G}^p and \mathcal{G}^c (resp. \mathcal{G}^c and \mathcal{G}^e) are empty, we have a static model in \mathcal{G}^e (resp. \mathcal{G}^p). If \mathcal{G}^e (resp. \mathcal{G}^p) is empty, then no factors that live within \mathcal{G}^c can ask for a neighbor in any following (resp. preceding) chunks.

The following is an informal definition of a DGM template.

Definition 140 (Definition of a DGM). *There are three sections of a graph, $\mathcal{G}^p = (V_p, E_p)$ called the prologue, $\mathcal{G}^c = (V_c, E_c)$ called the chunk, and $\mathcal{G}^e = (V_e, E_e)$ called the epilogue. Each of these sections can be any number of frames long. There are an additional set of edges E^- between any of the sections \mathcal{G}^p , \mathcal{G}^c , and \mathcal{G}^e , so these edges may be between adjacent or non-adjacent sections. A DGM is a factor graph \mathcal{G} that is formed by starting with \mathcal{G}^p , repeating \mathcal{G}^c any number of times, and ending with \mathcal{G}^e . Successive sections are connected with edges from E^- . Factors $\phi_c(x_c)$ in the template are given a particular time frame in which they reside.*

One or more of \mathcal{G}^p , \mathcal{G}^c , and \mathcal{G}^e may be empty. If \mathcal{G}^p and \mathcal{G}^e are empty, the unrolled template corresponds chunk sections that are independent from each other. If \mathcal{G}^p and \mathcal{G}^c (resp. \mathcal{G}^c and \mathcal{G}^e) are empty, we have a

static model in \mathcal{G}^e (resp. \mathcal{G}^p). If \mathcal{G}^e (resp. \mathcal{G}^p) is empty, then no factors that live within \mathcal{G}^c can ask for a neighbor in any following (resp. preceding) chunks.

Note that the definition of a DBN (definition 139) and DGM (definition 140) are very similar. In the DBN case we are only dealing with directed factors (i.e., a child and set of parents, as in a Bayesian network), while in a DGM we are dealing with factors. In the case of a DBN, each factor has a time location where it resides, and that is given by child variable. In the DGM case, the factor resides at a time given by a particular time value assigned to the factor. That is, in a DGM the variables are timed, and in a DGM both the variables and the factors are timed. We will make the definitions more precise as we go along and show examples. Note, right away, however, that these definitions allows one to express higher order models (i.e., from a chunk, one can ask for a neighbor more than one chunk into the past or even into the future, not to mention that a chunk itself might span multiple frames and could therefore contain within it edges over multiple frames). Note, however, that this can lead to complications, as discussed in §13.3.4.

Each block within a template has intra-vertex connectivity rules. The prologue is a graph over the vertices V_p but it also may contain edges that connect vertices in V_p to the other blocks, most typically the vertices V_c . The chunk is a graph over vertices V_c and can also in some cases include edges connecting either to the vertex set V_p or V_e or both. Symmetrically mirroring the prologue, the epilogue is a graph over vertices V_e and may also contain edge connections to the other blocks. It is even the case that vertices in V_p can connect directly to vertices in V_e . Arbitrary connections are not allowed, however, as there are certain rules (to be described more formally below) that must be obeyed in order to ensure that any template expansion leads to a valid graphical model.

Given an integer $\hat{\tau} \geq 0$, an “unrolling” or an “instantiation” of the template is a graph where \mathcal{G}^p appears once (on the left), \mathcal{G}^c appears $\hat{\tau} + 1$ times arranged in succession, and \mathcal{G}^e appears once on the right. Hence the smallest expansion is when $\hat{\tau} = 0$ corresponding to one chunk. If a prologue, chunk, and epilogue each are one frame long, then we have relationship $T = \hat{\tau} + 2$ where $\hat{\tau}$ is amount by which the template is unrolled (i.e., number of additional chunks beyond the chunk that exists in the template) and T is the number of frames in the expanded model.

The aforementioned rules ensure that once instantiated, there will always be some form of temporal independence property — that is, some notion of the present (which might need to be bigger than a chunk, see §13.3.4) renders some notion of the future, and of the past, independent. As we have seen, this property in a first-order Markov chain $Q_{1:T}$ means that each Q_t renders $Q_{1:t-1}$ and $Q_{t+1:T}$ independent. In a DGM, since each chunk cannot reach unboundedly into the past or future, there will always be some version of this property as well for a large enough expansion context window. We immediately see that given the above definition of a chunk, the temporal independence property might be generalized so that one must condition on more than a single time slice to ensure conditional independence of past and future. This means, for example, that for some $\Delta \geq 0$, the set of variables $Q_{t-\Delta:t+\Delta}$ will render Q_B independent of Q_A where B (respectively A) is any set of time indices less than $t - \Delta$ (respectively greater than $t + \Delta$). Before making this more precise (which we do in §8.11.3), we give a number of examples of the DBN template and unrollings.

8.11.2 Generalized DGM template examples

Our first example is one that recreates the expanded DBN model in Figure 8.74. The extended DBN template in this case has an empty epilogue as there is nothing different that happens at the end of any unrolled model. It is perhaps important to point out that an extended DBN template corresponds to a graph (shown in the middle in Figure 8.77) and a three-partition of that graph (shown as the different colors, red (prologue), green (chunk), and blue (epilogue) in Figure 8.77). Also shown are two unrolling amounts, unrolling three times (as shown on the left), and unrolling two times (as shown in the right). When unrolling by $\hat{\tau} = 0$, the

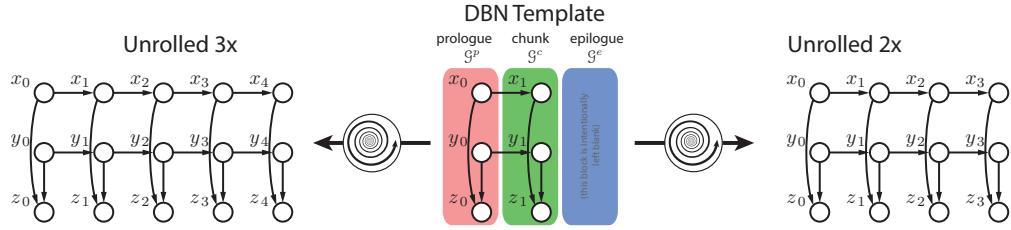


Figure 8.77: A generalized DBN template with a prologue, a chunk, and an empty epilogue that corresponds to Figure 8.74.

instantiated graph is the same as the template which means that the smallest number of frames for which this model covers is when $T = 2$.

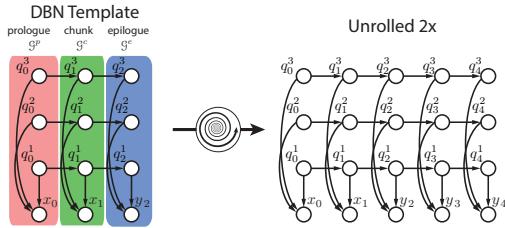


Figure 8.78: Factorial HMM DGM template with 3 chains.

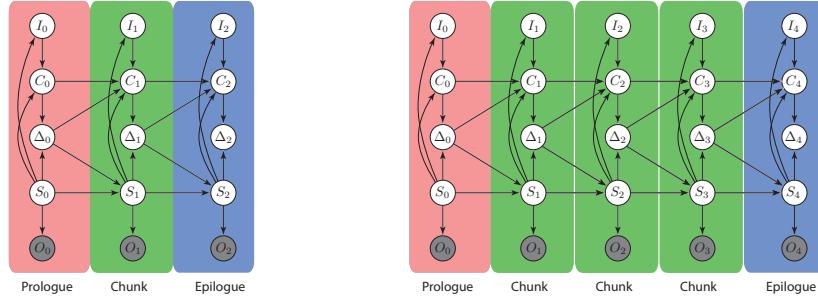
Note that Figure 8.77 is a generalized DBN template for a 2-chain factorial HMM. A three-chain version is shown in Figure 8.78.

Our second example in Figure 8.79 shows a template where, while the the epilogue is not empty, the epilogue has the same structure (i.e., set of vertices and edges) as the prologue. Just because the epilogue or prologue (or any two blocks) have the same structure, however, does not mean that having the block non-empty is redundant — it could be that the block has variables that are different in other ways — for example, they could be observed in the prologue and hidden in the chunk, or could have random variables with different domain sizes in the prologue vs. the chunk. Another difference, of course, is that the set of possible number of frames of an expanded model is more limited. That is, we must have the relationship $T = \hat{\tau} + 3$ for $\hat{\tau} \geq 0$.

It is worth pointing out at this point that each of the templates consist of vertices that are numbered according to time, and it is often the case that the same set of variables will repeat (even within a template) with the only difference being that the time indices are different. For example, in Figure 8.77, we see that each of the prologue, chunk, and epilogue consists of the set of variables $\{I_i, C_i, \Delta_i, S_i, O_i\}$ where for the prologue $i = 0$, the chunk $i = 1$, and the epilogue $i = 2$. In the unrolled model, the same pattern occurs in that the chunk is repeated and the indices of the random variables incremented. As mentioned above, it is important again to realize that even though two random variables in a DBN share the same name (e.g., I_1 and I_2 , or C_0 and C_1), the are entirely different random variables and may be governed by entire different distributions. Also, a random variable might be observed at one time (say $t = 0$) but hidden elsewhere.¹⁴

Our next example, shown in Figure 8.80, shows a basic HMM but where something different happens in the epilogue, and hence in the last frame of any expansion. In the figure, we see that there is a variable e_T that always occurs in the last frame (say T). This can be useful to force y_T to always be in a certain state at this last frame. More precisely, suppose we wish for $Y_T = \bar{y}$ to be the only state that occurs with non-zero probability. Then, we can make E_T observed (with value 1) and use the distribution $P(E_t = 1|Y_t = k) =$

¹⁴If it is the case that the random variables live in different sections, then the random variables might even have different random variable domain sizes, although such use is discouraged in GMTK.



(a) A DBN template that consists of a prologue, chunk, and epilogue, and that forms a simple segment model useful in bioinformatics [356].

(b) The template unrolled 2 times.

Figure 8.79: A template that contains non-empty prologue, chunk, and epilogue.

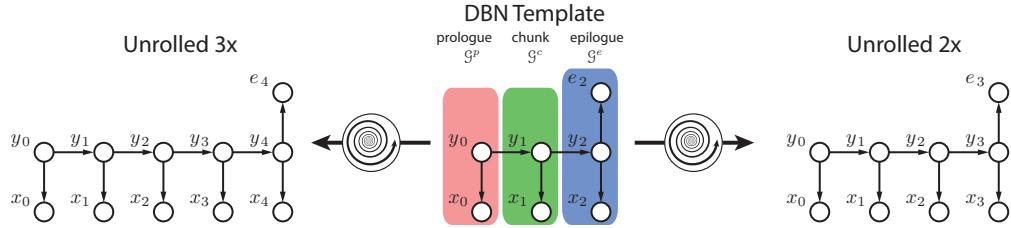


Figure 8.80: A generalized DBN template with a prologue, a chunk, and an epilogue having one extra variable e_3 , as well as two example unrollings. This template is for an HMM but where there is always an extra variable e_T in the very last frame.

$\delta(k = \bar{y})$. Then, whenever Y_T ends up at a value other than \bar{y} , it will be given a zero probability. We note in this example that in the template, there is only one instance of the variable e_2 (in the epilogue) and its time index is at three rather than, say, zero or one. Like above, we also have the relationship $T = \dot{\tau} + 3$.

Our next example, shown in Figure 8.81, demonstrates how sections can have more than one time-frame. Moreover, it demonstrates how the different sections might have differing numbers of frames from each other. In this case, G^p has only one frame, G^c two frames, and G^e has three frames. When blocks correspond to multiple frames, it puts further restrictions on the number of possible frames in the final unrolled model. The template itself (unrolling by $\dot{\tau} = 0$) corresponds to $T = 5$ frames, and unrolling by $\dot{\tau} = 1$ produces a model with seven frames. In general, we have the relationship $T = 2\dot{\tau} + 5$ in this template, which means we must have $T \in \{5, 7, 9, \dots\}$.

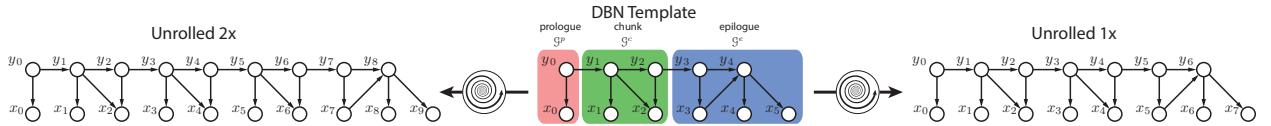


Figure 8.81: A generalized DBN template with a one-frame prologue, a two-frame chunk, and a three-frame epilogue, unrolled one time (right) and two times (left).

Because of the flexibility of the template, we can use it to create two separate Markov chains that evolve at a different rate. First, what do we mean by different rates?

On the one hand, we could have two Markov chains, say $(Y_t)_t$ and $(Z_t)_t$, each having a variable at each frame but where one tends, probabilistically, to change more slowly (or less frequently per unit time) than the other. For example, one of the chains $(Y_t)_t$ could use a length distribution that puts lower bounds on the shortest run a given state may last (i.e., a state can last, say, no less than 20 frames) while the other chain places no such restriction on length. Then, in any sampling of the chains, the subsegments during which $(Y_t)_t$ takes on the same state value would be much longer than the subsegments during which $(Y_t)_t$ takes on the same value. This might be occur, for example, with hierarchical HMMs where the highest level variable in the hierarchy tends to evolve slowly and at a “grander” scale than the lower-level variables.

Another form of multi-rate model is where there are multiple streams of random variables, but where the different streams use differing numbers of random variables per unit time. Here, we might have two Markov chains, where the first chain might have a random variable every time frame $(Y_t)_t$ for $t = 1, 2, \dots$ but the second one might have a random variable every other frame $(Z_\tau)_{\tau=2t}$ for $t = 1, 2, \dots$, half the rate as the first.

Both types of model might be called a multi-rate model. The fist can be set up via the parameters of the model and perhaps via some additional structural properties — many of these structural properties are used to set up a form of multi-rate hierarchical models commonly used in speech recognition (cf. Chapter 9).

An extended DBN template, moreover, can express both forms of multi-rate model. The first form is implicit and given by the CPTs, and the the second form of multi-frame model is given in the next example. Figure 8.82 shows a form of multi-rate mode, where the chunk can have different “streams” occurring at different rates. The different streams are coupled only every three frames.

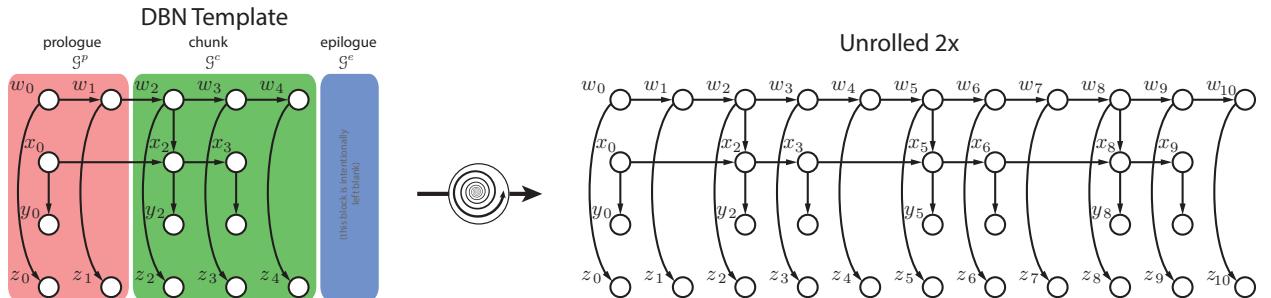


Figure 8.82: A generalized DBN template with a two-frame prologue, a three-frame chunk, and an empty epilogue, unrolled two times. The template corresponds to a form of multi-rate DBN since the x - and y -streams have random variables at $2/3$ the rate of the w - and z streams, yet they are coupled at times $2k + 4$ for $k \in \{0, 1, 2, \dots\}$.

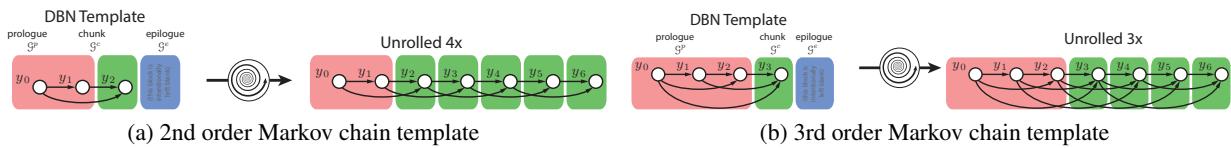


Figure 8.83: Generalized DBN templates for higher order Markov chains. Left, a 2nd-order Markov chain and right, a 3rd-order Markov chain template.

In §8.3, we discussed Markov chains and how they can have an order greater than just one. For example, Figure 8.4 showed examples of second- and third-order Markov chains. Indeed, it is possible to provide generalized DBN templates for such chains as shown in Figure 8.83. A key property of these templates is that, since the chunk can reach more than one frames back in time, the prologue must be long enough to allow this.

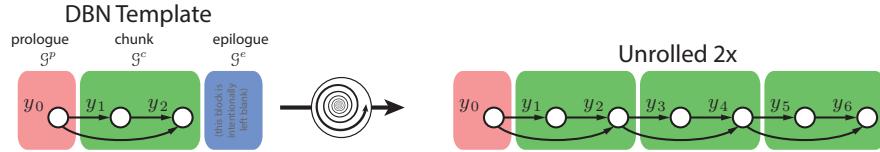


Figure 8.84: A small change to the chunk can make a big difference. By making the chunk two frames long as shown here rather than one frame long as shown in Figure 8.83a, the expanded model is entirely different, and in this case might be seen as a multi-rate multi-order Markov chain, where at even frames it is a second-order model while at odd frames it is a first-order model.

Figure 8.84 shows another example where a small change in the template can make a big difference in the family of models of any expansion. Comparing Figure 8.84 with Figure 8.83a, we see that the only difference in the template is the range of the chunk, which goes from one frame to two-frames long. Other than that, the templates are the same and contain exactly the same number of random variables and the same edge connectivity. When expanding Figure 8.84, however, two variables are created for every unit of unrolling and the single parent status of y_1 in Figure 8.84 is retained at every instantiation of the chunk. Hence, a bit of care needs to be taken when defining these templates.

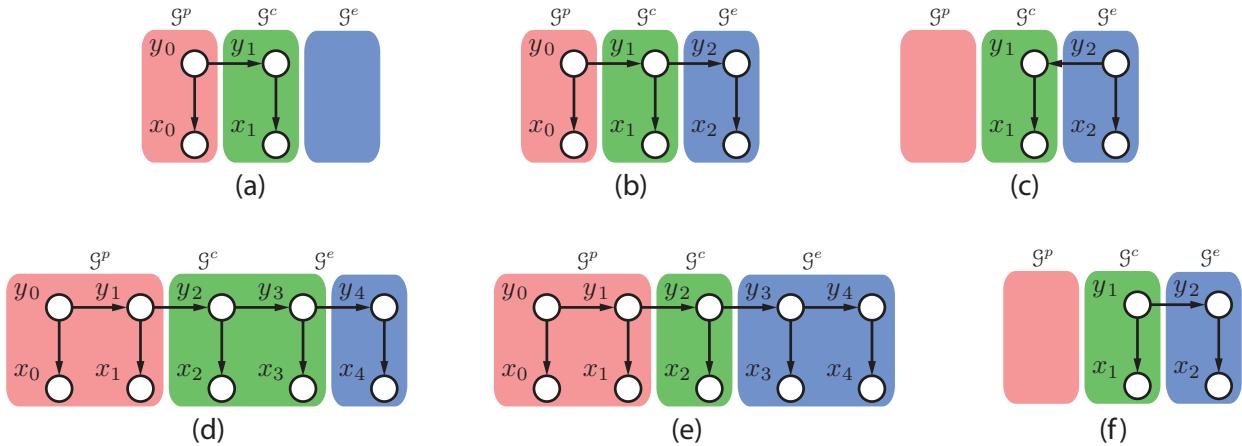


Figure 8.85: All templates correspond to HMMs except for (f). Note that the length constraints on T are different. For example, in (a), we must have $T = \hat{r} + 2$; in (b), we have $T = \hat{r} + 3$; in (c), we have $T = \hat{r} + 2$; in (d), we have $T = 2\hat{r} + 5$; in (e), we have $T = \hat{r} + 5$; and in (f), we have $T = \hat{r} + 2$ even though it is not an HMM.

Note that while the generalize template is flexible, the problem of choosing a template that can generate a given class of dynamic Bayesian networks is in some sense an ill-posed problem since there are multiple solutions. That is, it is possible to express the same expansion class in multiple ways. A perhaps startling example is that, for even a model as simple as a standard HMM, there are an unlimited number of templates (albeit with different constraints placed on the expanded length). Figure 8.85-(a)-(e), for example, shows examples of templates that generate the class of HMMs and the corresponding length constraints.

Figure 8.85-(c) is perhaps an interesting example since the temporal edge points backwards in time, $y_2 \rightarrow y_1$. Any expanded HMM, hence, would have arrows reversed in time. Since the template, and any unrollings of the template, do not violate any of the BN rules (i.e., a DAG), this is a valid template. While backwards time edges might be convenient for applications (and we give a few cases in Chapter 9), we see that this does not increase the model family since an HMM has no v-structures, so reversing the edge in this way does not change the model class.

Figure 8.85 also shows one example (Figure 8.85-(f)) that does not generate an HMM. The reason an HMM is not generated has to do with the rules for template expansion and stems from a question we have not yet addressed, namely where does a factor from a template live (or reside) in any expansion? In a BN, we take the convention the factor lives in the frame where the child CPT is. For example, if we have a CPT factor of the form $p(A_{t_1}|B_{t_2}, C_{t_3})$ where t_1, t_2, t_3 are distinct, then the conditional factor resides in the frame at time t_1 . Therefore, when we unroll Figure 8.85-(f), there is only one factor in \mathcal{G}^e involving $p(y_T|x_T, y_{T-1})$ and all the remaining factors are of the form $p(x_t|y_t)$ with no connection between the y_t 's except for between times T and $T - 1$. Note that we say it does not generate an HMM in a loose sense in that the family of distributions corresponding to this model, when expanded, does lie within the family of distributions of an HMM. Figure 8.85-(f)'s family, however, requires an additional constraint, namely that $Y_t \perp\!\!\!\perp Y_{t-1}$ for all $t \neq T$, not required by an HMM.

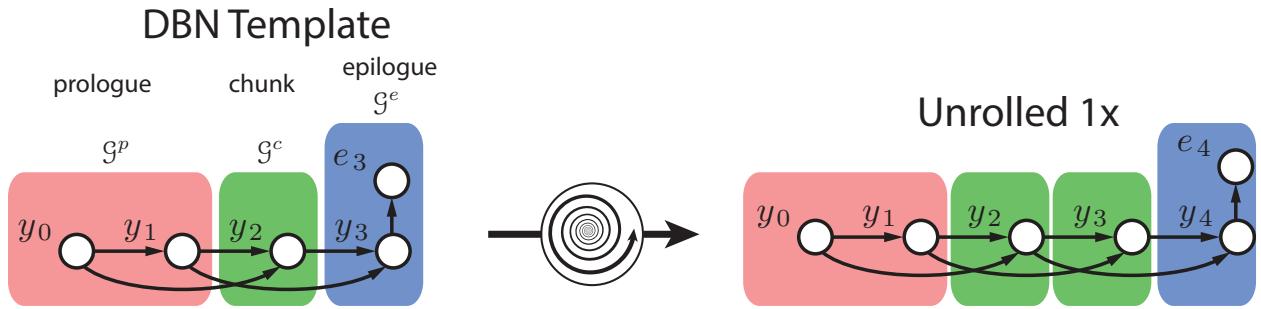


Figure 8.86: In this model, we have an edge in the template between the prologue and the epilogue. This does not mean that after an any amount of unrolling, we still have an edge between \mathcal{G}^p and \mathcal{G}^e . Instead, the edge from ending at y_3 and starting at y_1 in the template will be an edge ending at y_T for some T and starting at y_{T-2} . We say that the factor ending at y_3 in the template “resides” in the epilogue at time frame 3, and wherever the vertex y_3 is instantiated in an unrolling, any edges directed towards the corresponding vertex must come from the corresponding relative position in the instantiation.

The time location where a factor resides is also important in the context of defining an expansion in higher order models. For example, in Figure 8.86, we have an edge between the epilogue and the prologue. Does this mean that, after any unrolling, there will always be an edge between the epilogue and prologue? The answer is no, for several reasons, the primary one being that this would mean that there is never a temporal Markov property, since the chunk, or any sequence of chunks, would never render the past and the future independent. In this case, we use the fact that the factor $p(y_3|y_2, y_1)$ resides at time frame three, and after any expansion that factor reaches into the past by only three frames. We discuss this more in § 8.11.3.

In undirected graphical models, we have factors of the form $\phi(A_{t_1}, B_{t_2}, C_{t_3})$ so it is ambiguous where in time the factor should reside. In such case, further specification needs to be given to determine where the factor actually lives, what is called the *factor time*. In such case, we will grant a time value to the factor itself, and use notation such as $\phi_t(A_{t_1}, B_{t_2}, C_{t_3})$. This is natural in the case of factor graphs, where the factors themselves are represented as vertices in a bipartite graph. Without loss of generality, the time of a factor must be one of the times of its random variables. This means that, for example, $t \in \{t_1, t_2, t_3\}$.

8.11.3 Formal description of generalized DBN template

A generalized DBN template is a graph $G = (V, E)$ whose vertices are partitioned into three blocks: the prologue $\mathcal{G}^p = (V_p, E_p)$, the chunk $\mathcal{G}^c = (V_c, E_c)$, and the epilogue $\mathcal{G}^e = (V_e, E_e)$. We specify how the edges are partitioned below. For each vertex $v \in V(G)$, there is an associated time integer and we denote this as the function $t(v) \in \mathbb{Z}_+ = \{0, 1, 2, \dots\}$. We call any such graph, where a integer time is associated

with every vertex, a “timed graph.”

There are are pair of specific times (t_1, t_2) that are associated with the template that partition G into three sections. Some of the sections may be empty. The pair (t_1, t_2) can either be seen as providing the definition of \mathcal{G}^p , \mathcal{G}^c , and \mathcal{G}^e (so they are functions of (t_1, t_2)). Alternatively, we might say that $\exists(t_1, t_2)$ in the template with such properties. That is, there are two ways of defining DBN, one way would require t_1, t_2 to be part of the definition, and the other way would say that G above must have the property that there exists such a t_1, t_2 .

In either case, we have that $t_1 \leq t_2$, and that (t_1, t_2) designate the partitioning of the template as follows:

$$V(\mathcal{G}^p) = \{v \in V(G) : t(v) < t_1\} \quad (8.484)$$

$$V(\mathcal{G}^c) = \{v \in V(G) : t_1 \leq t(v) \leq t_2\} \quad (8.485)$$

$$V(\mathcal{G}^e) = \{v \in V(G) : t_2 < t(v)\} \quad (8.486)$$

In words, the vertices in the prologue block \mathcal{G}^p are those whose times are no more than t_1 . The vertices in the epilogue are those greater than t_2 , and the chunk are those between t_1 and t_2 inclusively.

The *time length of template* is $T(G) = 1 + \max_{v \in V(G)} t(v)$. Note that we add the one since the first frame is at time zero. The *time duration (or length) of the sections* \mathcal{G}^p , \mathcal{G}^c , and \mathcal{G}^e are $T(\mathcal{G}^p) = t_1$, $T(\mathcal{G}^c) = t_2 - t_1 + 1$, and $T(\mathcal{G}^e) = T(G) - t_2$. We also define the $\Delta(\tau)$ operator which take a set of random variables and shifts their time index by an integer (which could be negative). For example, let $U = \{Q_t, Q_{t+1}\}$ then $U_{\Delta(\tau)} = \{Q_t, Q_{t+1}\}_{\Delta(\tau)} = \{Q_{t+\tau}, Q_{t+1+\tau}\}$. As a result of this operation, we get a new set of vertices (i.e., a copy of the original ones, where the copies are isomorphic to the original set but where the copy has a time shift). Note that $\Delta(\tau)$ can apply to edges also, since edges are pairs of vertices, so $E_{\Delta(\tau)}$ just shifts the time of each vertex in each pair for each edge in E . Given a graph $G = (V, E)$ we have that $G_{\Delta(\tau)} = (V_{\Delta(\tau)}, E_{\Delta(\tau)})$ where $V_{\Delta(\tau)}$ (resp. $E_{\Delta(\tau)}$) is isomorphic to V (resp. E). E.g., if $v \in V(G)$, and $t(v) = i$ then the corresponding $v' \in V(G_{\Delta(\tau)})$ has $t(v') = i + \tau$.

Defining the generalized template isn’t as easy as just a graph partitioning, as we have so far done above. We also need rules that specify what edges may exist between the sections, and given such edges, how unrolling is allowed to proceed.

The generic DGM template therefore is a timed graph $G = (V, E)$ where there is a three-partition of the vertices of G

$$V = (V^p \cup V^c \cup V^e) \quad (8.487)$$

where $V^p = V(\mathcal{G}^p)$, $V^c = V(\mathcal{G}^c)$, and $V^e = V(\mathcal{G}^e)$. There is, however, also a partition of the edges of G into the following blocks of edges:

$$E = (E^p \cup E^c \cup E^e \cup E^{pc} \cup E^{cp} \cup E^{ce} \cup E^{ec} \cup E^{pe} \cup E^{ep}). \quad (8.488)$$

The blocks E^p , E^c , and E^e indicate intra-section (meaning strictly within section) edges for the prologue, chunk, and epilogue sections respectively.

Edges between the prologue and chunk are given by $E^{pc} \cup E^{cp}$; E^{pc} correspond to those edges that are due to a factor that resides within the prologue (e.g., in BN case, a child variable is in prologue). Such edges are said to “reside” in the prologue even though they connect between the prologue and the chunk. The edges E^{cp} corresponds to those edges that are due to a factor that resides within the chunk (e.g., in BN case, a child variable is in chunk). The edges E^{cp} are said to reside in the chunk.

The edges that connect the chunk and epilogue are given by $E^{ce} \cup E^{ec}$ where, similarly, the edges E^{ce} , which reside in the chunk, are due to factors residing in the chunk and edges E^{ec} , which reside in the epilogue, are due to factors residing in the epilogue.

There are also potentially edges between prologue and epilogue (see Figure 8.86) and are indicated as $E^{pe} \cup E^{ep}$. Like in the above, the edges E^{pe} reside in the prologue, and E^{ep} reside in the epilogue.

A template unrolling corresponds to taking the chunk and repeating it a certain number of times. Given an integer $\hat{\tau} \geq 0$, we can unroll the template by $\hat{\tau}$ by forming the graph

$$G_{\hat{\tau}} = (V_{\hat{\tau}}, E_{\hat{\tau}}) \quad (8.489)$$

where

$$V_{\hat{\tau}} = V^p \cup \bigcup_{\ell=0}^{\hat{\tau}} V_{\Delta(\ell T(\mathcal{G}^c))}^c \cup V_{\Delta(\hat{\tau} T(\mathcal{G}^c))}^e \quad (8.490)$$

and

$$\begin{aligned} E_{\hat{\tau}} = & E^p \cup E^{pc} \cup E^{pe} \cup \bigcup_{\ell=0}^{\hat{\tau}} \left(E_{\Delta(\ell T(\mathcal{G}^c))}^c \cup E_{\Delta(\ell T(\mathcal{G}^c))}^{cp} \cup E_{\Delta(\ell T(\mathcal{G}^c))}^{ce} \right) \\ & \cup E_{\Delta(\hat{\tau} T(\mathcal{G}^c))}^{ec} \cup E_{\Delta(\hat{\tau} T(\mathcal{G}^c))}^e \cup E_{\Delta(\hat{\tau} T(\mathcal{G}^c))}^{ep} \end{aligned} \quad (8.491)$$

Notice in the above that the edges that reside strictly in \mathcal{G}^p or \mathcal{G}^e are treated differently than those that reside in the chunk — i.e., they don't get repeated. In fact, none of the edges $E^p, E^e, E^{pc}, E^{pe}, E^{ec}$, and E^{ep} get repeated. Only edges that reside in the chunk, namely E^c, E^{cp} , and E^{ce} are repeated. This is the reason the template in Figure 8.85-(f) does not correspond to a standard HMM, since the edge (y_1, y_2) in the template resides in \mathcal{G}^e and the edge $y_1 \rightarrow y_2$ in the template is not repeated.

Hence, there is no edge stretching or “rubber banding.” That is, if there are any edges between \mathcal{G}^p and \mathcal{G}^e in the template, then after instantiation, for large enough $\hat{\tau}$, there will not any longer be edges between \mathcal{G}^p and \mathcal{G}^e .

The above therefore constitutes the semantics of unrolling. We can view the unrolling process in a DGM as the following: we first ignore the edges in the graph instantiate the vertices of the unrolled graph by copying the chunk $\hat{\tau}$ number of times, shifting the time indices as appropriate. Once we have such a graph, we add the edges between nodes depending on where the factors reside in the instantiated model. For any Bayesian network factor (a CPT), we add directed edges towards the child variable using the same relative positions that the edges have in the template. For undirected factors, we add edges between the corresponding random variables at the new time. Also, since each factor has a factor time, and since at least one random variable in the factor has a time the same as the factor time, we instantiate a factor also at the same relative time as its random variable having the same time. For example, given template factor $\phi_t(A_{t_1}, B_{t_2}, C_{t_3})$, and suppose that $t = t_1$. After unrolling, at some point the three random variables have times t'_1, t'_2 , and t'_3 , so we add an edge between the three variables at this time. Also, since $t = t_1$ in the template, we instantiate a factor $\phi_{t'_1}$ at time t'_1 . We do this for each such instantiation of the three random variables.

8.11.4 DGM Template Validity

The next question we must address is how can we be sure that the graph given in the template $G = (V, E)$ leads to a valid graph in the unrolled instance $G_{\hat{\tau}}$ in Equations (8.489), (8.490), and (8.491). For example, it could very well be that G is a valid Bayesian network when seen as a template, but for some unrolling $\hat{\tau} > 0$, then $G_{\hat{\tau}}$ does not make sense.¹⁵ As an example, suppose that there is a vertex in $X_t \in V(\mathcal{G}^c)$ that is involved in a factor $p(X_t | Y_{t-4})$ reaching back four time frames. For the template to be valid, there must exist a frame with a time index of at least four ($t \geq 4$ since frame numbers start at zero) within the

¹⁵It is also important see §13.3.3 for a discussion on template validity in the context of inference procedures, where additional restrictions are placed on the validity of a template.

chunk. Moreover, there must be a variable Y_{t-4} either in the chunk four frames earlier than X_t (meaning the chunk must be at least four frames wide) or alternatively Y_{t-4} must reside in the prologue. Lets assume that the latter is true, so that X_t 's parent in the template resides within the prologue as shown on the left in Figure 8.87.

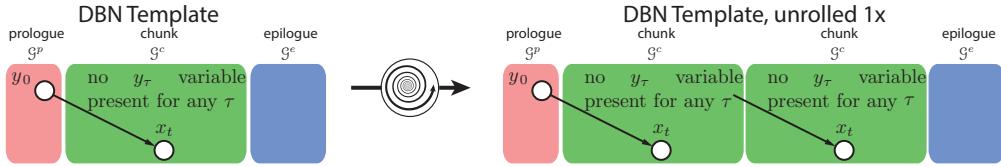


Figure 8.87: Left: Invalid template since when unrolled 1x (right), x_t 's parent does not exist.

Once the template is unrolled even one time, as shown in the right in Figure 8.87, then there can be a problem if the chunk does not also have a compatible variable. The problem here is that the chunk does not have any y_t variable and hence there is no way for the template factor $p(X_t|Y_{t-4})$, once it is instantiated in the unrolled model, to find a matching parent variable. The template therefore is invalid even though when unrolled zero times, it corresponds to a valid Bayesian network.

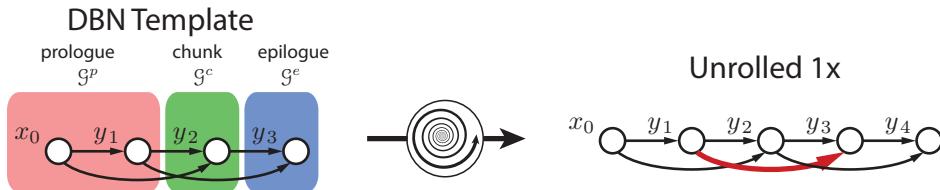


Figure 8.88: Left: Invalid template since when unrolled 1x (right), y_3 's parent y_1 (indicated by the red edge) does not match the template. In the template y_2 (who is the variable from whence y_3 is instantiated) must have two parents x_0 and y_1 , and it may be the case that x_0 and y_1 are completely different (e.g., they might have different domain sizes, so that $|D_{X_0}| \neq |D_{Y_1}|$).

Another example of an valid template before unrolling but invalid after unrolling is show in Figure 8.88. Here, the problem is that when a template variable (y_2 in this example) is expanded, the environments in any expanded model for any τ must match the environment it was in within the template, namely two potentially heterogeneous variables x_0 and y_1 . It could be that x_0 and y_1 are very different, so the CPT $p(Y_2 = y_2|Y_1 = y_1, X_0 = x_0)$ can not be used for $p(Y_2 = y_3|Y_1 = y_2, X_0 = y_1)$.

Hence, there are rules that must be followed in a template in order so that Equations (8.489), (8.490), and (8.491) lead to a valid expanded model for all $\tau \geq 0$. Clearly, if the template is valid, then so is the expansion for $\tau = 0$, so we consider only $\tau > 0$.

There are two key features of the template that lead to difficulties when the template is unrolled:

1. The neighbors adjacent to a vertex in the template might reside in a different section after expansion than before expansion.

For example, in valid templates in Figure 8.83 and the invalid templates in Figures 8.87 and 8.88, parents of template chunk vertices resided in the prologue in the template but after unrolling some (or all) of the can reside in earlier chunks.

This can mean that neighbors of nodes that are compatible with factors before expansion might not be compatible with factors after expansion. Hence, we need to be sure that there is a representative set of variables in any template expansion.

2. Edges incident to vertices that reside in \mathcal{G}^c can reach into the past by any number of chunks or could reach into the future by any number of chunks.

There is a reason for this. One of the goals of the template formulation is to be able to specify arbitrarily high (but fixed) order Markov chains (and their generalizations). I.e., the template should be able to express factors of the form $p(Y_t|Y_{t-1}, Y_{t-2}, \dots, Y_{t-h})$ for any fixed h , and this should be allowed in the template. For example, see Figure 8.83.

Moreover, edges can reach into the future for two reasons. First, if we have a Bayesian network, we are allowed factors with reverse-time edges, such as $p(Y_t|Y_{t+1}, Y_{t+2}, \dots, Y_{t+h})$. Such factors reside in the section where Y_t resides, so if Y_t resides in the chunk, edges can reach into the future. Also, in an undirected model, factors might reside in the chunk that involve variables not only from the past but also into the future.

We can summarize the requirements that would avoid the above problems as follows:

- Vertices that reside in the template chunk \mathcal{G}^c might have neighbors either in \mathcal{G}^p or \mathcal{G}^e but after expansion, instances of \mathcal{G}^c might require corresponding neighbors either at a different position within \mathcal{G}^p and \mathcal{G}^e or within other chunks as well. The chunks where these neighbors are required might be either to the left or to the right of the vertices .
- Vertices that reside in the template prologue \mathcal{G}^p might have neighbors either in \mathcal{G}^c or \mathcal{G}^e but after expansion, those neighbors might instead be required to be either in a different position within \mathcal{G}^e or within other chunks as well. The chunks where these neighbors are required might be will be to right of the vertices.
- And symmetrically, vertices that reside in the template epilogue \mathcal{G}^e might have neighbors either in \mathcal{G}^c or \mathcal{G}^p but after expansion, those neighbors might instead be required to be either in a different position within \mathcal{G}^p or within other chunks as well. The chunks where these neighbors are required might be will be to left of the vertices.

Any template that obeys the above requirements is called *valid*.

We next give set theoretic properties that, if satisfied, will ensure that the template is valid. First a bit of additional notation. Lets say that the graph $G = (V, E)$ is the template consisting of sections \mathcal{G}^p , \mathcal{G}^c , and \mathcal{G}^e . After unrolling by $\hat{\tau}$, we call the model $G_{\hat{\tau}} = (V_{\hat{\tau}}, E_{\hat{\tau}})$ as in Equations (8.489), (8.490), and (8.491). In the unrolled model, the prologue becomes $\mathcal{G}_{\hat{\tau}}^p$ and there is only one such section. Of course, $\mathcal{G}_{\hat{\tau}}^p$ is isomorphic with \mathcal{G}^p , and we moreover can say that $\mathcal{G}_{\hat{\tau}}^p = \mathcal{G}_{\Delta(0)}^p$ since the time values of the vertices have not changed. After unrolling, there are $\hat{\tau} + 1$ chunks, and we refer to them as \mathcal{G}_i^c for $i \in \{0, 1, \dots, \hat{\tau}\}$. We say that $\mathcal{G}_i^c = \mathcal{G}_{\Delta(iT(\mathcal{G}^c))}^c$ since \mathcal{G}_i^c has had its vertices and edges shifted by i times the length of a chunk relative to \mathcal{G}^c . Lastly, after unrolling there is still only one epilogue and we name it $\mathcal{G}_{\hat{\tau}}^e$ where $\mathcal{G}_{\hat{\tau}}^e = \mathcal{G}_{\Delta(\hat{\tau}T(\mathcal{G}^c))}^e$ since $\mathcal{G}_{\hat{\tau}}^e$ is shifted to the right by $\hat{\tau}$ times the length of a chunk.

First, we should also comment on precisely what we mean by set intersection of a set of timed vertices. Lets say that we have two sets A and B of timed vertices. Since each vertex v is timed, it has both a name and a time value, so $v = (n_v, t_v)$ where n_v is the name (and could be thought of as a strong) and $t_v \geq 0$ is an integer. For sets of timed vertices A and B , the intersection $A \cap B$ means the following:

$$A \cap B = \{v : u \in A, v \in B, n_u = n_v, \text{ and } t_u = t_v\} \quad (8.492)$$

In other words, the vertices need to have the same name and time index, and having both the same leads to timed vertex equality. Timed set intersection will be used in part to define notions of template validity.

We want to be able to designate the neighbors of \mathcal{G}^c in \mathcal{G}^p shifted to the right by a certain amount of time. Let Γ be a the neighbor function of the set of nodes S or of a graph G_s — that is, given a graph $G = (V, E)$,

and a subset $U \subseteq V$, a vertex $v \in V$ is a neighbor of U if $v \notin U$ and v has a neighbor within U . Then the neighbors of U is designated $\Gamma(U)$ (note, this is sometimes called the node boundary function). Note that the neighbors of a set U do not consist of the nodes internal to U . When we have a subgraph G_s of G , we use the same notion, so $\Gamma(G_s)$ corresponds to the neighbors in G not in G_s . Hence, we can ask either for $\Gamma(S)$ or for $\Gamma(G_s)$ where S is a subset of vertices, and G_s is a subgraph.

Thus, $\Gamma(\mathcal{G}^c)$ is the set of vertices outside of \mathcal{G}^c that are neighbors of *any* vertex that resides within \mathcal{G}^c . Keep in mind, Γ does not give internal neighbors, rather only the ones that are outside the section on which Γ is applied. We apply the neighbor function to \mathcal{G}^p and \mathcal{G}^e as well. We also use graph intersection notation in the usual sense, so that given two graphs G and G' , $G \cap G'$ is the induced graph on the vertices $V(G) \cap V(G')$. For convenience, we also allow a set of neighbors to intersect a graph, again returning the induced graph on the vertex set intersection.

Note that the left neighbors of a chunk can be denoted $\Gamma(\mathcal{G}^c) \cap \mathcal{G}^p$. When we unroll one time, we must have that any of \mathcal{G}_1^c 's neighbors to the left can be found. This is equivalent to the condition:

$$(\Gamma(\mathcal{G}^c) \cap \mathcal{G}^p)_{\Delta(T(\mathcal{G}^c))} \subseteq \mathcal{G}^p \cup \mathcal{G}^c \quad (8.493)$$

This can be generalized to unrolling by any amount $\dot{\tau}$ by saying that

$$(\Gamma(\mathcal{G}^c) \cap \mathcal{G}^p)_{\Delta(\dot{\tau}T(\mathcal{G}^c))} \subseteq \mathcal{G}_{\dot{\tau}}^p \cup \mathcal{G}_0^c \cup \mathcal{G}_1^c \cup \cdots \cup \mathcal{G}_{\dot{\tau}-1}^c \quad (8.494)$$

In fact, since \mathcal{G}^p is finite sized, if the above is satisfied, there will be some $\dot{\tau}_c^\ell$ such that for all $\dot{\tau} \geq \dot{\tau}_c^\ell$ we have that

$$(\Gamma(\mathcal{G}^c) \cap \mathcal{G}^p)_{\Delta(\dot{\tau}T(\mathcal{G}^c))} \subseteq \mathcal{G}_{\dot{\tau}-\dot{\tau}_c^\ell}^c \cup \cdots \cup \mathcal{G}_{\dot{\tau}-1}^c \quad (8.495)$$

meaning all of \mathcal{G}_t^c 's neighbors to the left lie within the previous $\dot{\tau}_c^\ell$ chunks and no longer reach back to the prologue. For example, $\dot{\tau}_c^\ell = 2$ for the template in Figure 8.83a and $\dot{\tau}_c^\ell = 3$ for the template in Figure 8.83b. In any event, Equation (8.494) specifies what must be true of the chunk's neighbors to the left.

As can be imagined, an analogous requirement must be true for the chunks neighbors on the right:

$$\Gamma(\mathcal{G}^c) \cap \mathcal{G}^e \subseteq \mathcal{G}_1^c \cup \cdots \cup \mathcal{G}_{\dot{\tau}}^c \cup \mathcal{G}_{\dot{\tau}}^e \quad (8.496)$$

where at some point $\dot{\tau} \geq \dot{\tau}_c^r$ the chunk no longer reaches into the epilogue and we have:

$$\Gamma(\mathcal{G}^c) \cap \mathcal{G}^e \subseteq \mathcal{G}_1^c \cup \cdots \cup \mathcal{G}_{\dot{\tau}_c^r}^c \quad (8.497)$$

Also for the prologue's neighbors on the right we have the requirement that:

$$\Gamma(\mathcal{G}^p) \subseteq \mathcal{G}_0^c \cup \cdots \cup \mathcal{G}_\tau^c \cup \mathcal{G}_\tau^e \quad (8.498)$$

where at some point $\tau \geq \tau_p^r$ the prologue no longer reaches into the epilogue and we have:

$$\Gamma(\mathcal{G}^p) \subseteq \mathcal{G}_0^c \cup \cdots \cup \mathcal{G}_{\tau_p^r}^c \quad (8.499)$$

And lastly, the epilogue's neighbors on the left:

$$(\Gamma(\mathcal{G}^e))_{\Delta(\tau T(\mathcal{G}^c))} \subseteq \mathcal{G}_\tau^p \cup \mathcal{G}_0^c \cup \mathcal{G}_1^c \cup \cdots \cup \mathcal{G}_\tau^c \quad (8.500)$$

where at some point $\tau \geq \tau_e^\ell$ the epilogue's no longer reaches into the prologue and we have:

$$(\Gamma(\mathcal{G}^e))_{\Delta(\tau T(\mathcal{G}^c))} \subseteq \mathcal{G}_{\tau-\tau_e^\ell}^c \cup \cdots \cup \mathcal{G}_\tau^c \quad (8.501)$$

It can be seen that the following will ensure that the template will be valid in the above sense for any expansion.

Theorem 141. If Equations (8.494), (8.496), (8.498), and (8.500) all hold for $\tau \leq \max(\tau_c^\ell, \tau_c^r, \tau_p^r, \tau_e^\ell)$ then the template will be valid for all larger τ .

This theorem is valuable since $\max(\tau_c^\ell, \tau_c^r, \tau_p^r, \tau_e^\ell)$ is bounded, and is upper bounded by the number of variables within the chunk, prologue, and epilogue. In fact, we get as simple corollary:

Corollary 142. If Equations (8.494), (8.496), (8.498), and (8.500) all hold for $\tau \leq |V(G)|$ then the template will be valid for all larger τ , where $|V(G)|$ is the number of vertices in the template.

One further complexity does arise, however, in the case of dynamic Bayesian networks. In this case, we need to ensure not only that the neighbors in the unrolled template match, but also that no directed cycles are ever introduced in any unrolling for any τ . Indeed, there can be examples where a template is acyclic but an unrolling is not, as shown in Figure 8.89.

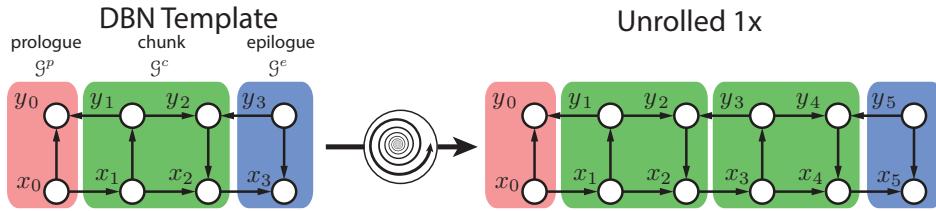


Figure 8.89: Left: Invalid template since when unrolled 1x (right), it creates a directed cycle between two chunk sections $x_2 \rightarrow x_3 \rightarrow y_3 \rightarrow y_2 \rightarrow x_2$.

Theorem 143. If there are no directed cycles introduced in any unrolling $0 \leq \hat{\tau} \leq |V(G)|$, then there are no directed cycles introduced in any unrolling $\hat{\tau} > |V(G)|$.

This theorem can be strengthened a bit using $|V(\mathcal{G}^c)|$ rather than $|V(G)|$.

There is one more property that we must ensure a template has. While we do not here discuss why this property is important (we defer this to our discussion on DGM inference in Chapter 13, in particular, §13.3.3 and §13.3.4), it is still important to take note of the template restrictions here.

We have the basic three sections \mathcal{G}^p , \mathcal{G}^c , and \mathcal{G}^e and an unrolling one time \mathcal{G}_1^p , \mathcal{G}_1^c , \mathcal{G}_2^c , \mathcal{G}_1^e where \mathcal{G}_2^c is \mathcal{G}_1^c shifted by time value $T(\mathcal{G}^c)$ and \mathcal{G}_1^e is shifted by time value $T(\mathcal{G}^e)$. We assume here that the template has been moralized or that the model is undirected so that in either case, all edges are undirected. For the template to be valid, we must have at least one of the following conditions being true. Either:

$$\Gamma(\mathcal{G}^p) \cap (\mathcal{G}^c \cup \mathcal{G}^e) = \Gamma(\mathcal{G}_1^p) \cap (\mathcal{G}_1^c \cup \mathcal{G}_2^c \cup \mathcal{G}_1^e) = [\Gamma(\mathcal{G}_1^p \cup \mathcal{G}_1^c) \cap (\mathcal{G}_2^c \cup \mathcal{G}_1^e)]_{\Delta(-T(\mathcal{G}^c))} \quad (8.502)$$

or

$$\Gamma(\mathcal{G}^e) \cap (\mathcal{G}^p \cup \mathcal{G}^c) = \Gamma(\mathcal{G}_1^e) \cap (\mathcal{G}_1^p \cup \mathcal{G}_1^c \cup \mathcal{G}_2^c) = [\Gamma(\mathcal{G}_1^e \cup \mathcal{G}_2^c) \cap (\mathcal{G}_1^p \cup \mathcal{G}_1^c)]_{\Delta(T(\mathcal{G}^c))} \quad (8.503)$$

In words, Equation (8.502) is saying that a set of vertex separators must be the same or isomorphic. That is, the set of vertices within $(\mathcal{G}^c \cup \mathcal{G}^e)$ that separate $(\mathcal{G}^c \cup \mathcal{G}^e)$ from \mathcal{G}^p must be the same as the separator within $(\mathcal{G}_1^c \cup \mathcal{G}_2^c \cup \mathcal{G}_1^e)$ that separates $(\mathcal{G}_1^c \cup \mathcal{G}_2^c \cup \mathcal{G}_1^e)$ from \mathcal{G}_1^p — this, in turn, must also be the same as the shifted set of vertices within $(\mathcal{G}_2^c \cup \mathcal{G}_1^e)$ that separate $(\mathcal{G}_2^c \cup \mathcal{G}_1^e)$ from $\mathcal{G}_1^p \cup \mathcal{G}_1^c$. Equation (8.502) is known as the basic left interface requirement of the template. An example of a template that satisfies this requirement is shown in Figure 8.90a, where on the top the two yellow vertices are the same as in the middle and are also the same as the bottom when the bottom two yellow nodes are shifted left by $T(\mathcal{G}^c) = 1$.

Equation (8.503) is known as the basic right interface requirement of the template, and has a symmetric set of requirements as does Equation (8.502). Figure 8.90a, where on the top the two yellow vertices are the

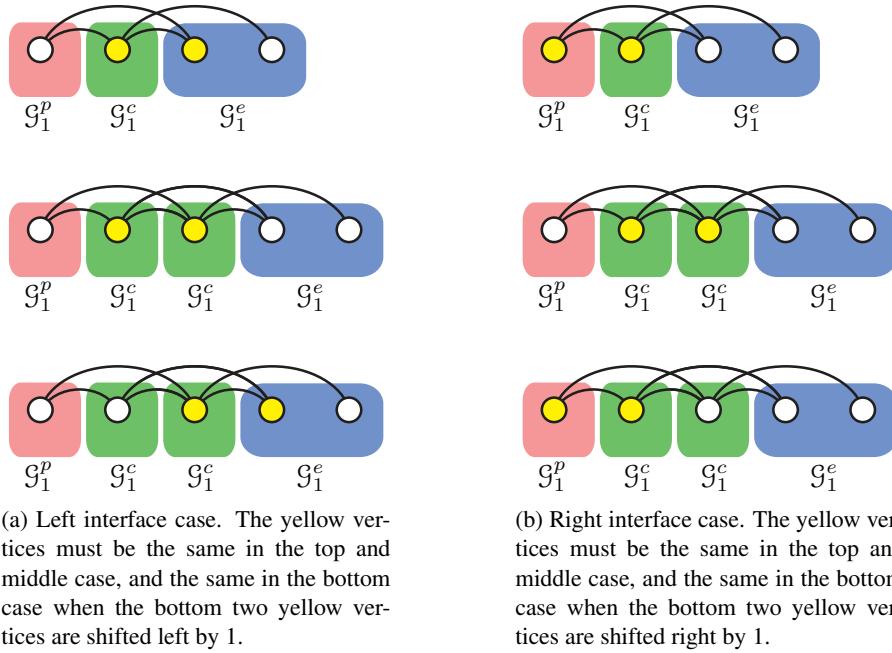


Figure 8.90: Interface requirements: an example template that satisfies these requirements

same as in the middle and are also the same as the bottom when the bottom two yellow nodes are shifted right by $T(\mathcal{G}^c) = 1$.

There is a slightly weaker set of requirement that are based on an additional parameter that we will fully discuss in §13.3.3 and §13.3.4, namely, the chunk skip parameter $S \geq 1$. The value S is the number of original chunks that may be used together in a single static inference component, and it can change the requirements on the template. This parameter and its meaning, however, can be skipped on first reading and in fact is only useful for more advanced models and inference strategies.

8.12 Other topics

8.12.1 HMMs can represent many DGMS

As we discussed in §8.9, an HMMs can represent many DGMs. Indeed, many DGMs can be flattened where the joint vector state space of the DGM is represented by a single monolithic integer. In some cases, there are advantages to doing this process. For example, the flattening can happen offline and within the flattened state space, any sparse structure can be represented by an optimized network during dynamic programming. That is, we can “remove the structure” from the DGM by representing the Cartesian product of all states using a single integer variable. For example, if (Q_t^1, Q_t^2) corresponds to a two-variable structured discrete state, this can be represented without loss of generality or specificity using $R_t = \ell Q_t^1 + Q_t^2$ for an appropriately large integer $\ell > 0$. This process is the essence behind weighted finite-state transducer minimization, determinization, and composition ([226, 198, 440, 308, 307, 306]) as mentioned in §8.4.4.5. This kind of process makes sense when the DGM is inherently hierarchical, where there are high-level processes that somehow control or guide lower level processes, as shown in Figure 8.72b.

Given that HMMs can represent so many DGMs, it is worthwhile to discuss a number of reasons when one should consider using DGMs, and when we should not use the flattened model, as we do in the next section.

8.12.2 Reasons for using DGMS rather than HMMs

HMMs can represent many DGMs. Of course, if a DGM cannot be flattened into an HMM (e.g., auto-regressive HMMs), then this is reason enough to use a DGM. In this section we give some reasons why one might wish to use a DGM even if it can be flattened into an HMM.

First, flattening a model means that the state space is no longer structured. This can in many cases lead to a greatly increased computational and memory cost of inference. For example, as we saw with a factorial HMM §8.9.1, flattening down to an HMM will increase the computational cost exponentially in the number of original factors [170].

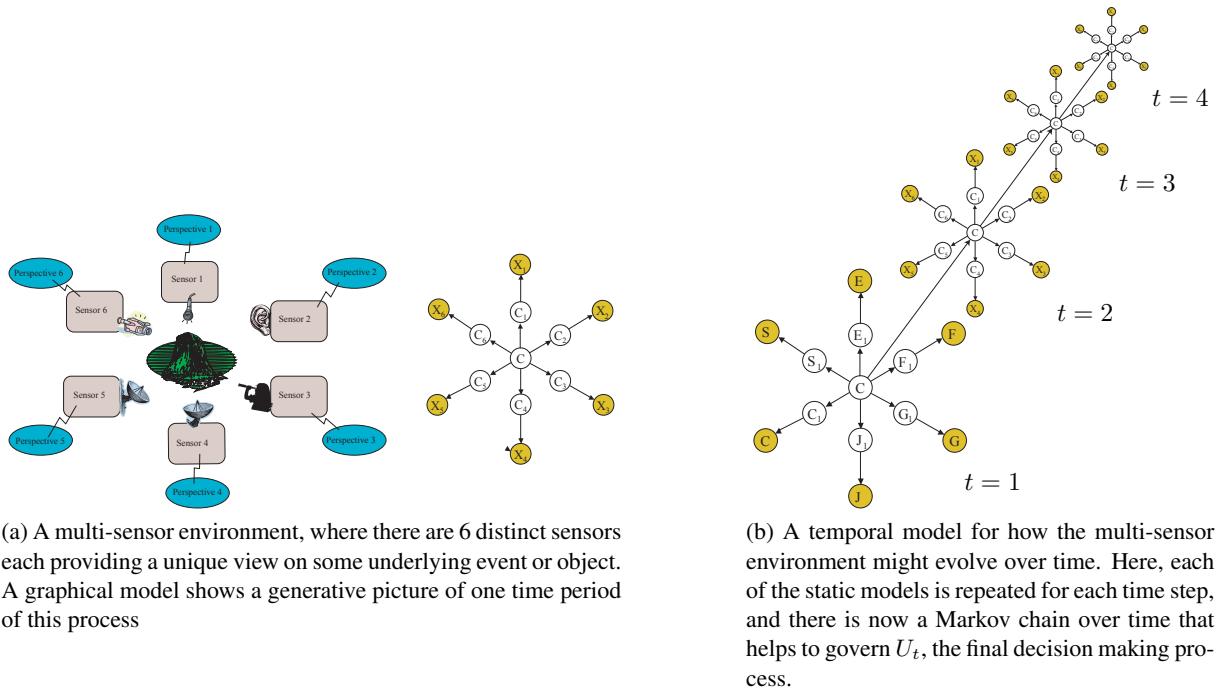


Figure 8.91: Sequence of stars for a multi-sensor temporal pattern recognition problem.

Another example is shown in Figure 8.91. On the left, we are in an environment where multiple different sensors are viewing an object and each sensor is giving a distinct view of the object. For example, the sensors could be multi-modal (auditory, visual, barometric pressure, etc.) and while the information provided by sensors might be partially redundant, each sensor provides a unique view of the object. Moreover, each sensor is involved in its own hierarchical model, and it is only at the highest level in each hierarchy is there a form of information fusion, where the high-level hypotheses from each sensor are combined into one unified hypothesis. The right of Figure 8.91 shows a temporal version of this model, where a first-order Markov chain helps to governs the evolution of the final high-level hypothesis, a model we call a *sequence of stars*. Hence, we see that the tree-width of this model is only one, but if we were to combine this into a single flattened HMM, there would be an enormous waste of compute. That is, the factorization in the sequence of stars would be lost if flattened down to a single HMM for a transducer. Indeed, such a star-shaped DBN would be hard for a transducer since the transducer would loose the decomposition along the star edges. The star model can be seen to have multiple simultaneous hierarchies, as simultaneous multiple hierarchical HMMs are coupled only at the very highest level — conditioned on the highest level in the hierarchy, the different hierarchical HMMs are rendered independent.

Indeed, with such multiple hierarchies, we see one of the inherent benefits of the graphical approach in

that any inference algorithm can continue to exploit such structure. WFST minimization, determinization, and composition would loose the hierarchical structure and hence could suffer from the same wasteful inference as would a flattened HMM.

Note, we will re-visit the notion of flattening again in Section 13.3.2.

A second reason for using a DGM over an HMM is that the DGM structure itself can be seen as a form of regularization. That is, during a training phase of the model, regardless of the loss function used, the structure acts as a set of constraints that insist on certain factorization properties being present in the model. These constraints would not be present during training of a flattened model, however. If one wanted to re-introduce such constraints during training of a flattened model, one could do so in the form of a constrained optimization procedure, but this is not superior to the original structure and its graphical representation.

Third, in a DGM there are often reusable modular components that repeatedly appear in various different forms. The length distributions described later in this article in Figure 8.79 is an example. By familiarizing oneself with these modular components, a model that looks excessively complicated at first is in fact quite simple. Once one is familiar with the underlying modular components, it is easy and efficient to discuss what, without the graphical approach, would be quite different techniques. For example, the original factorial HMM paper [170] gave an exact inference method that corresponds to one particular graph triangulation. The DGM approach avoids having to re-explain and re-derive dynamic programming algorithms for different models. There are in fact many different dynamic programming algorithms for a given model, and this is captured by the underlying triangulation of the DGM. Hence, one can give just the DGM and a triangulation and that would be sufficient to explain the underlying dynamic programming aspects of the model. We note here that since a DGM is inherently a temporal model, there will always be some form of dynamic programming used, even when performing approximate inference (see Chapter 13).

Forth, the inherently structured state space can be more natural for various domains. In the area of articulatory speech recognition, for example, where the state space consists of multiple streams of semi-synchronized human vocal-tract articulators [280], it makes much less sense to consider them in a flattened way. With a structured state space, certain probabilistic queries are easy to specify (e.g., in the above, we might wish $p(Q_t|\text{evidence})$ rather than $p(R_t|\text{evidence})$). Moreover, as with any graphical model, qualitative properties about and modeling assumptions within an application domain can be discussed, analyzed, and communicated efficiently .

8.12.3 Additional generic examples of DBNs/DGMs

Before moving on to inference, in this section we discuss a few additional examples of DGMs and DBNs in particular.

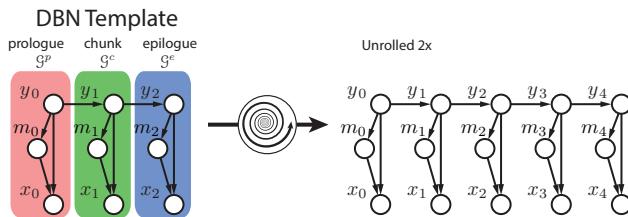


Figure 8.92: HMM with mixture observation distributions $p(x_t|y_t, m_t)$.

Our first example is a simple extension of the HMM that shows that the observation variables x_t have a mixture distribution, as seen in Figure 8.92. Here, we have at each time frame a mixture variable m_t that is influenced by the state y_t and that in turn influences the observation via $p(x_t|m_t, q_t)$. For example, if $p(x_t|m_t, q_t)$ was a Gaussian distribution for each value of m_t and q_t , then $p(x_t|q_t) = \sum_{m_t} p(x_t|m_t, q_t)$ is a Gaussian mixture distribution, leading to a Gaussian mixture HMM.

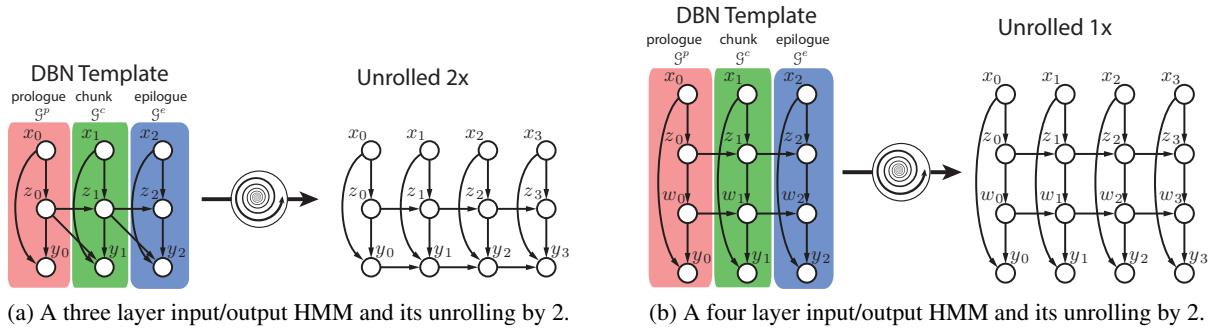


Figure 8.93: Input-Output HMMs.

Our next example is what has been called the Input/Output HMM (IO-HMM), a process that has an input sequence $x_{1:T}$, an output sequence $y_{1:T}$ and an intermediary Markov chain $z_{1:T}$ between the two. The first example is shown in Figure 8.93a where the output y_t is a function of both the input x_t and the Markov chain transition z_{t-1}, z_t . Hence, this IO-HMM is more of a Mealy/Jelinek style of transducer (§??).

A second example in Figure 8.93b shows a four-layer IO-HMM in the Moore/Rabiner style, where the output is a function of the input and the current state. Of course, one can imagine a number of variations of this model, depending on an application's need.

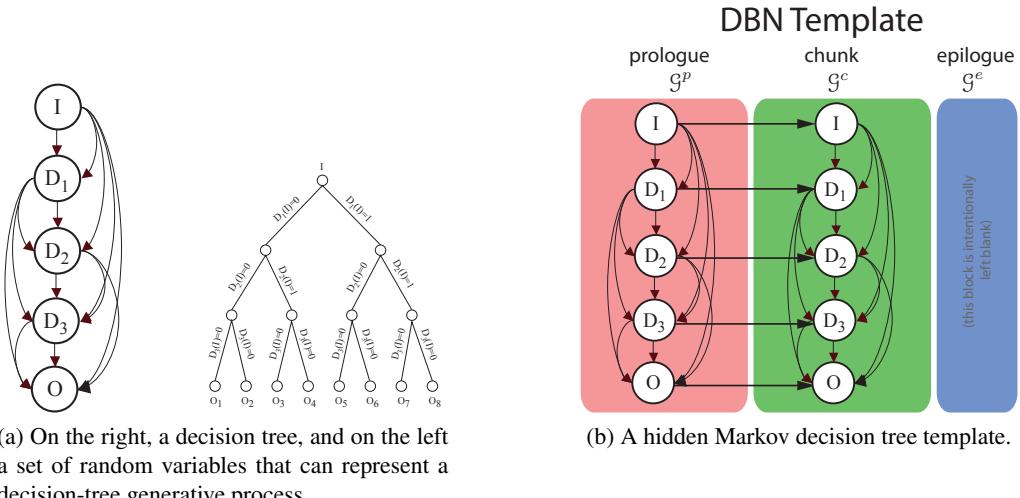


Figure 8.94: Hidden Markov Decision Tree.

Our next example shows what has been known as hidden Markov decision trees. That is, there is a generative model governed by a decision tree, where the decision made at each node in the tree is based on the input and also the sequence of decisions made before that node up to the root node of the decision tree. Such decision trees, at a given time frame, can be represented by a structure as shown on the left in Figure 8.94a. That is, D_1 is a random variable with conditional distribution $p(D_1|I)$ where I is the input. D_2 is a random variable with conditional distribution $p(D_2|D_1, I)$, and hence any decision made by D_2 is based both on the input I and the decision made at D_1 . This continues down until the output. The temporal version of this model, the hidden Markov decision tree, is shown in Figure 8.94b.

Our next example, Figure 8.95, shows a mixed memory Markov model. In fact, there are two Markov chains, $Q_{1:T}$ and $R_{1:T}$ both of which contribute to explaining the observations $X_{1:T}$. What makes this a

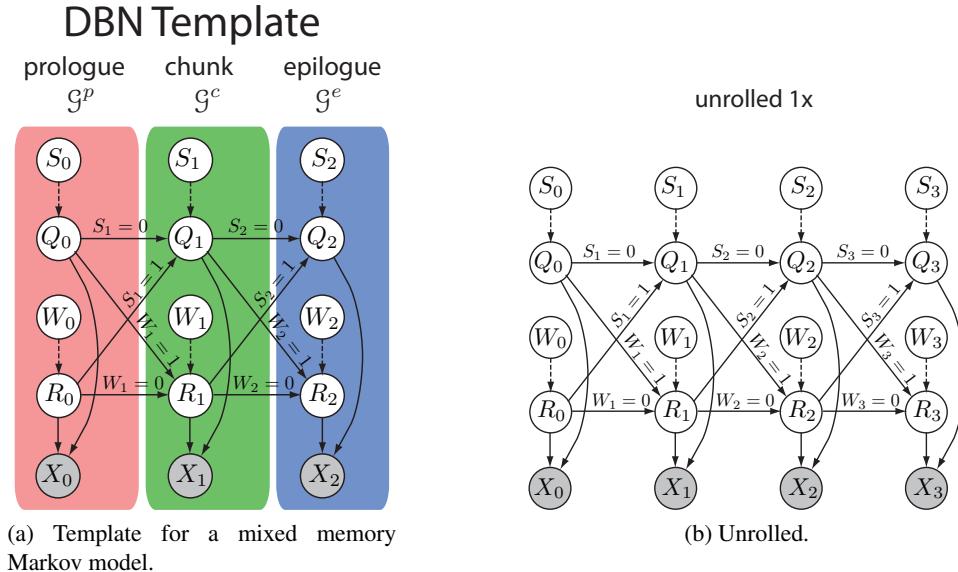


Figure 8.95: Mixed memory Markov model.

“mixed memory” model is that there are switching parents S_t and W_t at each time that determine how the Markov chains interact with the variables around them. As can be seen, Q_t has two Markov chain parents (ignoring the parent S_t for now), Q_{t-1} (as in a normal 1st order Markov chain) and R_{t-1} . Ordinarily, this would require a three-dimensional table $p(Q_t|Q_{t-1}, R_{t-1})$. Here, there is a binary-valued switching parent S_t which determines which of Q_t ’s parents are active at the moment. That is, we have:

$$p(q_t|q_{t-1}, r_{t-1}) = \sum_{s_t} p(q_t|s_t, q_{t-1}, r_{t-1})p(s_t) = p(q_t|s_t = 0, r_{t-1})p(s_t = 0) + p(q_t|s_t = 1, r_{t-1})p(s_t = 1) \quad (8.504)$$

Hence, rather than one three-dimensional table, we use two two-dimensional tables, something that can both lead to a substantial savings in memory and can offer a significant degree of parameter regularization during training.

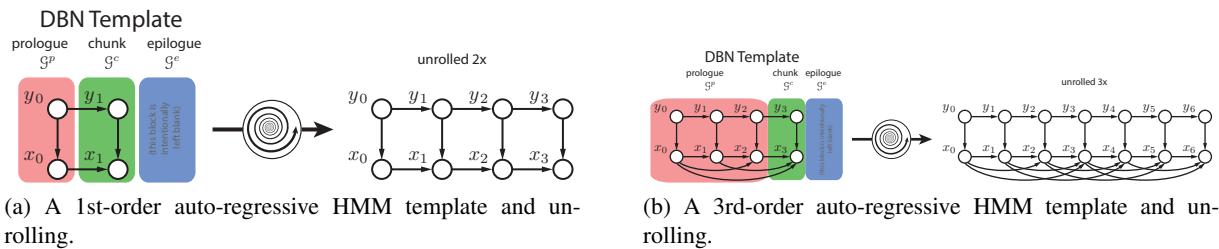


Figure 8.96: Auto-regressive HMMs.

Our next model is the auto-regressive HMM (or AR-HMM)¹⁶. An AR-HMM(K) as an HMM but with additional edges pointing from parent observations $X_{t-\ell}$ to the child observation X_t for $\ell = 1, \dots, k$ and

¹⁶In this paper, AR-HMM stands for “auto-regressive HMM.” An AR-HMM can of course use any, possibly even non-linear, implementation of the dependencies between the observations. Note that here AR-HMMs refers to an auto-regression over feature vector elements, similar to [235]. These AR-HMMs are not to be confused with the auto-regressive, linear predictive, or hidden filter HMMs in the past [344, 345, 224, 347]. These latter models are HMMs that have been inspired from the use of linear-predictive coefficients directly on the sampled speech signal itself.

for all t (See Figure 8.96).

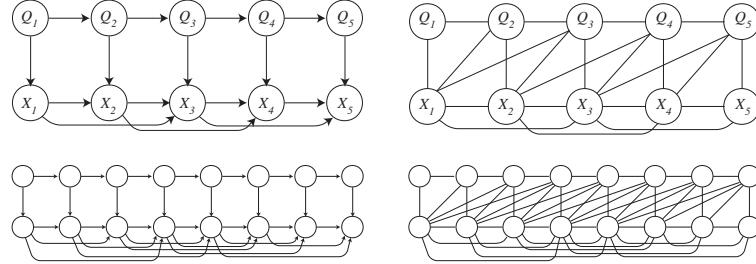


Figure 8.97: Bayesian network for an AR-HMM(2) (upper left) and its triangulated-by-moralized undirected graph (upper right). The same is shown for an AR-HMM(3) (lower left and lower right respectively). For the right graphs in this figure, one can see that all cycles (the left of X_t or Q_t) of length four or greater and that contain either X_t or Q_t (or both) will contain a chord.

One might wonder if adding all these edges to the AR-HMM relative to the HMM makes inference cost prohibitive due to the increase in clique sizes. The next set of theorems shows that the underlying computational complexity of the AR-HMM(K) is no worse than that of a standard HMM since the state space does not increase.

Theorem 144. A moralized AR-HMM(K) is already triangulated.

Proof. This is shown by induction. The case where $K = 1$ is clear, so assume it is true for $K - 1$ and then show it for K . An AR-HMM(K) can be viewed as an AR-HMM($K - 1$) with additional edges added between the parent X_{t-K} and child X_t for all t . Consider a cycle in an AR-HMM(K). If the cycle contains edges that are entirely contained in the subsumed AR-HMM($K - 1$), then it contains a chord by induction. The new edges that exist in a moralized AR-HMM(K) not contained in an AR-HMM($K - 1$) are the edges between X_{t-K} and X_t , and the edges between X_{t-K} and Q_t . Consider a cycle of length greater than 3 that contains at least one of these new edges, and consider the right-most node of that cycle. If the right most node is X_t (or there are two right-most nodes X_t and Q_t), then consider the two nodes adjacent to X_t in that cycle. These nodes must come from the set $\{Q_t, X_{t-1}, X_{t-2}, \dots, X_{t-K}\}$. But any two of these nodes are connected, either via the edges of the moralized AR-HMM($K - 1$), or via the edge (X_{t-K}, Q_t) . Therefore, the set forms a clique and therefore contains a chord. Suppose, on the other hand, the right-most node of the cycle is Q_t (meaning that X_t is not in the cycle). In this case, the edges of the cycle adjacent to Q_t either come from the AR-HMM($K - 1$) and contain a chord, or one edge of the cycle adjacent to Q_t is (X_{t-K}, Q_t) . But in this case, X_{t-K} is adjacent to every other node adjacent to Q_t in the cycle, so there is a chord. \square

Note that this theorem can also be proven using Theorem 4.5 in [98]. While there are many ways to triangulate a graph, the graph resulting from the above approach is called a *triangulated-by-moralized AR-HMM(K)*. It remains to be shown that the state space of the triangulated-by-moralized AR-HMM(K) is no larger than that of an HMM, leading to the following theorem:

Theorem 145. A triangulated-by-moralized AR-HMM(K) has no clique with more than two hidden variables

Proof. In an AR-HMM(K), no node has more than one hidden variable as a parent, all the other parents are observed. Therefore, moralization does not add edges between hidden variables. The remaining clique variables are observations, so the state-space size is only N^2 . \square

Like with an HMM, a triangulated-by-moralized AR-HMM(K) has only $O(T)$ cliques. Therefore, the complexity of an AR-HMM(K) is again only $O(TN^2)$ for any fixed K . There is, however, a constant cost associated with the number of additional dependency edges. Incorporating this cost, the complexity becomes $O(TN^2K)$ where K is the maximum number of dependency edges per observation.

It is important to realize that the above theorems rest on X_t being observed random variables. If it was the case that some or all of X_t were hidden, complexity would significantly increase. In such cases, one would probably need to resort to approximate inference methods [98].

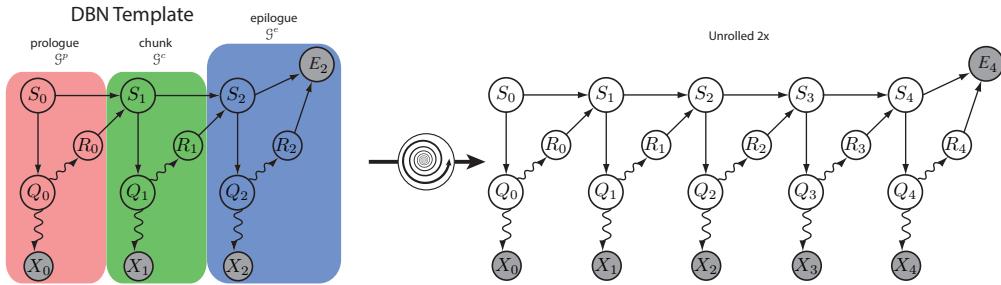


Figure 8.98: The basic triangle structure for stochastic sequencers.

Our next DBN is shown in Figure 8.98. Although the model it is generally applicable to a variety of problems, as given, it describes a problem in speech recognition. The essential idea is that one wants an HMM but where the Markov chain is such that certain states share observation distributions. That is, if Y_t is the state random variable and X_t is the observation, we should have it that for certain $i \neq j$, $p(X_t|Y_t = i) = p(X_t|Y_t = j)$.

One way to do this is to collapse all such states into a single state which uses the desired observation distribution. This is not always appropriate, however, as it can be that different states, even if they share the same observation distribution, do not share the same contexts. The classic example of this is in speech recognition where a given phone (corresponding to an HMM) might be used within different words, and the phone in different words can easily be surrounded by different phones (this is even true within a single word, “banana” for example). Another option is to keep each state distinct, and then copy the parameters for each such desired shared observation distribution. While this would work for inference, any constraints keeping the parameters would not be there, and the parameters corresponding what should be the same observation distribution would diverge from each other.

The triangle structure in Figure 8.98 addresses this issue explicitly through the graph structure.

Firstly, there is significant determinism in this graph in the form of a deterministic CPT. A deterministic CPT over a set of random variables $p(a|b, c, d)$ is one that has the following form:

$$p(a|b, c, d) = \begin{cases} 1 & \text{if } a = f(b, c, d) \\ 0 & \text{else} \end{cases} \quad (8.505)$$

where $f(b, c, d)$ is a deterministic (i.e., non-random) function of its arguments. This means that for a given b, c, d there is one and only one value of a that has any and all the probability, namely $f(b, c, d)$. Of course, a deterministic CPT can correspond to any number of parent variables. Also note that a deterministic CPT is different from sparsity in a general factor — in a deterministic CPT, there is one variable that is guaranteed to be determined given values for the other variables. In a factor ϕ , say over variables a, b, c, d , then there will be a collection of values of the variables that make the factor zero $\{(a, b, c, d) : \phi(a, b, c, d) = 0\}$.

Figure 8.98 uses deterministic CPTs in the following ways. Note that wavy edges mean the child variable is random, while the non-wave edges indicate the child variables are a deterministic function of its parents. First, the variable S_t is really a counter that counts up starting from zero, but it only increments its

previous value when $R_t = 1$. That is,

$$p(S_t = i | S_{t-1} = j, R_{t-1} = k) = \begin{cases} 1 & \text{if } i = j + 1 \text{ and } k = 1 \\ 1 & \text{if } i = j \text{ and } k = 0 \\ 0 & \text{else} \end{cases} \quad (8.506)$$

We can think of R_{t-1} as a trigger that allows the sequence variable S_t to increment, and otherwise S_t is just a copy of S_{t-1} . Next, the variable Q_t is a deterministic function of S_t . The idea is that the position within the model, indicated by S_t is used to index into an actual state value Q_t .

$$p(Q_t = i | S_t = j) = \begin{cases} 1 & \text{if } i = f(j) \\ 0 & \text{else} \end{cases} \quad (8.507)$$

where $f : D_{S_t} \rightarrow D_{Q_t}$ is a deterministic function that maps from the domain of S_t to the domain of Q_t . Hence, different positions are allowed to have the same underlying “state” by having f map different positions to the same state value. The variable X_t is random (indicated by a wavy edge) — the key thing is that the distribution of X_t is governed by its parent Q_t and not by the position variable S_t . Lastly, the delta variable R_t is a random function of the real state Q_t — the idea is that the state itself should determine the length distribution on how long we should stay in that state. The position does not have a say in the length of the state. The final variable E is observed and is also deterministic. It ensures that the model gives zero probability to any sequence that does not end with a transition out of the final position. That is, if say 10 was the final position of the model, then one possible implementation of the distribution of E would be:

$$p(E = 1 | S_T = i, R_T = j) = \begin{cases} 1 & \text{if } i = 10 \text{ and } j = 1 \\ 0 & \text{else} \end{cases} \quad (8.508)$$

This triangle model can be used for single word speech recognition, where S_t is the phone position of a word (i.e., how many distinct phones into a word are we at time t), Q_t is the phone of a given word, X_t is the distribution on acoustics given the state, an R_t is the phone length (which is geometrically distributed).

This basic structure is a widely used (modular) component in a variety of DBNs (cf. Chapter 9), so it is important to understand it. Indeed, there are other modular DBN components that, once understood individually, make understanding larger and very complex looking models much easier. Also note, deterministic CPTs are extensively used in DBNs, as shown in particular in Chapter 9.

In summary, there are many types of DBNs that express much more than either the HMM or the simple factorial models listed in §8.9. Some other ones are listed in [107, 312, 373, 249]. Note that the above models are all relatively generic examples of DBNs. That is, the DBNs in this section were developed for specific purposes but they are universally applicable in a variety of domains. In later sections, we will discuss instances of DBNs that focus on particular applications §9.1.

8.13 Structured Prediction in Machine Learning

In binary pattern classification, we map from a vector of features $x \in \mathbb{R}^N$ to a binary label value $y \in \{0, 1\}$. In multi-class pattern classification, we map from features $x \in \mathbb{R}^N$ to $y \in \{0, 1, 2, \dots, K\}$, a non-negative integer.

In an HMM or standard linear chain CRF, we map from length T sequence of vectors $x \in \mathbb{R}^{N \times T}$ to sequence of labels $y \in \{0, 1, \dots, K\}^T$. Moreover, different lengths of x correspond to different lengths of y (i.e., T can vary from sample to sample but both x and y within any sample have the same length). In a DGM, we map from a sequence of vectors $x \in \mathbb{R}^{N \times T}$ to a length T sequence of vectors $y \in \mathbb{R}^{M \times T}$,

where again different length x map to diff length y . A discrete DGM uses only integer labels, so here $y \in \{0, 1, \dots, K\}^{M \times T}$. In some sense, therefore, HMMs, CRFs, and DGMs can be considered to be “structured” in the sense that the label that is being mapped onto is modular in a certain way — it consists of component parts which can be considered both individually (i.e., a y_t for a particular $t \in \{1, 2, \dots, T\}$) or in a group (i.e., the vector (y_1, y_2, \dots, y_T)). Hence, even an HMM can be considered as a form of “structured prediction” as any method to do sequence prediction would never consider the sequence apart from its component parts. Being able to think of y ’s modular components is important for being able to deal with such objects in practice. For example, if we wish to produce a prior distribution over $y \in \mathcal{D}_Y$, we can do so by decomposition. Indeed, an HMM does exactly that, by using as a prior distribution a Markov chain. That is, $p(y) = \prod_{t=1}^T p(y_t | y_{t-1})$ which is obtained via the HMM factorization assumptions.

In many other cases, however, x might be more structured still, like pixels of an image, and y a set of labels for each pixel. May Markov random field models in computer vision, indeed, are really two-dimensional forms of HMMs, and the view of such models as smoothed (or regularized) unary potential functions (§8.4.8) in computer vision is quite common. Alternatively, x could be a variable-length sentence (sequence of words) and $y \in \mathcal{D}_x$ where \mathcal{D} is a set of possible parses of the sentence x . For example, there are many formal grammars for language (e.g., probabilistic context free grammars, and many others) In this case, given two $y^1, y^2 \in \mathcal{D}_x$, they might have very different properties or very different underlying sizes. The variable x could be a indicators of a set of N individuals, or could be behavioral observations about such individuals such as what they did on a given day — y , in this case, might be a graph (e.g., social network) over those individuals, and so the space of y would consist of all possible graphs over N vertices. It could even be that x is a set of behavioral observations about an unknown set of individuals and y could lie within the space of all possible graphs over an unknown number of vertices. Generalizing even further, x could be a sequence of observations over individuals, meaning each x_t would be a set of observations at time t about N individuals — y_t , then, is a graph over N vertices, that might change over time (a dynamic social network), and y is a sequence of social networks. In general, therefore, \mathcal{D}_X and \mathcal{D}_Y need not even be a finite (or a countable) sized class — these could range over any infinite sized set of objects that one is interesting in deducing given some evidence.

This last set of problems are most generally known as “structured prediction” problems in machine learning. In some sense, as mentioned above, even an HMM is “structured” in that the output is not a monolithic single scalar integer. However, in more recent times the notion of “structured” has become even more structured than just a sequence, as the output can be graphs, trees, matrices, and so on. How can one build and learn a classifier over such domains where each training samples might be very different and have a very different size than any other training sample?

The basic approach people take is one of feature functions, which transform objects of variable size and their labels (also of variable size) into a fixed vector length description of that object. That is, rather than associate an individual x a given individual y , we associate “features” (“attributes”, or “aspects”) of the x s and y s. The size of the object itself might even be a feature of the object. For any of the above scenarios, lets say that $x \in \mathcal{D}_X$ and $y \in \mathcal{D}_Y$. Feature functions take the form $f : \mathcal{D}_X \times \mathcal{D}_Y \rightarrow (\mathbb{R}_+ \cup \{\infty\})^K$ where K is the length of a non-negative feature vector that is used to describe a given input-output pair. That is, for $x \in \mathcal{D}_X$ and $y \in \mathcal{D}_Y$, $f(x, y)$ is fixed-length vector of features that describe x and y together. With each $f(x, y)$ itself lying in a finite sized set, then this breaks $\mathcal{D}_X \times \mathcal{D}_Y$ down into equivalence classes $(x, y) \in \mathcal{D}_X \times \mathcal{D}_Y : f(x, y) = \hat{f}$. In such case, rather than seeing the object as itself, we see it only as one of the equivalence classes. This is in many cases a necessary tradeoff since otherwise the variable size of objects within \mathcal{D}_X and \mathcal{D}_Y become can not be dealt with — to deal with such objects, we view them through the lens of finite length objects which inevitably will lead to some distortion.

Hence, we have turned x, y from a variable length set of objects with a variable length size (i.e., each x, y can not in the original form judged with a fixed length set of parameters) into a fixed length set of parameters. We can also derive scores for a given x, y pair. Using parameters $\lambda \in \mathbb{R}^K$ we score the pair

as $\lambda^\top f(x, y)$, which can be seen as an energy function. With such a feature vector, moreover, it is possible to derive probabilistic models over structured data. I.e., given a vector $\lambda \in \mathbb{R}^K$, of parameters define the following joint model:

$$p_\lambda(x, y) = \frac{1}{Z} \exp(-\lambda^\top f(x, y)). \quad (8.509)$$

Hence, we can use log-linear models for structured data. Since f might be infinite valued, it is also possible to represent pairs x, y as having zero probability (when $f(x, y) = \infty$). With this, one can derive learning criteria (such as maximum likelihood, conditional likelihood, max-margin, and so on). The success of a learning method depends on: 1) how good the set of features f are at describing important properties of the objects x, y ; 2) how appropriate the learning objective is to the goal of learning; 3) how approximate any inference method must be in order to produce a computationally feasible algorithm for the learning process; 4) how computationally feasible any exact or approximate probabilistic inference method is for computing probabilities $p(x, y)$.

It is possible to describe HMMs, CRFs, DBNs, and DGMs using this log-linear framework, which is quite general. Indeed, DGMs, with a fixed amount of unrolling, is only a finite-state model after all. More generally, however, structure prediction need not be finite state. Given this more general framework, inference methods techniques can not be as specific as they can for DGMs alone. Hence, in this article, we acknowledge the more general framework of structured prediction and that in many situations it is necessary, but we also must limit ourselves in our discussion to those models that can be well expressed using DGMs. GMTK, moreover, deals with dynamic graphical models where the state space, per unit time, is finite. Hence, we do not further consider the more general non-finite state structured prediction tasks and refer the reader to the text [11].

8.14 The problem with the Viterbi/MPE/MAP assignment in DGMs

In §8.4.11, we discussed the Viterbi algorithm, also known as MAP or MPE decoding. As a reminder, the goal is as follows: given an HMM $p(\bar{x}_{1:T}, y_{1:T})$, compute

$$y_{1:T}^* \in \underset{y_{1:T}}{\operatorname{argmax}} p(\bar{x}_{1:T}, y_{1:T}) \quad (8.510)$$

for a given set of observations $\bar{x}_{1:T}$.

Lets now consider the hierarchical HMM in Figure 8.72b will serve our purposes in this discussion. The issue we are about to describe can happen in many models, but we will use Figure 8.72b to describe it. Lets suppose, moreover, that $M = 2$ (two levels in the hierarchy). Viterbi decoding then becomes computing:

$$(y_{1:T}^{1*}, y_{1:T}^{2*}) \in \underset{y_{1:T}^1, y_{1:T}^2}{\operatorname{argmax}} p(\bar{x}_{1:T}, y_{1:T}^1, y_{1:T}^2) \quad (8.511)$$

Hence, if it is the case that the final goal of classification is to use the highest level objects $y_{1:T}^2$, and the secondary objects $y_{1:T}^1$ correspond to sub-parts of the primary objects, then any decision on the primary objects with Viterbi decoding is based on considering the most probable joint labeling $(y_{1:T}^{1*}, y_{1:T}^{2*})$. Thus, the final $y_{1:T}^{2*}$ decided in Viterbi decoding might not be the most probable primary object at all and in fact could be quite different than the most probable primary object. The fundamental issue is that $y_{1:T}^{2*}$ above can be quite different than the below computation:

$$(y_{1:T}^{2+}) \in \underset{y_{1:T}^2}{\operatorname{argmax}} p(\bar{x}_{1:T}, y_{1:T}^2) = \underset{y_{1:T}^2}{\operatorname{argmax}} \sum_{y_{1:T}^1} p(\bar{x}_{1:T}, y_{1:T}^1, y_{1:T}^2). \quad (8.512)$$

The issue is that the most probable $y_{1:T}^{2+}$ could have numerous compatible $y_{1:T}^1$ values. By “compatible”, we mean an assignment that does not give zero probability. Each such pair might be much less probable than the Viterbi pair, meaning:

$$p(y_{1:T}^{1*}, y_{1:T}^{2*}) \gg p(y_{1:T}^{1+}, y_{1:T}^{2+}) > 0 \quad (8.513)$$

where $y_{1:T}^{1+}$ is any assignment to $y_{1:T}^1$ that is compatible with $y_{1:T}^{2+}$. But the number of such $y_{1:T}^{1+}$ could be exponentially (in T) more numerous than the number of Viterbi assignments, so that we get:

$$p(y_{1:T}^{2*}) \ll \sum_{y_{1:T}^{1+}} p(y_{1:T}^{1+}, y_{1:T}^{2+}) = p(y_{1:T}^{2+}) \quad (8.514)$$

This is a well known problem, namely that Viterbi decoding is not maximum likelihood decoding, and can have real consequences in sequential modeling.

For example, in automatic speech recognition, one typically has the highest level of the model y_t^2 correspond to words and a middle level of the model y_t^1 correspond to pronunciations of such words. There might be words (such as “and”) that have very many different possible pronunciations in real conversational speech, and other words that have relatively few. However, whenever there is a large number of pronunciations, each such pronunciation has lower probability, and this lower probability is used in the Viterbi score (there is no summing over the numerous pronunciations). The Viterbi assignment, then, might choose the wrong word, one having a higher score only because it has fewer pronunciations each of which has higher probability, but when considered all together a lower probability.

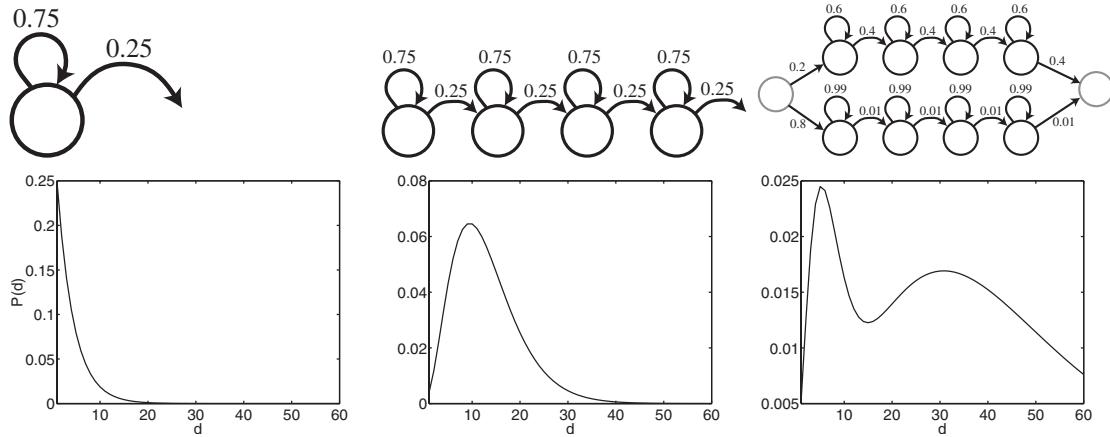


Figure 8.99: Three possible duration distributions produced by summing together geometrically distributed random variables. The effect of this can be achieved in an HMM with tied observations and particular Markov chain topologies. In each of the above cases, the states shown have tied observation distributions to achieve these duration distributions for the time during which the observation distribution is active in any given random sample of an HMM.

Another case where this can cause problems is in duration modeling. In a Markov chain, the time duration D that a specific state i is active is a random variable distributed according to a geometric distribution with parameter a_{ii} . That is, D has distribution $P(D = d) = p^{d-1}(1 - p)$ where $d \geq 1$ is an integer and $p = a_{ii}$. An HMM in its pure form has state duration distributions are inherently geometric, and geometric distributions do not always accurately represent typical length distributions of real segments of objects (and this is true in speech, biology, and many other sequential problems).

An HMM using “state-tying”, where multiple different states can share the same observation distribution, can induce other forms of distributions. If a sequence of n states using the same observation distribution

are strung together in series, and each of the states has self transition probability α , then the resulting distribution is equivalent to that of a random variable consisting of the sum of n independent geometrically distributed random variables. The distribution of such a sum has a negative binomial distribution (which is a discrete version of the gamma distribution) [403]. Unlike a geometric distribution, a negative binomial distribution has a mode located away from zero, as shown in Figure 8.99.

The problem arises in Viterbi decoding. Consider the Viterbi path in an HMM. Each such path through the hidden states is a product of scores, and each such product score has the same number of factors. In other words, regardless of the duration model, the Viterbi path has a score of the following form:

$$p(y_{1:T}^*, \bar{x}_{1:T}) = \prod_t p(x_t|y_t^*) \prod_t p(y_t^*|y_{t-1}^*). \quad (8.515)$$

In particular, the score of the path (state assignment) itself is $\prod_{t=1}^T p(y_t^*|y_{t-1}^*)$ which is a product of T factors. Lets consider this in the context of the three duration distributions in Figure 8.99 and consider the the maximum score of the product for various T . That is, consider the following construct

$$\text{length-score}(T) = \max_{y_{1:T}} \prod_{t=1}^T p(y_t^*|y_{t-1}^*) \quad (8.516)$$

We see that if all the observations were agnostic about the state, the length score would determine the Viterbi path, and the degree to which the state duration affects the Viterbi score, the length-score is the score of the path with highest value.

Lets consider the length score for each of the cases in Figure 8.99, On the left, it is clear that the highest score occurs when we stay in the same state for all T , leading to the score $0.75^T 0.25$ — this is not surprising since there is only one state in the model. Hence, the length score has a shape as given by the plot on the lower left.

In the middle of Figure 8.99, the highest score of length T would be of the form $0.75^{T-3} 0.25^4$ for $T \geq 4$ and 0 for $T < 4$. Since this is monotonically decreasing for $T \geq 4$, the highest possible length score occurs when $T = 4$, which means that rather than preferring lengths as shown in the plot in the middle, length-score(T) most prefers lengths of four, and has monotonically decreasing preference for longer lengths.

On the right of Figure 8.99, the highest score of length T has score $0.8 \times 0.99^{T-4} \times 0.1$ for $T \geq 5$ and zero for $T \leq 5$. the highest length score occurs for $T = 5$ and is monotonically decreasing thereafter. Like in the middle case, this distribution most prefers Viterbi paths that have length five.

Why is there such a disconnect between the length distributions given in Figure 8.99 and the lengths preferred by the Viterbi path? The reason is the maximization operation, namely that we are computing the Viterbi score and are not summing over all possible paths of a given length, as would be the case if we were to perform an operation similar to Equation (8.512). Another example of this issue comes about in figure Figure 8.79, which is further described in Chapter 11.

Unfortunately, doing mixed max/sum decoding as in Equation (8.512) is often intractable, and we discuss this further in §13.5.

Part IV

Applications of Dynamic Graphical Models

Chapter 9

Dynamic Graphical Model Examples for Automatic Speech Recognition

While these are meant for speech, many of the concepts are quite general and could be used for NLP and bio. It is therefore strongly recommended that regardless of your application, you read through the description of each of these models.

9.1 Graphical Model Speech Architectures

In this section, we describe a number of graphical architectures for various speech recognition decoder constructs. Note that each of the graphs presented are meant only for speech recognition "decoding" (meaning, producing a sequence of words from the acoustic waveform). In general, the graphical architectures used when training the parameters of a speech recognition system will be slightly different (and will include sub-graph structures for constructs such as counters, sequencers, lattices, and so on). Each of the following graphs falls under one or more of the following broad goals for using graphs for ASR.

The first goal for graphical models in ASR is to design structures that explicitly and efficiently represent typical or novel ASR control constructs. These may include features such as parameter tying, the sequencing of states through a list or lattice, smoothing by mixing, the order n in an n -gram language model, the size of the context in context-dependent phone models (e.g., mono- or tri-phones), the type of this context (within-word or cross-word tri-phones), single or multiple pronunciations per word, and so on. Any given ASR system might or might not implement some of these features (e.g., System X supports bi-grams but not tri-grams). Within the graphical models framework, however, essentially all features are available (given the appropriate general and efficient software). Such models, pioneered by [468], are now called "explicit graph representations" [38], where random variables can exist for such purposes as word identification, numerical word position, phone or phoneme identity, the indication of a phone transition, a count of the number of phones, and direct copies of variables from other time frames. There indeed can be multiple possible representations of the same underlying control structure, but these representations might have different degrees of conceptual simplicity and/or computational requirements. It is therefore imperative to design structures that are both simple to explain and fast to run. Explicit models stand in contrast to the fully "implicit approach", where much of this control information is represented by a single hidden Markov chain with a large sparse transition matrix, and where an integer state encodes all contextual and control information determining the allowable sequencing. The implicit approach, in fact, corresponds roughly to the result of flattening (e.g., composition and minimization [307]).

The implicit and explicit approaches are useful in different circumstances. The explicit approach allows for the representation of an unlimited number of constructs. Moreover, when the underlying set of hidden variables must have factorization constraints, it is natural to use an explicit model. If the underlying model

consists of multiple hierarchies, the explicit approach is again quite natural. On the other hand, if no factorization is needed, if the CPFs are often sparse (containing many zeros), and if the underlying operation is always a single hierarchical composition of multiply-nested Markov chains, then the implicit approach can be both natural and very fast [307].

A second goal in forming graphical models for ASR is in latent knowledge modeling. Here, a set of (partially) factored dynamic hidden variables are used to represent any unknown but presumed to exist phenomenon (or “cause”) of the speech signal. This includes knowledge from very high linguistic levels to the very low acoustic level. For example, hidden variables can represent dialog act, word or phrase category, pronunciation variant, speaking rate, mode/style/gender, word morphology, vocal-tract or articulatory configuration, acoustic channel condition, noise condition, or Gaussian component. Knowledge modeling using graphical models is a promising area for further research as many believe that high-level knowledge is underutilized in existing state-of-the-art speech recognition systems. Graphical models can provide the infrastructure in which this knowledge can be exploited.

A third goal consists of proper *observation modeling*. Most speech recognition systems represent speech as a sequence of feature vectors corresponding to overlapping windows of speech. The feature vectors (most often Mel-frequency cepstral coefficients, or MFCCs) are then represented by state-conditional multivariate Gaussian mixtures. It is quite easy to extend such a representation by allowing relationships between individual elements of the current feature vector and elements in other feature vectors either before, during, or after the current one. Such a model can more appropriately match the underlying statistics extant in speech as represented by the current series of feature vectors.

Lastly, the most challenging problem (involving all three of the above) is that of automatic structure learning. Here, the problem is to automatically determine a best graph structure (or a set of good structures) using the available training data [45]. This problem is particularly difficult when learning hidden structures, as not only is data for the hidden structure unavailable (making it difficult to learn), the existence of each such hidden variable can be put into question. Therefore, many standard statistical model selection techniques are inappropriate.

In the next several sections, we will see how these four goals arise when we examine a number of different graphs for representing the speech recognition process. We will start with a fairly basic graph (for a bi-gram decoder) and expand on this idea to the much more elaborate.

9.1.1 Basic phone-based bi-gram decoding structure

Our first model is a simple phone-based decoder that uses bi-gram language models and single-state phones (Figure 9.1). The figure corresponds to a GMTK template unrolled one time: the first frame is the prologue \mathcal{G}^p , the second and third frames are each a chunk \mathcal{C} , and the last frame is the epilogue \mathcal{E} . The structure explicitly represents the unflattened and hierarchical constructs that go into building such an ASR system with these attributes even though, when flattened, it can be represented by an HMM. Each such attribute is implemented using a separate temporal layer of random variables in the graph. We define the following random variables at time t : \mathcal{W}_t is the word, \mathcal{W}_t^{tr} is the word transition, \mathcal{W}_t^{ps} is the word position, \mathcal{P}_t^{tr} is the phone transition, \mathcal{P}_t is the phone, and \mathcal{O}_t is the acoustic feature vector. The graph thus specifies the following factorization for a length T utterance:

$$\begin{aligned} & p(\mathcal{W}_{1:T}, \mathcal{W}_{1:T}^{tr}, \mathcal{W}_{1:T}^{ps}, \mathcal{P}_{1:T}^{tr}, \mathcal{P}_{1:T}, \mathcal{O}_{1:T}) \\ &= \prod_t p(\mathcal{O}_t | \mathcal{P}_t) p(\mathcal{P}_t^{tr} | \mathcal{P}_t) p(\mathcal{P}_t | \mathcal{W}_t^{ps}, \mathcal{W}_t) p(\mathcal{W}_t^{tr} | \mathcal{W}_t, \mathcal{W}_t^{ps}, \mathcal{P}_t^{tr}) p(\mathcal{W}_t^{ps} | \mathcal{W}_{t-1}^{tr}, \mathcal{W}_{t-1}^{ps}, \mathcal{P}_{t-1}^{tr}) p(\mathcal{W}_t | \mathcal{W}_{t-1}, \mathcal{W}_{t-1}^{tr}) \end{aligned}$$

Note, however, that it is always the case that $\mathcal{W}_T^{tr} = 1$ to ensure that a proper final word is decoded. Some of the CPFs are deterministic (represented by straight lines in the figure, see two paragraphs below for

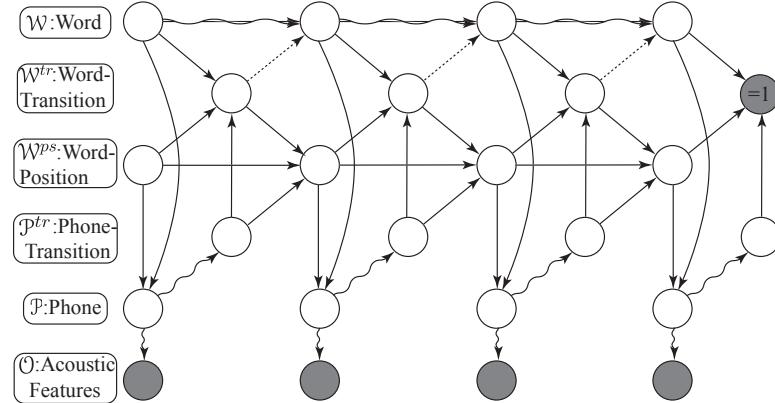


Figure 9.1: Explicit representation of a phone-based ASR decoder with a bi-gram language model. Observations are depicted as shaded nodes, and hidden variables are unshaded. Also, deterministic dependencies are represented by straight lines, and purely random ones are represented by wavy lines. If a line is both straight and wavy, it switches between deterministic and random based on a switching parent, connected via a finely-dashed edge. Each time frame has (from top to bottom) a variable for: word identity, word transition, word position (position of current phone within a word), phone transition, phone, and acoustic feature vector. Note that the final word transition binary variable is observed to equal the value 1, thus disabling any decodings that do not end in a complete word.

further explanation) and some are purely random (represented by zig-zagged or wavy lines). To compute the probability of the observations, we must perform the sum:

$$p(\mathcal{O}_{1:T}) = \sum_{W_{1:T}, W^{tr}_{1:T}, W^{ps}_{1:T}, P^{tr}_{1:T}, P_{1:T}} p(W_{1:T}, W^{tr}_{1:T}, W^{ps}_{1:T}, P^{tr}_{1:T}, P_{1:T}, \mathcal{O}_{1:T})$$

and then exploit the factorization property as specified by the graph to best distribute sums into products to reduce computation. It should be noted that in this graph, many of the terms in the sum will have value zero since one or more of the factors at each time frame will themselves be zero. This is because many of the factors correspond to CPFs that are deterministic. One of the goals of inference in such graphs, therefore, is to take advantage of this sparsity in order to avoid the unnecessary computation of summing multiple zeros together which takes time but does nothing.

We next describe each individual CPF. First, the acoustic features \mathcal{O}_t are a random function of the phone in the CPF $p(\mathcal{O}_t | \mathcal{P}_t)$. As is typical in an HMM system, each value for \mathcal{P}_t will select a particular Gaussian mixture to be used to produce probability scores for that phone. Next, the phone transition variable P_t^{tr} is a binary indicator that specifies when the model should advance to the next phone — P_t^{tr} takes its cue from the phone variable \mathcal{P}_t in the CPF $p(P_t^{tr} | \mathcal{P}_t)$, meaning that each phone may have its own separate geometric duration distribution. Note that the phone transition variable is also purely random, since for each phone there is a non-zero probability of both staying at that phone state and moving on to the next phone (similar to the state transition probability in the classical HMM).

The phone variable \mathcal{P}_t is a purely deterministic function of its parents W_t^{ps} and W_t . This means that, given the current word and word position, the phone is known with certainty. Another way of saying this is that $p(\mathcal{P}_t = i | W_t^{ps} = k, W_t = l) = 1$ if $i = f(k, l)$ for a deterministic function $f(k, l)$ of its arguments k, l , and the probability is zero otherwise. In this model, therefore, each position of a word corresponds to one and only one phone (so multiple pronunciations are not represented, although they easily could be). Another deterministic relationship is the word transition. Here, a word transition $W_t^{tr} = 1$ occurs only if the model makes a phone transition $P_t^{tr} = 1$ out of the last position $W_t^{ps} = k$ of a given word $W_t = w$ where w is the index of a word that is comprised of k total phones.

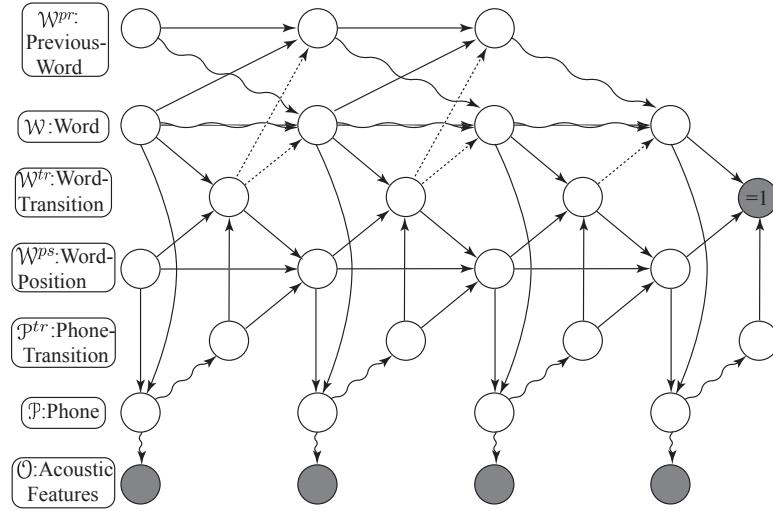


Figure 9.2: Tri-gram language model decoder

At the beginning of the utterance, the word variable W_1 starts out using a unigram distribution over words in the vocabulary. Also, the word position W_1^{ps} starts out at value 0 with probability 1 (e.g., $p(W_1^{ps} = 0) = 1$). After the first frame, these variables take on more complex distributions. First, the word position variable W_t^{ps} has essentially three behaviors depending on the value of its parents: 1) it might not change from a frame to the next frame, 2) it might increment in value by one (i.e., $W_t^{ps} = W_{t-1}^{ps} + 1$), or 3) it might reset to zero. First, if there is no phone transition ($P_{t-1}^{tr} = 0$) then the word position does not change (so $p(W_t^{ps} = i | W_{t-1}^{ps} = i, W_{t-1}^{tr} = j, P_{t-1}^{tr} = 0) = 1$, and this is true for all j , 0 or 1 in this case, but note that when $P_{t-1}^{tr} = 0$ we will always have that $W_{t-1}^{tr} = 0$). If there is a phone transition ($P_{t-1}^{tr} = 1$) and the model is not in the last position of the word, then the word position will increment with probability one. Lastly, if there is a phone transition ($P_{t-1}^{tr} = 1$) and also the model is in the last position of the word, then that will cause a word transition to occur $W_{t-1}^{tr} = 1$ which will subsequently cause the next word position to be reset to zero (so $p(W_t^{ps} = 0 | W_{t-1}^{ps} = i, W_{t-1}^{tr} = 1, P_{t-1}^{tr} = 1) = 1$, for all i). Note that this behavior is entirely deterministic, and could easily be described using a set of if-then rules or a decision tree. The deterministic behavior of the graph is in fact what helps to implement various ASR control structures.

The final variable needing description is the word variable at frames greater than one. The word variable uses the switching parent functionality (see also [41]), where the existence or implementation of an edge can depend on the value of some other variable(s) in the network, referred to as the switching parent(s). The edge from the switching parent to its child is drawn as a finely dashed edge in our graphs. In this case, the switching parent is the word transition variable. When the word transition is zero ($W_{t-1}^{tr} = 0$), it causes the word variable W_t to copy its previous value, i.e., $W_t = W_{t-1}$ with probability one. When a word transition occurs $W_{t-1}^{tr} = 1$, however, it switches the implementation of the word-to-word edge to use a “bi-gram” language model probability $p(W_t | W_{t-1})$. Strictly speaking, this graph does not switch parents, but it does switch implementations of a CPF, from a deterministic function to a random bi-gram language model.

Looking broadly at the graph, we see that most of the variables are hidden (unshaded) but some are observed. The observed variables include the acoustic feature vectors (e.g., MFCCs). As an ASR system typically uses feature *vectors* (modeled say using a Gaussian mixture), here the vector is represented simply using a single node. By doing this, any statistical assumptions made at the acoustic level between individual feature vector elements are not explicitly stated (but see Section 9.1.7 below). The other observed variable is the final word transition, which is always observed to be one. By insisting that the final word transition is unity, the graph ensures that all decodings end in a word transition. This makes sure that any decodings

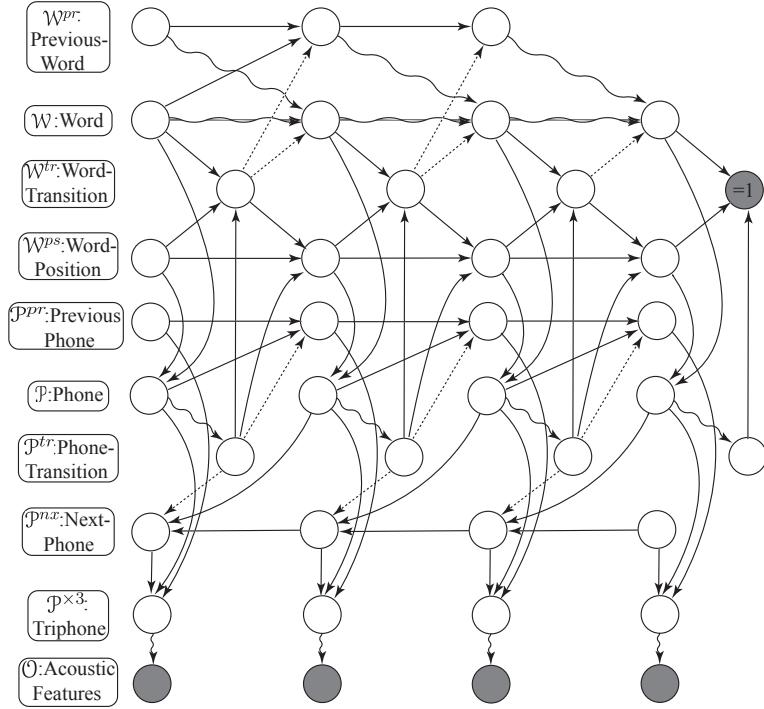


Figure 9.3: Cross-word tri-phone decoder

representing only a partial final word (meaning the final word transition would have value zero) are never considered. The ability of having this special behavior in the last frame is easily enabled by the extended DBN template (Section ??).

9.1.2 Basic phone-based tri-gram decoding architecture

In many ASR systems, moving from a bi-gram to a tri-gram language model can require a significant change in internal system implementation. Under the graphical models framework, however, it is easy to express such a construct. In fact, the graph from the previous section can be extended in only a minor way to represent tri-gram language models. As is well known, going from a bi-gram to a tri-gram will increase the state space of the model by a factor of $|W|$, the vocabulary size [206], and our case is no exception. The ease of going to this new model, however, will be quite apparent.

In this and other graphs, all the variables evolve at the rate of the frame, rather than the rate of the word. Therefore, to implement a tri-gram it is not sufficient to just add an edge in Figure 9.1 from W_{t-2} to W_t . This is because the word from two frames ago is most often the same as the current word, and is not the same as the unique word that occurred before the previous unique word, which for a given hypothesis might have occurred many frames into the past. Instead, we must explicitly keep track of the identity of the word that was most recently different — i.e., the identity of the word just before the previous word transition, regardless of when in the past that transition took place. We do this with an additional *previous word* variable W_t^{pr} in Figure 9.2. When there is no word transition ($W_t^{tr} = 0$), both the word W_t and previous word W_t^{pr} variables do not change when moving to time $t + 1$. When a word transition ($W_t^{tr} = 1$) occurs, the new previous word W_{t+1}^{pr} gets a copy of the previous current word W_t with probability one. Moreover, the new current word W_{t+1} is chosen with the tri-gram probability, but it conditions on the previous current word W_t and the *previous previous word* W_t^{pr} , as needed by a tri-gram.

As mentioned in Section 8.11, by adding this new previous word variable, the cost of doing inference

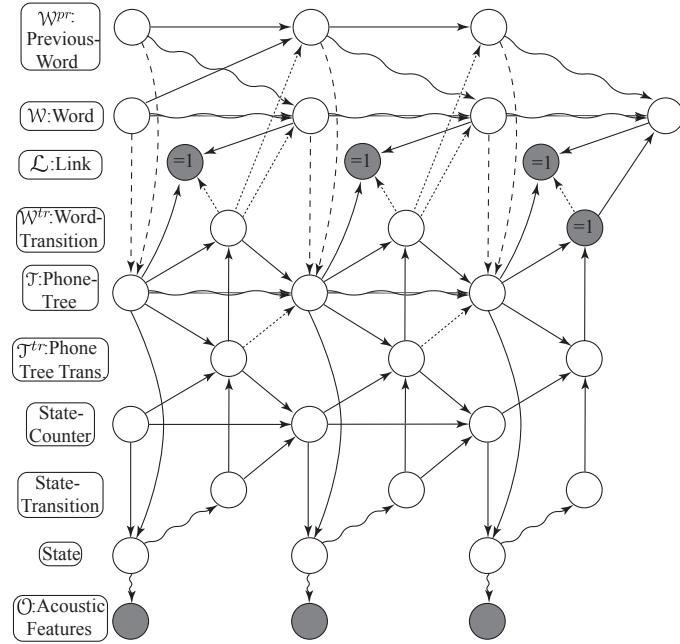


Figure 9.4: Tree-structured decoder

with this DBN has increased by a factor of $|W|$. In this case, however, the same DBN inferential machinery is used for both bi-gram and tri-gram, one of the key advantages of the graphical modeling framework.

9.1.3 Cross-word tri-phone architecture

Another technique that is typically employed by speech recognition systems is that of cross-word tri-phones [452]. Tri-phone models are those where the acoustic observation is conditioned not only on the currently hypothesized phone, but also makes the assumption that the current acoustics are significantly influenced by the preceding and following phonetic context (i.e., coarticulation). Tri-phone models do this by saying that the distribution over the acoustic frame depends on the current, previous, and next phone.

Note, however (and similar to the tri-gram language model case), it is not sufficient in Figure 9.2 just

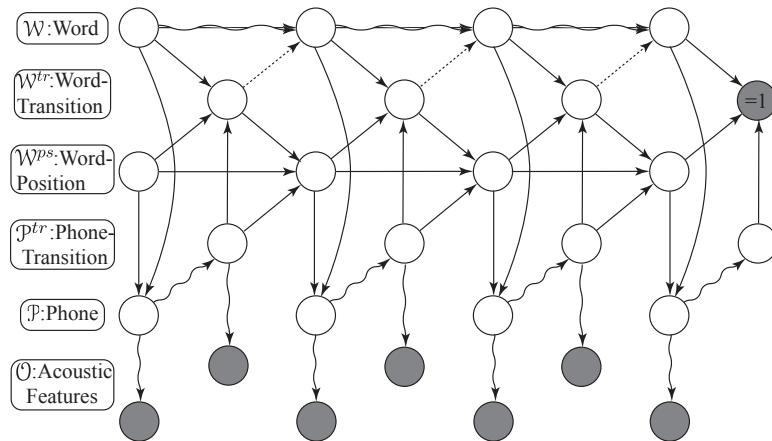


Figure 9.5: Transition specific decoder

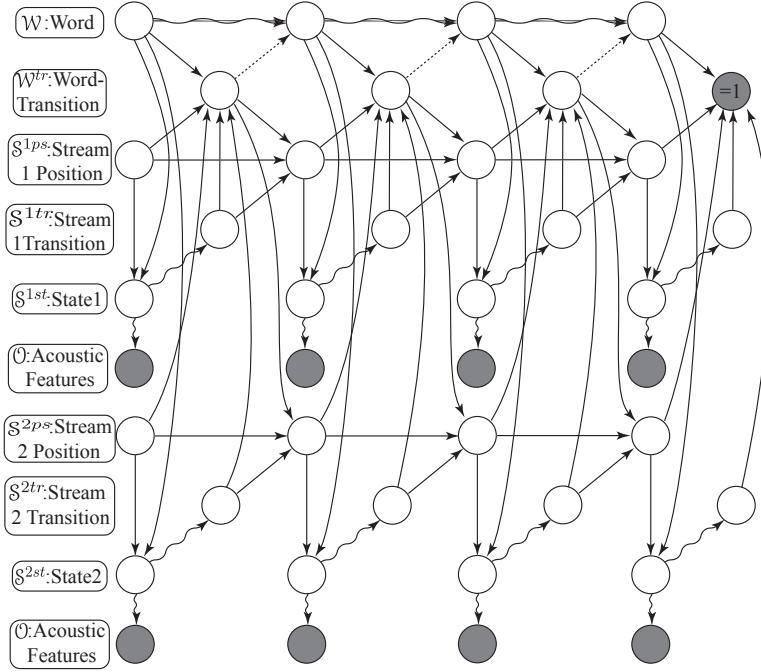


Figure 9.6: Generic multi-stream semi-asynchronous decoder

to add an edge from the previous \mathcal{P}_{t-1} and the next \mathcal{P}_{t+1} phone to the current observation \mathcal{O}_t , since these random variables indicate only the phones of the previous and next frames, and not what the previously and next actually used phone was and will be. Indeed, for any given random event, the phone of the previous and next frames might (and often will) be the same as the current frame.

We solve this problem again in a graphical setting. Our solution makes use of a novel feature in the GMTK template, namely the use of backwards time edges. The solution is shown in Figure 9.3. The top part of the graph shows the graph construct for the tri-gram language model we saw before — in particular, the phone variable is a deterministic function of the word and word position. But in this case the phone layer of the graph has three (rather than one) variables, the previous phone variable \mathcal{P}_t^{pr} and a next phone variable \mathcal{P}_t^{nx} which when combined together with the phone \mathcal{P}_t produce the tri-phone variable $\mathcal{P}_t^{\times 3}$. When a phone transition does not occur $\mathcal{P}_t^{tr} = 0$, the phone variable keeps its same value $\mathcal{P}_{t+1} = \mathcal{P}_t$ from frame to frame (since neither the word changes, $\mathcal{W}_{t+1} = \mathcal{W}_t$, nor does the position increment, $\mathcal{W}_{t+1}^{ps} = \mathcal{W}_t^{ps}$), the *next* previous phone retains its value $\mathcal{P}_{t+1}^{pr} = \mathcal{P}_t^{pr}$, and the *current* next phone \mathcal{P}_t^{nx} simply copies its future value from \mathcal{P}_{t+1}^{nx} . When a phone transition does occur ($\mathcal{P}_t^{tr} = 1$), then the current phone \mathcal{P}_t gets copied to the *next* previous phone \mathcal{P}_{t+1}^{pr} , a new phone \mathcal{P}_{t+1} is chosen based on the incremented new word position \mathcal{W}_{t+1}^{ps} , and that new phone is then copied backwards in time into the *current* next phone, i.e., $\mathcal{P}_t^{nx} = \mathcal{P}_{t+1}$ via the backwards time edge for use at time t . The backwards time edges have thus represented anticipatory coarticulation effects in the speech process. Note, however (and similar to the tri-gram language model case), that it is not sufficient to place a link from \mathcal{P}_{t+1} directly to the tri-phone variable $\mathcal{P}_t^{\times 3}$ or observation \mathcal{O}_t because most often \mathcal{P}_{t+1} will just be the same as \mathcal{P}_t . That is, \mathcal{P}_{t+1} is the phone of the next frame, but it is not the phone that will next be used — \mathcal{P}_{t+1} is the next used phone only when $\mathcal{P}_t^{tr} = 1$, at which point it is copied back to the current next phone \mathcal{P}_t^{nx} , and to earlier current next phones \mathcal{P}_{τ}^{nx} , for $\tau < t$ as needed until the previous phone transition.

This process also works across word boundaries yielding a true cross-word tri-phone model. When the model starts a new word, the last phone of the previous word is copied into \mathcal{P}_t^{pr} . Moreover, when we are at the last phone of a word, \mathcal{P}_t^{nx} will indicate the first phone of whatever the next word will be, regardless of

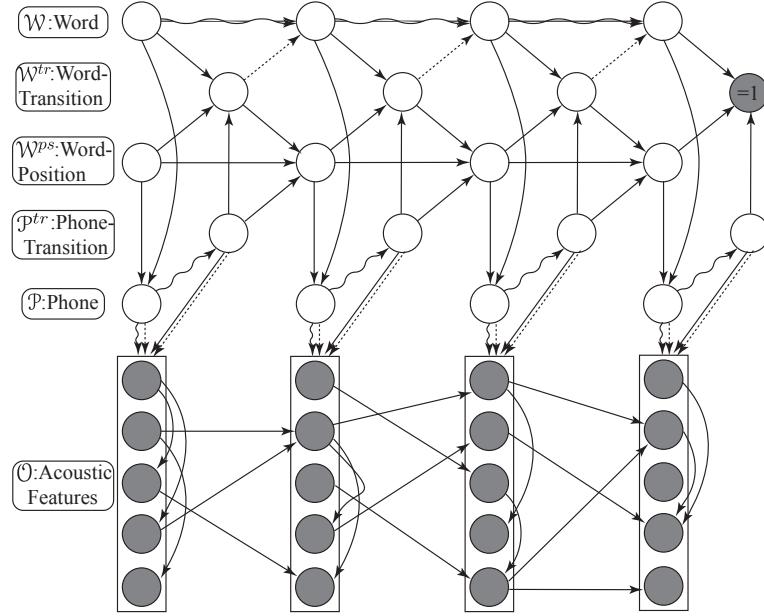


Figure 9.7: Transition and phone dependent buried Markov model decoder

when in the future it starts.

9.1.4 Tree-structured lexicon architecture

Even for HMM-only large vocabulary speech recognition systems, the decoding search space can be computationally prohibitive. It is most often necessary to arrange and reorganize states in this space in a manner that is efficient to search, and one way is to use a *tree-structured lexicon* [452] where the prefixes of each word are represented explicitly, and the state space probabilistically branches only when extending the prefixes of words that no longer match.

Such a construct can be represented using the graph structure shown in Figure 9.4. In this graph, a variable is used which corresponds to a phone tree \mathcal{T}_t . This variable encodes the prefix-tree for the entire word lexicon in a large but very sparse stochastic matrix. Each state of this variable either corresponds to a “non-terminal” state (which represents multiple words all of which have some prefix in common), and “terminal” states (which represent a single word). There are two operations \mathcal{T}_t might do over a frame depending on the phone tree transition variable \mathcal{T}_{t-1}^{tr} which also acts as a switching parent (dashed edge from \mathcal{T}_{t-1}^{tr} to \mathcal{T}_t). If $\mathcal{T}_t^{tr} = 0$ (so there is no phone tree transition), then $\mathcal{T}_{t+1} = \mathcal{T}_t$ with unity probability. If there is a phone tree transition, however, then the next phone tree state is governed by the sparse phone tree probability table $p(\mathcal{T}_{t+1} = j | \mathcal{T}_t = i, \mathcal{T}_t^{tr} = 1, W_t^{tr} = 0)$. At some point, we will reach a terminal phone tree state that corresponds to the end of a word, lets say $\mathcal{T}_t = k$. When making a phone tree transition out of this state ($\mathcal{T}_t^{tr} = 1$), this will cause a word transition $W_t^{tr} = 1$ to occur which will choose a new word. The issue, however, is that we have not yet applied the true language model probability score for this new word (since it was not known until now) in the context of the two previous words. Moreover, it is not possible simply to allow the language model structure (top two rows in the graph) to probabilistically choose the next word, since the word at this point is already pre-determined by the terminal state in the phone tree. Therefore, a soft evidence [32, 336] construct is employed. Here, a special and always observed link variable $\mathcal{L}_t = 1$ is used to insist on consistency between the word corresponding to the terminal state of the phone tree variable and the next word chosen by the tri-gram language model, but this consistency is enforced only when a

word-transition occurs. This is realized by having W_t^{tr} be a switching parent of \mathcal{L}_t . When there is no word transition ($W_t^{tr} = 0$), then \mathcal{L}_t is uncoupled from its other parents W_{t+1} and T_t (and thus has no effect). When there is a word transition, the event $\mathcal{L}_t = 1$ is explained (with non-zero probability) only when the next word $W_{t+1} = w$ is the word corresponding to the terminal state of the current phone tree variable T_t . We can moreover extend this model to provide early state pruning by allowing the phone tree probability table to have scores for the most optimistic of the subsequent words [452]. We can also provide for the context dependence of such scores by including edges from W_t and W_t^{pr} to T_t (indicated in the figure by long-dashed edges). Also, we can model the case when certain phone tree states are both non-terminal and terminal, meaning that a valid word might be a prefix of another word. Both cases must be considered, and this can be done by having the word transition W_t^{tr} be a stochastic function of the phone tree T_t so that such a state will both indicate a word, but also continue on to cover the words for which it is a prefix.

In summary, using just deterministic and random variables, and the soft-evidence construct, we have re-created a tree-structured speech decoder system, but again all out of the same basic building blocks used to produce other models.

9.1.5 Transition Explicit Model

One advantage of the graphical modeling approach is that it helps to solve problems that under more conventional approaches are more difficult to deploy. In this section, we describe one possible example. In the past, it has been argued that speech regions corresponding to spectral transitions might include much if not all of the underlying message [163, 37]. By making a simple modification to the graph in Figure 9.1, it is possible to include a construct that puts different emphasis on the speech signal depending on the current phone-transition condition.

In Figure 9.5, relative to Figure 9.1, we have attached to the phone transition P_t^{tr} another edge and observed variable. The observation could, for example, consist of features that are designed to provide information about spectral transition [163]. One option, say in an HMM system, would be to append this information to the standard feature vector, perform dimensionality reduction or feature selection, and hope for the best. The potential problem, however, is that the novel information might be relevant only part of the time, might not be needed at this level of classification (between different phones), and thus might even introduce noise in the system. Instead, in the figure we have focused this information directly on the part of the speech recognition system most likely to be beneficially influenced, namely the phone transition variable. Therefore, the graphical modeling framework and its machinery have made expressing this novel model very easy.

9.1.6 Multi-observation & multi-hidden stream semi-asynchronous architecture.

It is also easy to define a generic semi-asynchronous multi-stream and/or multi-modal model for speech recognition as shown in Figure 9.6. The streams may lie both over the observation space (e.g., multiple streams of feature vectors) and also over the hidden space (multiple semi-synchronous streams of hidden variables).

The word variable, W_t , is at the top of the structure. In the previous models each word was composed of a sequence of phones, but now we generalize this to being composed of two sequences of generic “states.” This allows us to have two independent representations of a word. Examples include an audio and a video feature stream [177, 167, 323, 322, 23], differing streams of audio features [461], different articulatory sequences [112, 237, 281, 359, 279, 185], or different spectral sub-band streams [304, 101, 178]. The position variables S_t^{1ps} and S_t^{2ps} are counters which define the current position in the sequence. The current state, S_t^{1st} (respectively S_t^{2st}), is calculated from W_t and S_t^{1ps} (respectively S_t^{2ps}). The state variables could have unique values for each word and counter value combination, or (as in a phone representation) could

have the same values for many word and counter value combinations. The transition variables, S_t^{1tr} and S_t^{2tr} , are random and determine when the sequences transition into the next state. The word transition variable, W_t^{tr} , determines the end of the word. The state sequences are free to evolve at different rates for the duration of the word, but in order for a word transition to occur, both sequences must transition out of their last state for the current word – this means that for $W_t^{tr} = 1$ we must have that both $S_t^{1tr} = 1$ and $S_t^{2tr} = 1$, and also that S_t^{1ps} and S_t^{2ps} are at their last respective value for the current word W_t . Note that there is no requirement that the two sequences use the same number of states per word, nor is there any requirement that the two sequences line-up in any way — effectively, in accumulating the probability of the word, all alignments are considered along with their corresponding alignment probability. Moreover, additional constraints can be placed on the various alignments by including edges directly between the variables S_t^{1ps}, S_t^{1st} and the variables S_t^{2ps}, S_t^{2st} . Alternatively, a soft evidence [32, 336] mechanism can be used to place symmetric constraints on the stream variables. In either case, certain alignments can be either probabilistically weighted or entirely removed from consideration with non-zero probability.

In this model, words are the points of synchrony because the two streams must start and end together at word boundaries. One can also envision a model where the points of synchrony are not the words but are rather sub-word (syllables, articulatory gestures), or suprasegmental (phrases, sentence boundaries) aspects of speech. Moreover, it is quite easy to generalize to more than two streams.

9.1.7 Architectures over Observed Variables

In this last example, we see a graph that explicitly represents various factorization properties over the observation vectors themselves (Figure 9.7). In this graph, the observation explicitly shows the length M vector of acoustic features at time t (this is written as $\mathcal{O}_{1:M,t}$), and where each of the scalar observation elements $\mathcal{O}_{j,t}$ might depend on a different scalar observation element $\mathcal{O}_{i,\tau}$ at a different position and/or different time, either earlier or later. Note that this model shows that the dependencies over observations switch from frame to frame, and they switch based on both the current phone P_t and the current phone transition state P_t^{tr} , variables that are both normal and switching parents, and so both solid and finely-dashed edges are displayed. Note that such models have been called auto-regressive HMMs [436, 66], or for the case with specific element-wise dependencies as depicted in the figure, Buried Markov models (BMMs) [31].

Unlike the models of the earlier sections, BMMs cannot be flattened into an equivalent HMM because the additional edges are between continuous feature vectors, and so this influences the model in non-discrete uncountable ways, something that can not be exactly flattened into a countable (or even finite) state space. Moreover, BMMs provide a promising vehicle for structure learning [39, 116] since the structure to be learned is over sequences of only observed feature vectors (rather than hidden variables). Efficient and provably optimal structure learning methods exist in certain cases when the variables are observed [93, 195, 318]. In past work, in fact, it was shown how to learn the structure in a discriminative fashion over such variables [31].

9.1.8 Polyphase speech recognition

In speech recognition, the speech signal is typically processed as a sequence of overlapping windows of acoustic spectra in time, typically of width 25ms, stepping by 10ms at each window. Certain sub-word phonetic units, which might last anywhere from 10ms to 40ms in duration, might not have an ideal overlap with any given fixed spacing of the windows. A short phone unit might exist in a frame only with other flanking units, thereby increasing the confusion in identifying such a unit. A long unit might be truncated by the fixed-width window. It is of interest, therefore, to develop methods that might mitigate such effects and this is the idea behind polyphase speech recognition. While the author readily admits that this is not a standard method for speech recognition, and it rather is a fairly new approach with only some initial results,

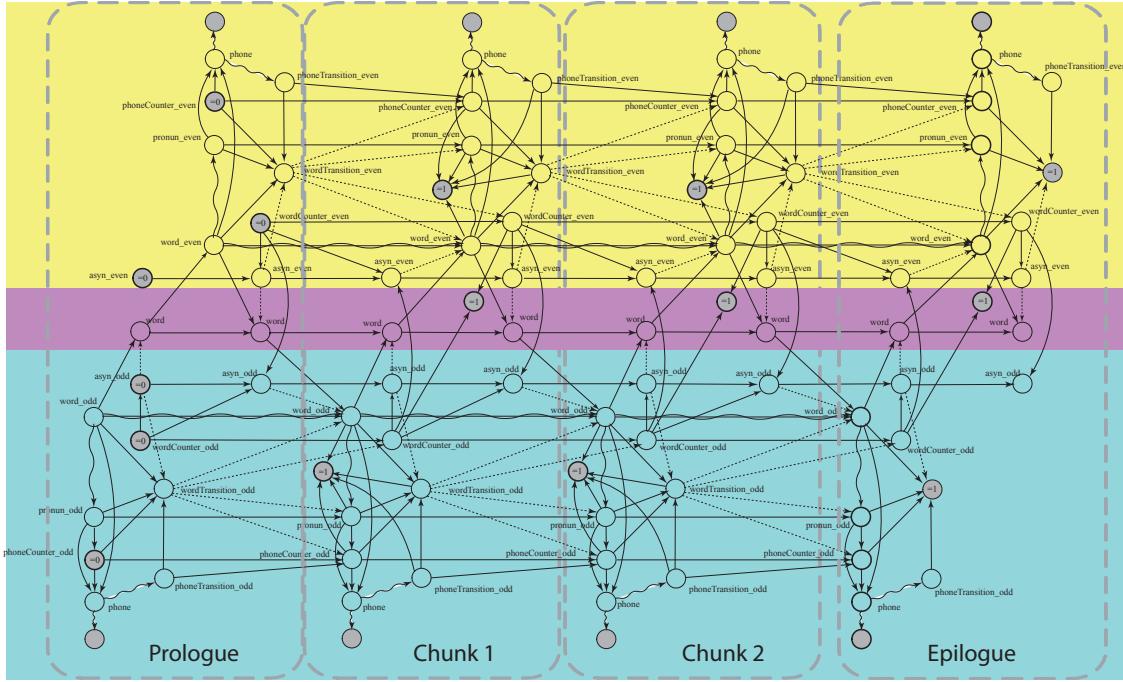


Figure 9.8: A DBN template that represents the process of polyphase automatic speech recognition for two phases, from [269].

the resulting DGM is interesting in that it demonstrates the flexibility and power of the underlying paradigm.

Figure 9.8 shows the complete DBN for 2-phase polyphase speech recognition. We describe only the essential ideas regarding the working of this graph and refer the reader to [269] for more details. The speech signal, rather than being divided into a succession of overlapping windows, is divided into two separate streams, each one a succession of overlapping windows, but each is out of phase with each other by a duration equaling the window step size divided by the number of streams (2 in this case). Therefore, the graph above is such that the even frames correspond to one stream, and the odd frames correspond to another stream. Each of the even and odd frames constitute a complete speech recognition system in that each arrives at its own hypotheses for the sequence of words for its own stream. The word hypotheses, however, are decided jointly and this is accomplished by an integrating network between the two streams. The integrating network allows the two streams to be partially de-synchronized in that one stream might advance to the next word a certain number of frames earlier in time than the other stream. Asynchrony, however, is constrained so that the two streams may not become out of sync by more than a distance of one word. The integrating network works by keeping track of the leading and following stream for a given hypothesis. The following stream's hypothesis is not allowed to fall too far behind the leading stream, and the following stream is required to advance to the same next word that the leading stream advanced to. Any hypotheses where the lag between the two streams becomes too large, or where the following stream does not advance to the appropriate next word, are precluded by the integrating network from scoring with non-zero probability.

The model can be generalized to any number of streams, and this is done in [269] to 3 streams. In this more general case, the integrating network keeps track of the leading stream (the one that first hypothesizes the next word) and constrains all other streams to make the same hypothesis when they get to the point where they wish to advance to their next word. More details of this model, as well as results, may be found in [269].

9.1.9 Architecture Summary

Looking over the architectures from the previous sections, we see that many of the constructs are already available in ASR systems. A key advantage of GMs is that all of these constructs can be represented in a single, unified framework. In general under this framework, many other modifications to the above given structures can be quickly utilized (Section 9.1.5 shows one such example), and even slight modifications can lead to quite different ASR systems. While some of these modifications will improve performance and others will not, graphical model machinery can make it easy to both express a new model, and with the right software tools, to rapidly prototype it in a real ASR system. In addition, since the space of possible directed graphical models is so large, it is likely that even radically new ideas can be represented and rapidly prototyped in this framework without having to re-write an entire ASR system. This is an important promise of the graphical modeling paradigm, but it does require a working toolkit.

Chapter 10

Dynamic Graphical Model Examples for Natural Language Processing

Chapter 11

Dynamic Graphical Model Examples for Computational Biology

11.1 Example: Segment Modeling in Bioinformatics

Bioinformatics is an important research area involving learning and reasoning about biological molecules (e.g., DNA and proteins). For example, one might wish to predict the 3-D structure of a protein using only the sequence of constituent amino acids, or given a DNA sequence the goal might be to decide which sub-segments code for currently undiscovered proteins and to determine how can one automatically infer their ultimate function. DGMs are quite appropriate for this domain, and there have already been many successful instances of HMMs and its variants in bioinformatics [218].

We next give an example DGM that is useful in sequential modeling of segments. The DGM (which is actually a DBN, see Figure 8.79) corresponds to a modular reusable DBN component that allows for non-geometric state duration distributions. Each state, represented by variable S_t , may have its own state duration and determines the distribution over observations O_t like an HMM. Unlike an HMM, however, this model has quite different state duration behavior. A state change is triggered by a binary state change indicator variable Δ ; that is, $S_t = S_{t-1}$ when $\Delta_{t-1} = 0$, but when $\Delta_{t-1} = 1$, then S_t is chosen randomly based on $P(S_t|S_{t-1})$. The actual duration distributions of each state is determined by variables S , I , and C , and may be one of: 1) a fixed length distribution; 2) an arbitrary multinomial distribution, and 3) a state-dependent negative binomial distribution. To implement a fixed (non-probabilistic) duration, when $\Delta_{t-1} = 1$, then C_t gets set to the desired length and it decrements deterministically (i.e., $C_t = C_{t-1} - 1$ with probability 1) until $C_t = 0$ at which point it triggers a state transition. In this case, C_t ignores I_t . To produce an arbitrary multinomial duration distribution, rather than deterministically assigning C_t to a length, C_t gets a random length based on a multinomial distribution $p(C_t|S_t, \Delta_{t-1} = 1)$. C_t then decrements as in the first case until it reaches zero. To produce a negative binomial distribution corresponding to the sum of k geometric distributions, C_t is deterministically set to k , but then C_t decrements only when I_t (a state-dependent binary indicator variable) is unity. Of course if $k = 1$, then we recover the standard HMM geometric duration, which would also be an option for some states. As can be seen, by explicitly encoding the various length distributions within the structure of the DBN, there is considerable flexibility for duration modeling.

The above example is a modified and simplified version of a DBN used as a method to predict the topology of transmembrane proteins [356]. Transmembrane protein prediction is a sequential labeling task, where each amino acid of a protein is labeled as belonging to one of the following four classes: cytoplasmic loops, membrane-spanning segments, non-cytoplasmic loops, and signal peptides. Different classes have very different duration properties.

Chapter 12

Dynamic Graphical Model Examples for Activity Recognition

Part V

Inference In Dynamic Graphical Models

Chapter 13

Inference In Dynamic Graphical Models

Inference in dynamic graphical models is quite different than inference in static graphical models. There are operations that one needs to perform in order to produce efficient exact or approximate inference in dynamic models that are not applicable with a static model. This is owing to the distinct shape that a dynamic model has, namely that its much wider than it is higher, as seen below.

Consider, for a moment, the DBN given in Figure 9.1. This is a fairly uncomplicated model and when looking at it like this (a template unrolled one time) it seems like perhaps any static graphical model method perhaps could apply. When the model is unrolled, however, it starts to take on quite different properties. Once the model is unrolled 12 times, for example, it might look more like Figure 13.1a.

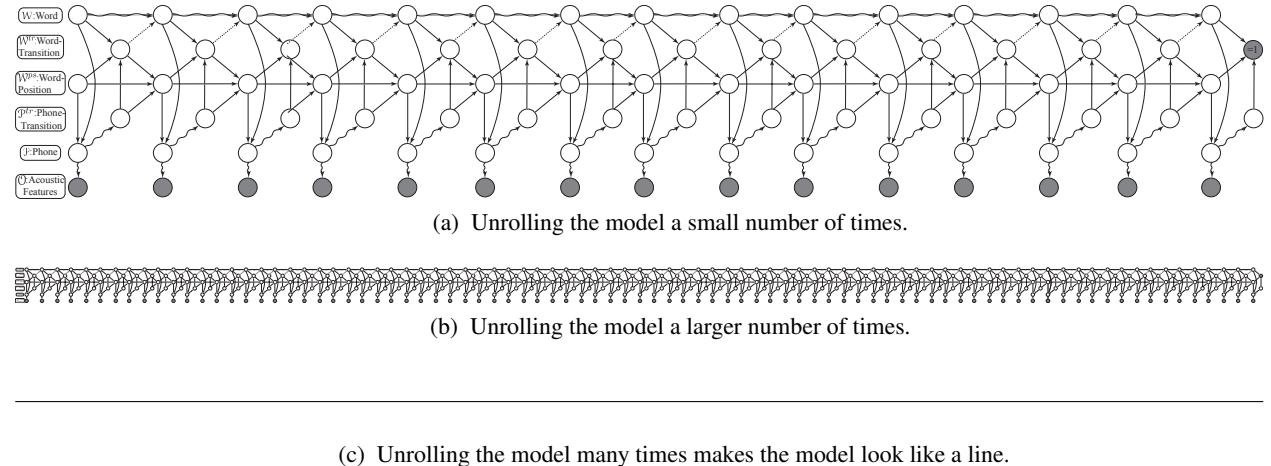


Figure 13.1: Unrolling the DBN given in Figure 9.1 many times. Ultimately, unrolling a model enough times means the model from afar appears as a horizontal line. This gives clues to the forms of inference that should be done, and that it should done using some form of left-to-right or right-to-left strategy.

If we unroll the model many more times, it would look like Figure 13.1b. And if we unroll it still more times, to a length common in many biology applications, the model might look still more something like the horizontal line in Figure 13.1c. Note that this is a property of any DGM, that ultimately the model becomes much “winder” than it is “higher,” something not necessarily true for graphical models in general. Hence, when designing inference strategies for general DGMs, it would be wise to be aware of this DGM distinction.

On the other hand, as we have said, there is always some form of temporal Markov property in a DGM (§8.3). Therefore, once a set of operations on a DGM are performed, it is possible in many cases to reduce DGM inference to static model inference, and many of the advanced methods for static inference can be

applied to the dynamic case as well.

The goal of this chapter is not so much to give a complete overview of exact and approximate methods in static graphical models. Rather, our goal here is to discuss and then ultimately outline the operations on DGMs that can enable the use of static graphical model inference techniques. We do cover a bit of the static inference methods, as there are some constraints that must be obeyed and that are handed down from the operations that transform from the dynamic to the static. We also note that the operations that are considered here in this chapter are equally applicable to importance sampling, particle filtering, and sequential Monte Carlo methods as well.

13.1 Inference in HMMs: The Search Space

The reader is to be reminded of the discussion of the inference methods for HMMs. This includes the forward/backwards (α, β) recursions and its variants in §8.4.11 including the push strategy in §8.4.4.7, and other operations such as computing the k -best assignments in §8.5. Therefore, we have already discussed inference in HMMs. Why, then, do we have a full section here again on HMM inference? The reason is that we will be discussing aspects of inference that are applicable to the more general DGM setting even though at first we will still consider only the HMM.

That is, we discuss inference in a slightly different setting, one that shows how HMM inference is akin to search in artificial intelligence, and also one that helps us to develop inference for DGMs. In particular, the methods here are appropriate when the state space gets very large, and when even sending one message (corresponding to one $O(N^2)$ time-step of the α -recursion) is computationally difficult. The reader should recall the HMM inference procedures that lead to these complexities in order to compare to their analogues in the DGM case. We describe in the below the junction tree and inference algorithm for HMMs, and how it compares to several standard methods for HMM inference, namely the *synchronous* and the *asynchronous* method of decoding. This will allow us, later in the paper, to describe how these approaches relate to DGM triangulation.

The trellis of an HMM (cf. §??) shows the underlying $O(TN^2)$ cost of an HMM — at each time step, we see N states indicated as nodes of the trellis and between two time steps we see at most N^2 links between trellis nodes.

In general, in exact HMM inference, the goal is to “visit” each node in the HMM trellis, and this can be viewed as a form of search procedure [373]. The problem of “search” in artificial intelligence corresponds to intelligently expanding a very large space of possible elements as efficiently as possible. For example, given a factored function over variables indexed by set V , such as $f(x_V) = \prod_C \phi(x_C)$, where each $C \subset V$, the goal might be to identify a maximum element $\text{argmax}_{x_V} f(x_V)$. An alternative might be to sum f for all values of x_V . One can imagine doing either of these naïve using a set of nested loops, where we first iterate over all values x_1 , and then x_2 , and so on, and at the deepest level, when all elements of x_V are instantiated, we can evaluate f . In general, any partial or complete set of variable assignments is called a “hypothesis” and along with each (partial) hypothesis is a (partial) score based on the set of factors that may currently be evaluated. There are many types of search, including standard depth, breadth, best first, and A* search procedures all of which are detailed in [373] and which are summarized in Figure 13.2. These algorithms can in general perform much better than naïve nested loops, and they do so by making decisions based on scores of partial hypotheses.

The use of search methods in sequential models for speech recognition in fact has a long history [325, 200]. Most search methods are applied to domains where the search space expands exponentially with the depth of the search, so the search expansion takes the form of a tree, and the deeper the search proceeds, the number of possible hypotheses expands exponentially (as shown in Figure 13.2).

With an HMM (or any finite-state sequential model), however, the search space has quite a different

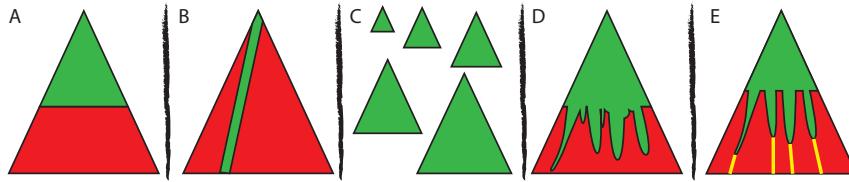


Figure 13.2: Various forms of search expansion shown graphically. In each case, green depicts partially expanded hypotheses, red unexpanded portions of the space, and yellow continuation heuristics. A: a stage in breadth first search, where all partial expansions at the same depth are expanded before proceeding to deeper depths. B: a stage in depth first search, where complete assignments to all variables are available one after the other, but with often minimal changes between successive assignments. C: iterative-deepening depth-first search, where successively deepening limited-depth depth first search procedures are run one after the other. D: best-first search, where different depths of different hypotheses might be simultaneously and partially expanded based on the scores of partial hypotheses. D: A*-style search, where partial hypotheses might be expanded using a continuation heuristic which estimates the remaining score necessary to complete a partial hypotheses. In E: the paths used by continuation heuristic are shown in yellow. If the continuation heuristic is optimistic regarding the continuation (meaning that it is an overestimator if the score is a probability or a utility, or an underestimator if the score is a cost or a distance), then it is called an admissible heuristic.

general shape, one that is more akin to a very wide parallelogram (see Figure 13.3). When t (which is the time variable, but acts also as the depth variable when considering HMM inference as search) is small, the state space can (in general) expand exponentially in t . At some critical point, the state space saturates. The latest point that saturation can occur is when $t = N$, meaning that there is non-zero probability for all states to have been visited. After this point, the number of states per time step is fixed at at most N and this continues until t is close to T . At that point, the effective search space essentially contracts since there are often only a few states that are allowed to end the search and be given a non-zero score. For example, in speech recognition, one must end the search at the end but not in the middle of a word.

Another difference from standard search is that with an HMM, the depth (which corresponds to the time length T in the DGM case) is very large, so the search space really does not look like an upside-down tree as in Figure ??.

Search in HMM really corresponds to the α calculation, which can proceed as usual using the alpha recursion Equation (??) from §8.4.11.1. Alternatively, search can proceed from time-step to time-step using a push strategy as shown in including the push strategy given in §8.4.4.7.

Search in HMMs can for the most part be broken down into one of two approaches: synchronous vs. asynchronous. In synchronous (also called Viterbi) search, hypotheses are expanded in temporal lock-step, where partial hypotheses that end at time t are expanded one at a time so that new partial hypotheses end at time $t + 1$. This continues for all t so that at any given time there are never extant partial hypotheses with end-points at more than two successive time points. This is essentially breadth-first search and is shown in Figure 13.4 A and B. The standard α -recursion and the push strategy are both synchronous search approaches.

Asynchronous search, on the other hand, is such that the end-points of partial hypotheses can be at arbitrarily different time locations. Each partial hypothesis h has its own end-point consisting of a time and state pair, and its own current score s . The hypotheses may be expanded without needing to abide by any temporal constraint (see Figure 13.4 C and D). Asynchronous search is also called stack decoding, the reason being that one often keeps a priority-queue (implemented as a stack) of hypotheses and the hypotheses expansion occurs in a best-first order. It is also useful in stack decoding to have a form of continuation heuristic (a value that estimates the score of continuing from the current point to the end of the utterance) so that the best-first decisions are based on the combined current hypothesis score and its

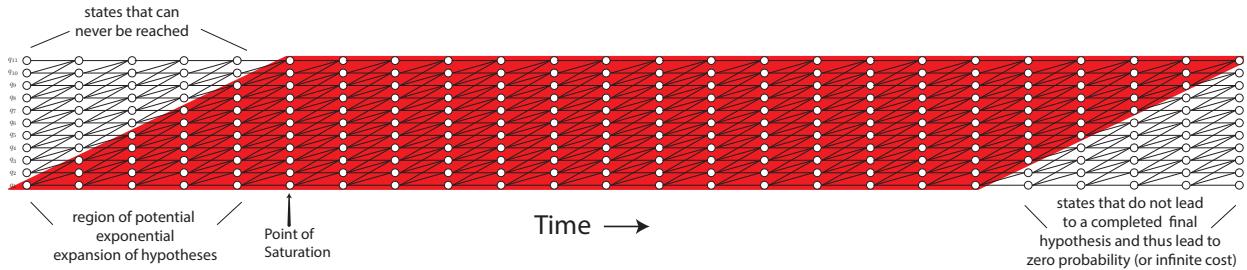


Figure 13.3: Example of the parallelogram shape of the HMM search space. The highlighted red corresponds to the expandable portions of the state space — the white portion is either unreachable (i.e., “burn in”, at the beginning of the state space and left of the parallelogram, the reason being that one must start the search in state q_1 , which might correspond to the beginning of a word in speech recognition), or leads to an failed search (at the end of the of the state space and right of the parallelogram, the reason being that one must end the search in state q_{11} which might correspond to the end of a word). On the left, the state space expands with t and this expansion can be exponential — what is shown in figure is not exponential but it can be. This expansion occurs up to the earliest point of saturation (indicated) at which the per-unit-time state-space (since it is a finite-state model) can no longer grow (in the figure, the expansion shown is linear, due to the nature of the HMM’s transition matrix). When approximate inference schemes (such as pruning) is used, it may be that the point of saturation is never reached.

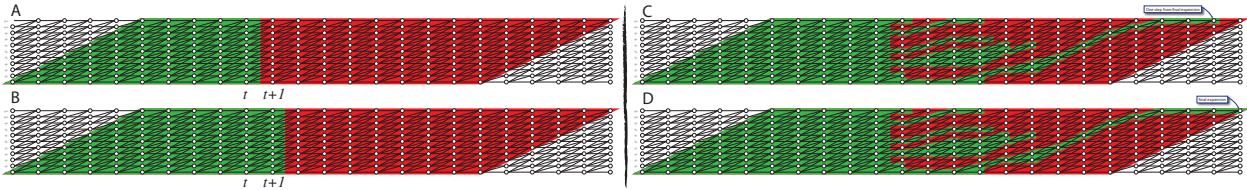


Figure 13.4: The parallelogram search space of HMMs. Left A & B: asynchronous search: A: All states up to and including but nothing beyond time t have been expanded. B: all states at time $t + 1$ are expanded synchronously. Right C & D: asynchronous search. Much like best first or A* search, arbitrary partial hypotheses may be expanded at any given time. In C, we are one step from the final expansion, which is shown to occur in D.

continuation. If the continuation heuristic is optimistic, then it is “admissible” and we have an A*-search procedure [335, 175]. When the scores are probabilities, an admissible heuristic is one that gives an upper bound of the true continuation probability — when the scores are costs (e.g., negative log probabilities), an admissible heuristic gives an lower bound on the true cost.

Both synchronous and asynchronous search procedures have been widely used in automatic speech recognition in the past. Synchronous procedures have for the most part bested their asynchronous brethren perhaps due to their overall efficiency, simplicity, and performance [200]. One of the advantages of synchronous search is that pruning (which is a form of approximate inference) is quite simple, both conceptually and to implement. Assuming that M_t is the maximum score of a set of states at time t , two simple widely used pruning strategies [325] are as follows: with *beam pruning*, we remove all partial hypotheses that are some fraction below M_t , and with *K-state pruning* (sometimes called histogram pruning), only the top K states are allowed to proceed. *K*-state pruning is particularly attractive since it can be efficiently implemented using either a histogram [325], or by using a fast quick-sort like algorithm to select in $O(N)$ time the top $K < N$ out of N entries. Another reason for the popularity of synchronous search is that, when the state space is very “deep” as shown in Figure 13.4, it can be a challenge to find good continuation heuristics.

Much research has gone into mitigating these problems, such as fast-match heuristics [175] (where a simpler structure is used to obtain continuation scores which are then applied to complex structures), and coarse to fine based dynamic programming (where coarse-grained version of the problem is solved first, and then refined). It is challenging for such heuristics to perform well, perhaps because they need to make inferences and decisions based on events that are quite distant in the future. In general, A* search seems to work better for short and fat search problems than for long and thin search problems.

All need not lie at the extremes, however, as there can conceivably be hybrid approaches that straddle the fence between asynchronous and synchronous search. For example, a constraint can be placed on the maximum temporal distance between the end-point of any two partial hypotheses. Another approach would be to expand hypotheses based on some underlying (but constraining) data structure. In fact, this is something that a junction tree could easily do. Normally in a junction tree, the form of message passing is based either on the Hugin or the Shenoy-Shafer architectures [290]. The Hugin style message passing is summarized in Figure 2.6-II. Each of these approaches, however, assume that the cost of a single message is itself tractable, which is not the case in many applications due to the very large number of possible random variable values (e.g., in speech recognition, consider the typical vocabulary size of a large vocabulary system, which might be more than 250,000). Therefore, even individual exact messages in a junction tree might be computationally infeasible and this is where the search methods become useful.

Before moving on, we give a bit of terminology. A clique is a fully connected subgraph of some graph, and the nodes of a junction tree can be seen as a clique. When discussing inference, a clique is really a table of entries, where each row of the table consists of a set of random variable values and a score associated with that. We call this a *clique table*. For example, given a clique with five random variables A, B, C, D, E , a clique's table could be seen as a table implementation of a function $\{(a, b, c, d, \psi(a, b, c, d)) : a \in D_A, b \in D_B, c \in D_C, d \in D_D, e \in D_E\}$. Note that the value $\psi(a, b, c, d)$ might include information from many different parts of the graphical model depending upon the current stage in inference. For example, after collect evidence, the values of ψ are akin to the α -recursion values in an HMM.

We now consider the case where a search procedure is used to perform the junction tree message by expanding the entries within a clique. The expansion is constrained to apply only within the clique so that the decoding is “synchronous” but only at the clique level (we are not allowed to expand a clique C_{t+1} until clique C_t has been expanded). Lets suppose that the clique variables are indexed by $C = (1, 2, \dots, n)$ and that there are a number of separators coming into the clique during the forward (or collect-evidence) phase of computation. Clique expansion could then take the form of:

Algorithm 15: Naïve clique expansion algorithm.

```

1 for  $y_1 \in D_{Y_1}$  do
2   for  $y_2 \in D_{Y_2}$  do
3      $\ddots$ 
4     for  $y_n \in D_{Y_n}$  do
5        $\psi(y_1, y_2, \dots, y_n) \leftarrow \psi(y_1, y_2, \dots, y_n) + \phi_C(y_1, y_2, \dots, y_n) \prod_S \phi_S(y_S);$ 
```

where C is a clique and $S \subseteq C$ ranges over all the incoming separators to the clique. This form of expansion itself can be quite expensive if the clique is large. Within a clique, however, the expansion need not occur in the variable order given above and in fact can occur in any variable order. Moreover, there need not really be any given fixed order — variable order search is well known in the constraint satisfaction literature [8], and in this context, when all the variables are timed, variable order search is much like an asynchronous decoding procedure.

For example, consider the junction tree for the HMM given in Figure 8.44, where each clique consists only of successive variables Q_t, Q_{t+1} . If we were to perform constrained asynchronous search in such a

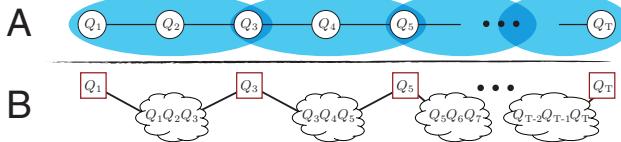


Figure 13.5: A: The DGM for the effective state space of the HMM. B: a junction tree corresponding to the HMM, where each clique has three random variables.

junction tree, where cliques are expanded one after another but expansion may be two-frame limited-extent asynchronous within a clique, then we have recovered synchronous search in HMMs, since in synchronous search it is never the case that more than two frames can be expanded simultaneously.

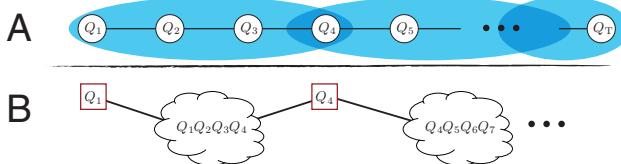


Figure 13.6: A: The DGM for the effective state space of the HMM. B: a junction tree corresponding to the HMM, where each clique has four random variables.

However, since cliques can consist of any number of random variables, and could even span over short stretches of more than two time steps [35], then the cliques in the junction tree can effectively limit the extent of the hypothesis expansion. For example, consider the alternative junction tree for an HMM, shown in Figure 13.5. In this case, each clique consists of three rather than two random variables. This is still a junction tree, it is just that the cliques are bigger. Ordinarily this would lead to an expansion of the search space if one were to do the more naïve clique expansion in Algorithm 15, the state cost would become $O(N^3)$. If instead of Algorithm 15, a limited extent asynchronous search is performed, one could recover the the $O(N^2)$ cost but would also have more flexibility regarding hypothesis expansion.

Figure 13.6 shows a 4-variable clique junction tree for the HMM. In the triangulated graph corresponding to these junction trees, the triangulation is no longer minimal (meaning that some of the fill-in edges may be removed and the triangulation property still holds). In some cases, however, non-minimal triangulations can be useful [14]. In this case, in fact, the non-minimal triangulation is used to restrict the degree of asynchrony in an HMM expansion. By forming various triangulations (and corresponding junction trees) we can experiment with an even wider variety of different search expansion constraints for an HMM. Below, and in particular Section 13.4.2, we generalize this idea to DGMs where a junction tree can be used to limit the scope of local within-clique search methods.

In the next section, we describe how general DGM inference methods can be derived that are based on the idea that a junction tree can be used to limit the scope of local within-clique search methods. While this can be done for any graphical model, dynamic or otherwise, we will also see how the unique nature of dynamic models render this approach particularly useful.

We note that the two aforementioned pruning options can easily be extended to the junction tree case. Once a clique is expanded, we can compute the top scoring set of random variables in the clique table, say it has score M_t . Then, any clique entries with score outside of the beam (using either beam pruning or K -state pruning) can be removed, and then the messages can proceed from there.

Since the expansion can be asynchronous within a clique, moreover, it can be useful to establish a continuation heuristic so that only the most promising hypotheses are expanded (similar to a best-first search), but in this case, since we know that the expansion is limited to go no farther than the temporally latest vari-

able within a clique, the continuation heuristic need not extend beyond a clique. On the other hand, since we are not expanding the model out to the last frame T (as is done in standard asynchronous decoding) to do this well, we would need to know M_t before we compute it. This chicken-and-egg problem, however, has a solution that suggests new approach to pruning based on online-prediction, as described in the sections below.

13.2 Inference in DGMs

In its most general form, performing probabilistic inference in a graphical model corresponds to computing marginal probabilities for all cliques in the graph given the evidence. To make this more concrete, let \mathcal{C} be the set of all cliques in the triangulated graph, where for each $C \in \mathcal{C}$, we have that $C \subseteq V$. The set of random variables is X_1, X_2, \dots, X_N where $N = |V|$ and some subset $E \subset V$ have known values, meaning that they are “evidence” nodes or are “observed”. The joint distribution then becomes $p(x_{V \setminus E}, \bar{x}_E)$ which is a function only of the non-evidence nodes $V \setminus E$. The goal of graphical model inference is to produce the set of clique marginal probabilities of the form $p(x_{C \setminus E}, \bar{x}_E)$ for all $C \in \mathcal{C}$.

In the case of an HMM, a key goal of inference is to produce $p(Q_{t-1}, Q_t, \bar{x}_{1:t})$ for all t since pairs of successive state variables constitute all cliques in the DGM (see Figure 8.44). Computing these clique potentials is needed for learning the parameters of a graphical model, or finding the most likely assignments to the hidden variables.

Note that some applications might not need all of the clique marginals, as discussed in §8.4.11.5 and §8.4.11.8. For example, we might need only one clique marginal if that is the sole interest for a particular application. In other cases, we might not need to include all of the evidence in all queries. Kalman filtering (and those more general queries for DGMs inspired by Kalman filtering) is such an example. For example, in Kalman filtering, we need to compute queries of the form $p(q_t | \bar{x}_{1:t})$. Kalman smoothing needs to produce queries of the form $p(q_\tau | \bar{x}_{1:t})$ with $\tau < t$, and Kalman prediction needs queries of the same form but with $\tau > t$. For any given application, however, the underlying computations that lead to producing all clique marginals are quite similar to those that produce a subset of the queries, although there might be some computational benefit to producing a subset of queries. In this section, therefore, we concentrate on producing all clique marginal queries since it is most general.

The belief propagation algorithm when applied to a junction tree will indeed produce all clique marginals. Thus, the junction tree algorithm computes the queries that are necessary for learning, and is a superset of what one might want for applications such as filtering, smoothing, and prediction. But we are interested in aspects of this algorithm that are unique to the DGM case.

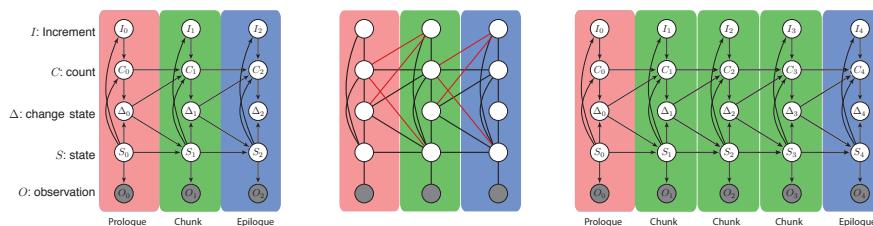


Figure 13.7: Left: A DBN template that consists of a prologue, chunk, and epilogue, and that forms a simple segment model useful in bioinformatics [356]. We saw this before in Figure 8.79. Middle: The moralized template where fill-in edges are shown as red. Right: The template unrolled 2 times.

A key goal is to deduce an inference algorithm for any $p \in \bigcup_{\dot{\tau} > 0} \mathcal{F}(G, \mathcal{M}, \dot{\tau})$ using the information only in the family of the template $\mathcal{F}(G, \mathcal{M}, 0)$ and perhaps a small amount of unrolling $\mathcal{F}(G, \mathcal{M}, \dot{\tau}')$ for some constant $\dot{\tau}'$. That is, the template should be sufficient to deduce the computational properties and inference

procedure for any amount of unrolling. Doing so means that the cost in deducing such an inference algorithm is amortized over the use of (G, \mathcal{M}) .

Given a DGM template G , for any fixed $\hat{\tau}$, any $p \in \mathcal{F}(G, \mathcal{M}, \hat{\tau})$ factors with respect to a fixed graphical model, namely G unrolled $\hat{\tau}$ times. One option to consider therefore is to unroll the graph for each desired $\hat{\tau} \geq 0$ and treat each unrolled graph as static graphical model. With this set of graphs, we can deduce a separate inference procedure (using, say, belief propagation) for each such unrolling without any sharing of information between the other unrolled models. This naïve approach, however, does not achieve the aforementioned goal. We will show, however, using only $\mathcal{F}(G, \mathcal{M}, \hat{\tau}')$ for small constant $\hat{\tau}'$, how to deduce an inference algorithm for $\mathcal{F}(G, \mathcal{M}, \hat{\tau})$ for any $\hat{\tau} > 0$.¹

If the DGM is a Bayesian network template (meaning the model corresponds to a Dynamic Bayesian network), the first thing that occurs is a template moralization step (as in Figure 13.7-middle), so that any factor in the BN template can find a containing clique in the resulting undirected model. We assume in the below that all DGMs are undirected and that we are working with the family of MRFs — this assumption does not make the procedures any less applicable to other families of models, such as DBNs or temporal CRFs.

In the next several sections, we describe how inference can be performed efficiently in very large state space dynamic graphical models. While these methods are general, these are the methods that have been implemented in GMTK and so we describe them in some detail here.

The techniques below (starting in §13.3.1) generalize the methods mentioned in §13.1 in the case of an HMM, and they consist of two phases. The first phase is graph-theoretic and organizational, and allows the DGM to be turned into sections of a junction tree in which inference can be performed. Within each section, almost any static graphical model inference methods can be used — this is quite useful, since we can bring to the table the entire arsenal of both exact and approximate inference methods developed for static models, and then apply them to dynamic models. Once an inference strategy is chosen within each section, these sub-junction tree sections can be spliced together sequentially so as to represent any length of temporal signal. This can, moreover, be done in a way so that there is no loss of optimality relative to the optimal exact DGM inference algorithm. In other words, it is possible to show that even though inference is performed by splicing together blocks of graph, within the space of inference algorithms lies the best possible exact inference algorithm had no splicing been performed (and only unrolling performed).

The next phase is a particular kind of inference, and constitutes a form of message passing that also generalizes HMMs. It is a hybrid synchronous/asynchronous algorithm where the degree of asynchrony is dependent on the triangulation and underlying junction tree. The message passing algorithm, moreover, is formulated so that the many approximate beam-pruning options that have been highly successful in HMM inference can be applied here as well.

13.3 Graphical aspects of inference in DGMs

In this section, we concentrate on graphical aspects of inference in DGMs.

13.3.1 The Case of Unroll and Compute

As mentioned above, since it is not viable to unroll the graph for each T and then repeatedly re-solve the inference problem, there must be some way to deduce inference strategy based only on the template. Before we do that, however, we consider the case of unrolling the graph to length T and then solving inference using the elimination algorithm. While this is not ultimately how inference will be solved, it introduces a number of concepts that will be important to know when we solve inference in the template case.

¹We consider unrolling a small and constant number of times, $\hat{\tau}'$, a reasonable step in deducing inference properties for all

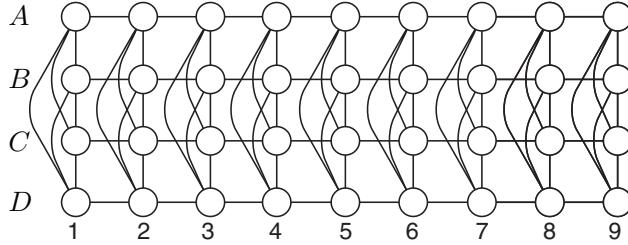


Figure 13.8: The graph depicting the state space of the factorial HMM when $M = 4$. Note that the observations are not shown since they do not change the state space. Moreover, this corresponds to the moralized (but not yet triangulated) Factorial HMM. For simplicity, we give the variables the name A, B, C, D .

13.3.1.1 Unroll and Compute with the Factorial HMM

We saw that α and β recursions were just elimination orders on the HMM graphical model. Moreover, we saw how the α -recursion corresponds to eliminating variables in sequential order from left-to-right (i.e., increasing t) and β recursion is elimination in sequential order from right-to-left. In this section, we'll generalize these recursions to more structured dynamic models.

Lets start with the factorial HMM, which is discussed in §8.9.1. In fact, to be concrete, lets consider the case when $M = 4$ and also remove the observation from the picture since it does not influence the state space². Once we do this, we get a graph as shown in Figure 13.8, where at each time step we have variables A_t, B_t, C_t , and D_t . It can be seen that the tree-width of this graph is four, so the best we can hope for, when doing exact inference, is a clique size no more than five.

If we eliminate variables strictly left to right (meaning no variable at time $t + 1$ is eliminated before all variables at time t are eliminated), and we eliminate in the order (A_t, B_t, C_t, D_t) (i.e., top to bottom) we'll encounter cliques of size at most five (in general $M + 1$). Also, there will be M such cliques encountered at each frame t . Therefore, the $O(r^{2M}T)$ naïve complexity is significantly reduced to only $O(Mr^{M+1}T)$, and this is achieved using only a strictly left-to-right frame-by-frame elimination scheme. The graph with fill-in edges (and differently named variables) is shown in Figure 8.71f. A junction tree for this model would have M cliques for each time t , each clique would contain no more than $M + 1$ variables, and one of those M cliques would have an edge to one clique at time $t + 1$ and another of those M cliques would have an edge to one clique at time $t - 1$.

§8.9.1 also claimed that an elimination order obtained by taking any permutation of the variables to eliminate at time t before proceeding to time $t + 1$ will also be optimal. Also, while we used a strictly left-to-right elimination order, since the graph is temporally symmetric, we would have achieved the same results had we done a strictly right-to-left elimination procedure. Hence, we can derive generalized α and β recursions for the factorial HMM, except that we have a $2(M!)$ distinct recursion formula rather than just 2. We also saw that there was no benefit, and in fact only a detriment, to start the elimination process at any variable other than at the temporal extremes, assuming that $T \geq 3$. In other words, there is no advantage (and even a detriment) to doing anything other than a frame-by-frame elimination order for the factorial HMM.

$\hat{\tau} >= 0$. We still need not unroll for all $\hat{\tau}$.

²To clarify, suppose we have an HMM of the form $p(\bar{x}_{1:T}, y_{1:T}) \propto \prod_t \phi(\bar{x}_t, y_t) \phi(y_t, y_{t-1})$. Since \bar{x}_t is a constant for all t , we can absorb each factor $\phi(\bar{x}_t, y_t)$ into the factor $\phi(y_t, y_{t-1})$ as $\phi'(y_t, y_{t-1}) = \phi(y_t, y_{t-1})\phi(\bar{x}_t, y_t)$ giving a model $p(\bar{x}_{1:T}, y_{1:T}) \propto \prod_t \phi'(y_t, y_{t-1})$. Also see Figure 8.44.

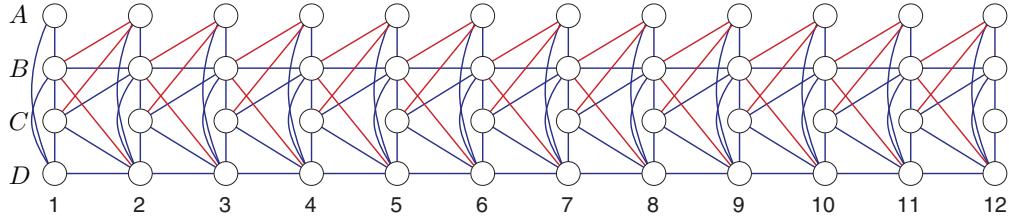


Figure 13.9: The MRF from Figure 13.7-middle unrolled to 12 frames.

13.3.1.2 Unroll and Compute with a more complex example

Consider next the DGM from Figure 13.7-middle that has been unrolled a number of times, as shown in Figure 13.9. Note again that the observations have been removed since they do not contribute to the state space, and the factor containing the observations can be absorbed into any factor containing the state variable S (as we did in Figure 8.44). There are four variables per frame and 12 frames in the figure — for easy naming of variables, the nodes now have a column name (the integer frame number) and a row name (A, B, C, or D). If we consider the elimination algorithm applied to this graph (seen as a static graphical model), there will be $48! \approx 10^{61}$ different possible elimination orders. As mentioned above, the goal of elimination is to produce the smallest clique in the reconstituted triangulated graph. With a dynamic graph we can rule out a number of elimination orders immediately. For example, consider starting the elimination at node D(6). That will immediately couple together its neighbors D(5), C(5), B(5), A(6), B(6), C(6), and D(7) leading to a clique of size 8. If we were to next eliminate C(6), that would yield yet another size-8 clique. Given that a naïve triangulation and junction tree consisting of a chain of cliques of the form $\{B(t), (C)(t), (D)(t), (A)(t+1), (B)(t+1), (C)(t+1), (D)(t+1)\}$ for $t = 1, 2, \dots$ has a maximum clique of size 7, eliminating node D(6) first seems like a poor initial choice. If we instead were to first eliminate A(6), this would yield the clique A(6), B(5), C(5), D(6), B(6) of size 5, but then if we next eliminated B(6) that would lead to a clique of size 8 and if we next eliminated D(6) it would lead to a clique of size 7.

The model above only has 12 frames, but in general an unrolling of a DGM could have orders of magnitude more frames than this. It seems intuitive, therefore, that since the model is ultimately going to look like a “horizontal line” of Figure 13.1c, some form of left-to-right inference procedure is most promising. This means that variables should be eliminated starting at the left and moving to the right (forward in time), or alternatively doing the opposite and starting from the right and moving to the left. For example, if we were to eliminate all variables in frame 1 in Figure 13.9, then no matter what the order, the largest clique will be no more than size 7.

If moreover we were a bit smarter about the elimination order, we could do much better. For example, consider the following elimination order: A(1), D(1), B(1), C(1), A(2), D(2), B(2), C(2), A(3), ..., then the largest clique is of size 5. Eliminating in right-to-left order of the form: A(12), C(12), B(12), D(12), A(11), C(11), B(11), D(11), ... also results in a size-5 maximum clique size. In both cases, we see a periodic pattern in the elimination ordering. In the left-to-right case, we eliminate nodes in slice t in order A, D, B, C before eliminating any nodes in slice $t + 1$. In the right-to-left case, we eliminate nodes in slice t in order A, C, B, D before eliminating any nodes in slice $t - 1$.

From the above example, it might seem like we can thus constrain the elimination order such that it always eliminates one slice of variables before the next one — the example shows that if we are performing a left-to-right elimination, we can eliminate all variables in time t before we eliminate any in time $t + 1$ (and analogously in the right-to-left case) and still have an optimal elimination order. This leads us to the following definition:

Definition 146 (Frame-by-frame elimination). *In a left-to-right frame-by-frame elimination order, for all t ,*

all variables in frame t are eliminated before we move on to frame $t + 1$. The right-to-left case is similar. We might also refer to this as slice-by-slice elimination.

Hence, in both the factorial HMM and here, a frame-by-frame elimination order was capable of producing the optimal elimination order. This is quite important as we've gone from having $(TN)!$ possible orderings to having only $N!$ possible orderings, where T is the length of the model and N is the number of variables per frame.

In the generalized DGM template, where we have a prologue, chunk, and epilogue, we can extend this definition so that we eliminate all variables within a section before moving onto the next section.

Definition 147 (Section-by-Section elimination). *In a left-to-right section-by-section elimination order, for all t , all variables in section t are eliminated before we move on to section $t + 1$. The right-to-left case is similar.*

Most of the time, since it is \mathcal{G}^c that is being repeated after an unrolling, section-by-section elimination is really chunk-by-chunk elimination. Also, if a chunk consists of a single frame, then section-by-section elimination is identical to frame-by-frame elimination.

This now begs two questions:

1. Can we always, for all models, eliminate all variables in time t before we eliminate any in time $t + 1$ (and analogously in the right-to-left case) in such a way that the elimination order is optimal?

If so, this would mean we need only consider elimination orders within a single slice or section, something that would significantly reduce the combinatorial explosion of elimination orders in an unrolled graph.

2. A DGM has factorization structure within each time slice (or within each section). The factorization properties make for better inference. If we do an elimination order that is left-to-right, can we be sure that this factorization is preserved in the elimination cliques that are encountered along the way?

We answer these questions below.

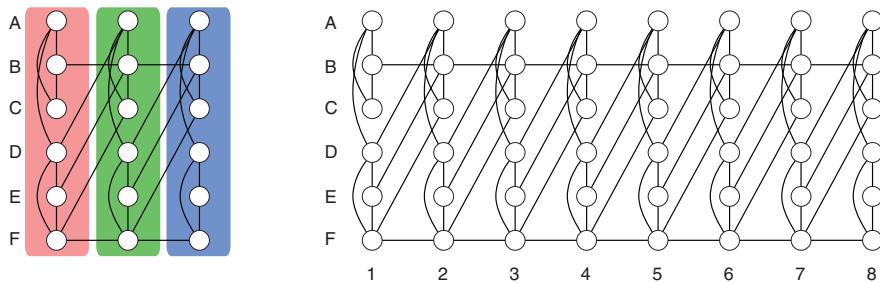


Figure 13.10: A DGM where eliminating one slice before the next (in either left-to-right or the right-to-left order) is not optimal. On the left is the DGM template and the right is the template unrolled to 8 frames. If we use only the left interface method or only the right interface method, the largest clique size in the optimal triangulation is 5. If, on the other hand, we find the optimal interface separator using the max-flow based vertex-cut procedure mentioned in the text, then the size of the largest clique in the optimal triangulation reduces to 4.

13.3.1.3 Where slice-by-slice elimination fails

We can answer question 1 right away. In fact, this need not be the case as shown in Figure 13.10. Here, no matter what the ordering, if we eliminate nodes at time frame t before starting to eliminate nodes in time

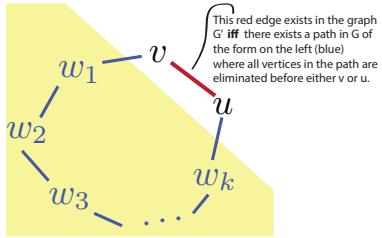


Figure 13.11: This red fill-in edge exists in G' iff there exists a path as shown by black edges where all vertices incident to two black edges (i.e., w_1, w_2, \dots, w_k) are eliminated before v and w . In general, any two vertices connected by a path between nodes eliminated earlier than the two nodes will end up with an edge.

$t + 1$ (or eliminating at slice $t + 1$ before slice t) the smallest maximum clique size will be of size 5. If the elimination order is allowed a bit more flexibility, the smallest maximum clique size is 4 — consider the elimination order $(C(1), A(1), B(1), E(1), D(1), C(2), A(2), F(1), B(2), E(2), D(2), C(3), A(3), F(2), B(3), E(3), D(3), C(4), A(4), F(3), \dots)$.

Exercise 148. Show that this corresponds to the best possible elimination order achievable in this graph.

Note that after we eliminated the first two nodes $C(1)$ and $A(1)$, the pattern $B(t), E(t), D(t), C(t+1), A(t+1), F(t)$ for $t \geq 1$ repeats. In this case, the optimal periodic pattern did not appear until we allowed a less restrictive elimination order to occur, and also we needed to eliminate a few “burn in” nodes at the left of the graph. Hence, it is not always optimal to consider only slice-by-slice (or section-by-section) elimination orders.

Question 2 also has an answer that seems worrisome, however, and the reasons are as follows. On the one hand, slice-by-slice elimination might seem beneficial since there are many fewer permutations of nodes to consider to eliminate than otherwise (as mentioned above). On the other hand, there are unfortunately certain detrimental and unavoidable consequences to doing this, and this relates to how an elimination order adds edges to a graph. In a graphical model, it is the missing edges that provide the “structural” property in the family of distributions, and that allow for efficient inference schemes to be derived. When elimination is performed, it can only add edges which in turn can only reduce structure. Unfortunately, when performing such elimination, some and perhaps even quite a lot of the underlying structure is lost. This follows from a critically important theorem in the field of DGMs:

Theorem 149. Rose’s Entanglement Theorem (Lemma 4 in [367]).

Let $G = (V, E)$ be an undirected graph with a given elimination ordering $\sigma : \mathbb{Z}_+ \rightarrow V$ that maps G to $G' = (V, E')$ where $E' = E \cup F_\sigma$, and where F_σ are the fill-in edges added during elimination with order σ . Then $(v, w) \in E'$ is an edge in G' iff there is a path in G with endpoints v and w , and where any nodes on the path other than v and w are eliminated before v and w in order σ . TODO: note that this theorem is also given as Theorem 52, should only have one version of the proof.

Hence, if there is a path $(v = v_1, v_2, \dots, v_{k+1} = w)$ in G such that

$$\sigma^{-1}(v_i) < \min(\sigma^{-1}(v), \sigma^{-1}(w)), \text{ for } 2 \leq i \leq k \quad (13.1)$$

then then $(v, w) \in E'$ is an edge in G' , where σ is the elimination order (integers to nodes) and σ^{-1} maps from nodes to their elimination order number. Some call this the *entanglement* theorem, and it is depicted in Figure 13.11.

Proof of Theorem 149. Only if: We show by induction on $l = \min(\alpha^{-1}(v), \alpha^{-1}(w))$ that given an edge v, w in G_α^* with such an l , there exists a v, w chain with the desired property (property in the equation above). An edge v, w in G_α^* can either have been in G_α , or it could only be in G_α^* after elimination according to α has taken place.

Start with $l = 1$. I.e., if edge (v, w) (with such an l) is in G_α , then it is also in G_α^* . In such case, we can define the chain as $\mu = [v, w]$ where the property clearly holds (w.l.o.g. we may assume $\alpha^{-1}(v) = 1$).

If edge (v, w) (with such an l) is not in G_α , then it must be in $F(G_\alpha)$, a fill-in edge. But with $l = 1$, one of v or w is eliminated first which means that the edge could not exist. Therefore, this case can't occur.

Next, assume result holds for $l \leq l_0$ and consider the case $l = l_0 + 1$.

If v, w is in G_α , then again the result clearly holds since again we can define the chain as $\mu = [v, w]$.

If otherwise, $(v, w) \in F(G_\alpha)$, then we have by the definition of $F(G_\alpha)$ the existence of an $x \in V$ such that $\alpha^{-1}(x) < \min(\alpha^{-1}(v), \alpha^{-1}(w))$ and that edges (x, v) and (x, w) exist (i.e., the edge (v, w) was created during the elimination of some node $x \in V$ via edges (x, v) and (x, w)). We also have that (x, v) and (x, w) are in G_α^* as well (since they must exist since the elimination of x created the edge (v, w)). Since $\min(\alpha^{-1}(v), \alpha^{-1}(x)) < \min(\alpha^{-1}(v), \alpha^{-1}(w))$ (and a similar inequality for w and x on the l.h.s.), the induction hypothesis means that there exists satisfying x, v and x, w chains in G_α . Combining these chains gives the required v, w chain.

To prove the converse (a chain in G_α implies an edge), we do it by induction on k the length of μ . If $k = 1$, then v, w are connected in G_α , so are they in G_α^* . Suppose it holds for $k \leq k_0$ and consider case $k = k_0 + 1$. Let $\mu = [v = v_1, v_2, \dots, v_{k+1} = w]$ and choose $x = v_i$ where $\alpha^{-1}(v_i) = \max\{\alpha^{-1}(v_j) | 2 \leq j \leq k\}$. The induction hypothesis implies that edge (v, x) and (x, w) exist in G_α^* , and since x is the last (max order) node, those two edges means that edge (v, w) will be created in G_α^* when x is eliminated. \square

Thus, if there is a path between two vertices, say A_t, B_t where the path goes through nodes that exist at times strictly earlier than t , then those two nodes will end up with an edge if we do a slice-by-slice elimination order. If all vertices at time t have a path between them consisting of nodes strictly in the past, then all the vertices at time t become a clique. This will couple together variables that otherwise might have nicely factored (since all the vertices of a slice are typically not a clique like in the vectorial HMM of Figure 8.69b).

It consequence, therefore, is that slice-by-slice interface orders have a set of vertices that must become complete (and hence are a subset of a maxclique) — such vertices therefore become “compulsorily completed,” something we call a “compulsory clique.” We also call any such edges that are added to the DGM compulsory edges. Such edges are distinct from the “moralization edges” (the ones that are due to graph moralization of a Bayesian network), and “triangulation edges”, the ones that are caused by a triangulation of the graph.

This means that, in such graphs, a quick-and-dirty lower bound on the exact inference complexity with slice-by-slice elimination is exponential in the number of nodes in a given time slice that are connected to the previous slice. Since these vertices are completed and thus form a clique in any slice-by-slice elimination scheme, exact inference can not cost any less than this.

As a convention, we will show all original graph edges as black, edges due to the moralization process in a Bayesian network as green, edges due to a graph triangulation (fill-in edges) as red, and compulsory edges due to interface completion as blue.

Now in some cases, the compulsorily completed set of vertices are ones that would be completed anyway in the optimal elimination order and hence a slice-by-slice or section-by-section elimination order is no worse than any other elimination order. This is the case, for example, with the factorial HMM we saw in Figure 13.8 discussed in §8.9.1. Another example of this case is discussed above for the example shown in Figure 13.9.

But in other cases, Figure 13.10 for example, we saw that slice-by-slice elimination fails. In such cases, the compulsorily completed set of vertices when doing a slice-by-slice (section-by-section) elimination order is larger than if we were to relax the elimination order and allow some variables to be eliminated in later time slices/sections before all variables in the current slice/section.

One particularly egregious example of how this is a problem is given in Figure 13.12. Here, we can see that the template corresponds to a model that, when unrolled, is always a 1-tree, which means that exact inference should be possible in $O(r^2)$ where r is the cardinality of each random variable. If we were to per-

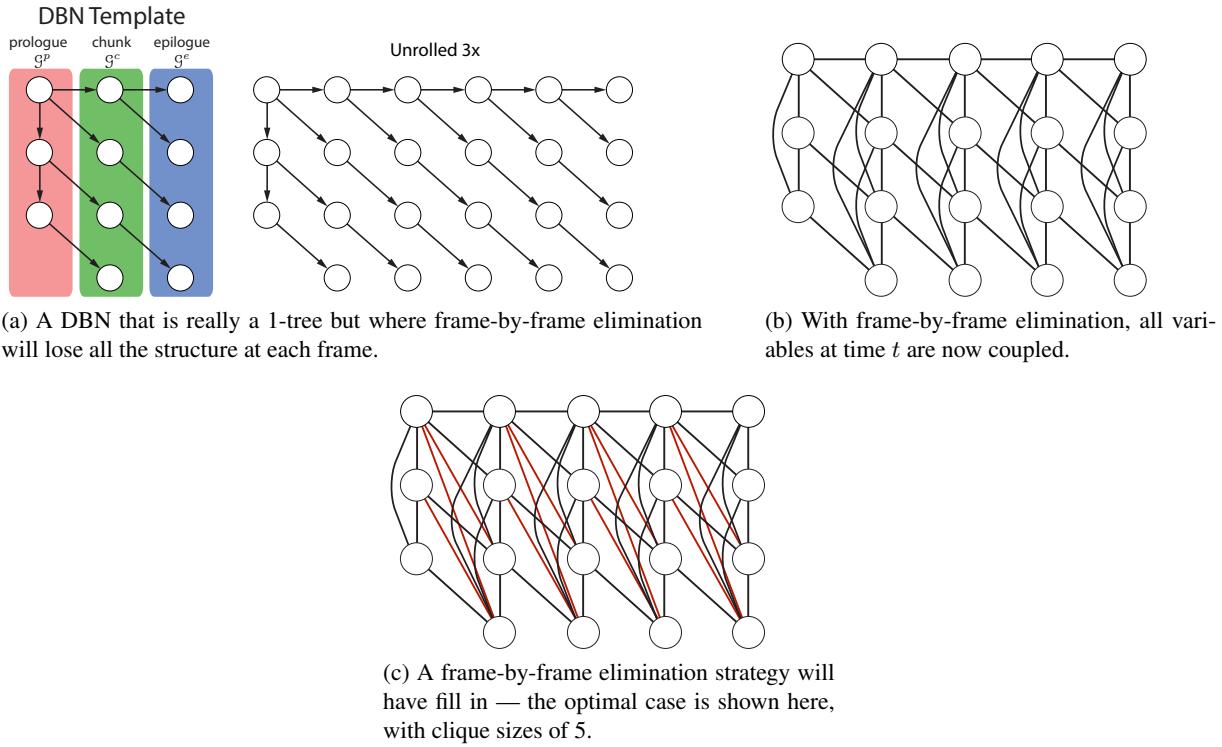


Figure 13.12: An example template for which the consequences of Rose’s theorem and a frame-by-frame elimination strategy is particularly egregious.

form a strict slice-by-slice elimination, then by Rose’s theorem, all variables within a given time slice would become coupled as shown in Figure 13.12b. When actually doing a frame-by-frame elimination scheme, we will get additional fill-in edges, leading to a clique size of five, as shown in Figure 13.12c. Hence, this example shows that a slice-by-slice elimination will go from what the optimal inference strategy would be $O(Tr^2)$ to something much worse $O(Tr^5)$. This example, more over, can be extended to any number, say M , of variables per frame, and the slice-by-slice elimination in such case would have complexity $O(Tr^M)$.

Our goal is to still be able to extract the inherent 1-tree property of Figure 13.12 (and those templates like it, for example Figure 13.10, and to do this all within the DGM template framework without having to unroll and compute as in §13.3.1, where we unroll by the desired amount to achieve length T and then find some elimination order within the unrolled graph.

Moreover, we want a more systematic way to identify the periodicity in the ordering, and one can then use this periodicity to form a chunk of junction tree which itself can be unrolled to any length. Also, we haven’t yet discussed memory implications of the elimination schemes above. Like in the HMM case, where computational cost is $O(N^2T)$ and memory cost is $O(NT)$, we wish to achieve good both computational and memory performance.³ And we want to produce an inference procedure based only on the template without first needing unbounded unrolling to figure this out.

It turns out that the two questions above (Question 1 and Question 2) are themselves related. That is, by relaxing the requirement that we do a slice-by-slice or section-by-section elimination order, we can reduce the size of the compulsory clique, sometimes quite significantly. The next section shows how this can be done.

³We moreover want to achieve high accuracy in any approximate inference scheme as well.

13.3.2 Finding Periodicity

Given any periodic signal $s(t)$, one has a choice as to where the beginning and ending of a period should be. That is, we can pick an arbitrary time point t_0 and if T_0 is (what is known in the signal processing literature as) the fundamental period, we have that

$$s(t_0) = s(t_0 + i(kT_0)) \quad (13.2)$$

for any integers i and k . That is, any periodic signal with period T_0 is also periodic with period kT_0 for any positive integer k . With $k = 1$, the fundamental period T_0 is the smallest value where Equation (13.2) still holds for all i . Normally, the choice of t_0 is arbitrary, and $k \geq 1$ can be any value for the equation to hold. However, we (especially in signal processing) typically wish to use the smallest possible integer $k = 1$ leading to the fundamental period.

DGMs also have a period — after all, the template is unrolled and once it is unrolled enough we start seeing a repeating pattern. For example, consider again the DGM that has been unrolled a large number of times, as shown in Figure 13.9. As shown, the unrolled graph will have a periodicity, meaning that a group of nodes will repeat over and over with increasing t . As we repeat copies of \mathcal{G}^c , then \mathcal{G}^c itself defines the beginning and ending of a period — the vertices in \mathcal{G}^c that are incident to vertices in the previous chunk are the beginning of the period, and the vertices in \mathcal{G}^c who are incident to edges that connect to the right chunk, along with those incident edges, constitute the end of the period. So an unrolled DGM can in some sense be seen as a periodic graph, as further demonstrated in §13.3.1.

When we do a left-to-right elimination strategy or a right-to-left elimination strategy, the elimination orders ultimately themselves can become periodic as well. This suggests that whatever processing that is done, it can be decided upon only once, within a periodic segment, and then reused for all repeated segments corresponding to an unrolling. We can view the problem as excising (or extracting) a portion of the fully unrolled graph based on flanking splice points, and determining an inference strategy only once within that excised portion. When that portion is repeated in time, the cost of determining the inference strategy within the excised portion is amortized over the entire length of the graph — the degree of amortization increases with the length T .

The question then is as follows: what periodic portion should we use and how to find it? In the case of DGM inference, in fact, the location of the start (like t_0) and end of a period, and the use of periods other than the fundamental period (analogous to the case where $k > 1$ in Equation (13.2)) can have significant consequences for inference. In this section, we address this issue.

Now as mentioned above, since the model is likely going to be much wider than it is high, some form of left-to-right (or right-to-left) inference procedure will be optimal. The reason for this is if we start eliminating nodes in the middle having incident edges on both the left and right, this will produce much larger clique sizes. It is important to realize, however, that a *strictly* left-to-right or right-to-left elimination order need not be optimal. As we will see below (or above, in the unroll and eliminate case), there are cases where the elimination order need not be strictly either monotonically increasing or decreasing in terms of time — what is meant here is that while the elimination order might not be exactly temporally monotone (increasing or decreasing), it will still be constrained in that there will be a fixed time window (say δ) such that no variables will be eliminated at time $t + \delta + 1$ before all variables at time t are eliminated. We call this *bounded monotone* and it means that if t_k and t_{k+1} are the time points of two variables that are successively eliminated, then $t_{k+1} - t_k \leq B$ for some finite value B ($B = 1$ for a strictly left-to-right order, and $t_k - t_{k+1} \leq B$ for right-to-left order).

13.3.2.1 Modified Sections

Just a word on terminology. We started with three sections, the prologue \mathcal{G}^p , chunk \mathcal{G}^c and epilogue \mathcal{G}^e . What we will do is transform this template into a modified template that consists of three modified sections,

a (modified) prologue, chunk, and epilogue. Several things will be different about this modified template:

1. the modified sections will not be a partition, meaning that two sections can have a non-empty intersection;
2. the modified sections might consist of sub-portions of an original section;
3. the modified sections might consist of multiple copies of the original sections, i.e., the modified chunk might consist of multiple original chunks from a limited extent unrolled model;
4. all of the above might simultaneously be true.

Moreover, there might be multiple stages of processing that the template goes through before it reaches its final (modified) state. In each case, we will call refer to the modified sections as the modified prologue (respectively modified chunk, modified epilogue), and use the notation $\hat{\mathcal{G}}^p$ (respectively $\hat{\mathcal{G}}^c$, $\hat{\mathcal{G}}^e$) to refer to these.

There are several reasons that the original template sections might become modified, and they are discussed at length in the sections below, but we here summarize them for easy reference.

- The sections will be modified based on either the left or the right interface of the sections. The left (respectively right) interface are those vertices that communicate directly to adjacent section on the left (respectively right). This is described in §13.3.2.2, §13.3.2.3, §13.3.2.4, and §13.3.2.5.
- The sections might be modified in response to the vertex cut span M parameter and the chunk skip S parameter. The sections, moreover, might be modified based on finding an optimal vertex cut between successive sections (which depend on M and S). This is described in §13.3.2.9.
- The sections might be modified in order to ensure that there is a first-order Markov property at the level of the sections. That is, in any unrolled model, we are certain that if we condition on the variables in a section (or a separator between two successive sections), then the past and future will be rendered independent. This is described in §13.3.4.

13.3.2.2 Left Interface

First consider left-to-right slice-by-slice elimination. After we are finished eliminating vertices at time t , the vertices on the right in slice $t + 1$ that are compulsorily completed are in some sense an interface to the next slice, and in fact are an interface to all future variables. In fact, those vertices that have been completed form a separator between those variables at and before time t and those variables at or after time $t + 1$. It could be that the compulsorily completed vertices consists of the entire right slice as in the factorial HMM Figure 13.8, but it might also be a subset of the right slice as shown in Figure ?? or Figure 13.10. Its pretty easy to see that the compulsory completed nodes on the right in slice $t + 1$ consist of all vertices in slice $t + 1$ that are incident to an edge going between slice t and slice $t + 1$.

Before we make this more formal, we offer a bit more notation. Given an edge consisting of times nodes $e = (u, v) \in E(V)$, with $t(u) < t(v)$, define two operators on edges, one that returns the the vertex incident to edge e that is later $\partial^>e = v$, and one that returns the the vertex incident to edge e that is earlier $\partial^<e = u$. If $t(u) = t(v)$ then $\partial^>e = \partial^<e = \emptyset$. Also, given such an edge e with distinctly timed nodes, then $(\partial^<e, \partial^>e)$ is the edge with the nodes ordered in increasing time even if the original edge had vertices ordered by decreasing time.

Next, if $E' \subset E(G)$, then define $\partial^>(E') = \{v \in V(G) : (u, v) \in E' \text{ and } t(v) > t(u)\}$ as the set of nodes that are at a later time then then some other node in the nodes of E . Also, if $E' \subset E(G)$, then define $\partial^<(E') = \{v \in V(G) : (u, v) \in E' \text{ and } t(v) < t(u)\}$ as the set of nodes that are at an earlier time then then

some other node in the nodes of E . Note, $\partial^<(E')$ and $\partial^>(E')$ are not necessarily disjoint although they might be. This notation gives us an easy way to get the temporally related nodes from a set of edges.

Lets first consider the nodes of \mathcal{G}^c that interface on the left. Recall, we've got a DGM template $G = (V, E)$ with $V = (V^p \cup V^c \cup V^e)$ and for now assume that $E = (E^p \cup E^c \cup E^e \cup E^{pc} \cup E^{cp} \cup E^{ce} \cup E^{ec})$ (that is, for the moment we assume $E^{pe} = E^{ep} = \emptyset$ in Equation (??), but we address this case again in §13.3.4). Define $V^{c,L} = \partial^>(E^{cp} \cup E^{pc})$ to be the “left interface” nodes. These are the nodes of a chunk that interface to the chunk’s neighboring chunk immediately on the left. Indeed, if we perform a strictly left-to-right, section-by-section elimination, then for all but trivial graphs⁴, $V^{c,L}$ will become a clique. We can see a template and the left interface within a chunk in Figure 13.13.

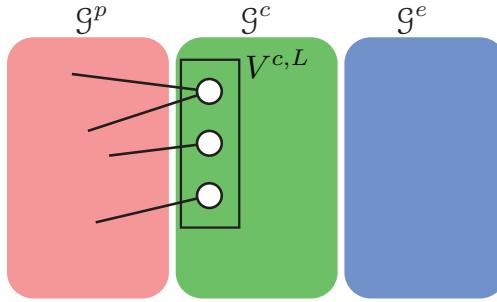


Figure 13.13: Basic left interface. The set of nodes enclosed in a box and labeled $V^{c,L}$ is the chunk’s left interface to the section immediately on the left. In this case, we show the template so $V^{c,L}$ is \mathcal{G}^c ’s left interface to \mathcal{G}^p although in an unrolling, $V^{c,L}$ is \mathcal{G}^c ’s left interface to other chunks.

When all nodes earlier than chunk $t + 1$ are eliminated and when there is one connected component per chunk, there will be a set of nodes that are forcibly completed in chunk $t + 1$, namely those nodes entirely in chunk $t + 1$ that have neighbors in chunk t . In a DBN those nodes have been called the *interface* [244, 246, 104], *backward interface* [448], or the *incoming interface* [313]. In the context of DGMs, we will call this simply the “left interface”.

Definition 150. *The set of vertices in a section (either a chunk or an epilogue) that are neighbors to a section on the left is connected is called the section’s left interface.*

The reason the set is called the left interface is because it consists of vertices that connect to the section on the left. Note, however, that the left interface may also be defined as the nodes within the right adjacent section that are neighbors to some vertex in the current section. One can think of this as follows: we connect to the right adjacent section via its left interface. This hopefully will not cause confusion.

Given any graph or subgraph $G = (V, E)$, define the completed graph operator $\text{comp } G = (V, V \times V)$. That is, $\text{comp } (\cdot)$ takes a graph and adds fill-in edges between all vertices to create a complete graph.

We now define a modified chunk that consists of a chunk and its right adjacent chunk’s left interface. That is, we add to the chunk all of the vertices that are adjacent to any vertex in a chunk. We use the complete operator, and define modified sections of the template G that consist of original sections of the template with some of the vertices completed and some additional nodes consisting of the left interface of a sections to the right. Define the modified template as $\hat{G} = (\hat{\mathcal{G}}^p, \hat{\mathcal{G}}^c, \hat{\mathcal{G}}^e)$ where

$$\hat{\mathcal{G}}^p \triangleq \mathcal{G}^p \cup \text{comp } \mathcal{G}^c[\partial^>(E^{pc} \cup E^{cp})] \quad (13.3)$$

$$\hat{\mathcal{G}}^c \triangleq \mathcal{G}^c \cup \text{comp } \mathcal{G}^c[\partial^>(E^{pc} \cup E^{cp})] \cup \text{comp } \mathcal{G}^e[\partial^>(E^{ce} \cup E^{ec})] \quad (13.4)$$

$$\hat{\mathcal{G}}^e = \mathcal{G}^e \cup \text{comp } \mathcal{G}^e[\partial^>(E^{ce} \cup E^{ec})] \quad (13.5)$$

⁴One example of such graphs have multiple connected components for any unrolling.

In words, the modified prologue is the original prologue and adds the chunk's left interface, and all the associated incident edges. The modified chunk is the original chunk with its left interface completed, and adds the epilogue's left interface (along with any associated incident edges). The modified epilogue consists of the original epilogue with its left interface completed.

The completed sections of the template are shown in Figure 13.14a which also shows $V^{e,L}$ the left interface of \mathcal{G}^e . The key issue is that if we do section-by-section elimination from left-to-right, we will have both $V^{c,L}$ and $V^{e,L}$ completed by Rose's theorem. Therefore, if we want to separate out this process, we can form new division of the template into modified sections where the nodes that are completed are placed in their cluster. The three modified sections $\hat{\mathcal{G}}^p$, $\hat{\mathcal{G}}^c$, and $\hat{\mathcal{G}}^e$ (with completed vertices and vertex augmentations) are shown in Figure 13.14b. Note that unlike the original (unmodified) sections \mathcal{G}^p , \mathcal{G}^c , and \mathcal{G}^e which form a partition of the template (so no vertex in common to any two of \mathcal{G}^p , \mathcal{G}^c , and \mathcal{G}^e), the modified sections no longer partition the template since they will have overlap whenever the interfaces are non-empty.

Note that in many of the figures below, we will show the modified sections that consist of the original sections as well as the interfaces. The original sections will be shown in slightly darker shaded regions, while the modified sections will be shown as the original illumination of shaded regions. Hence, in the modified sections, the interfaces are shown in the shaded within the original illumination. Figure 13.14 is an example of this.

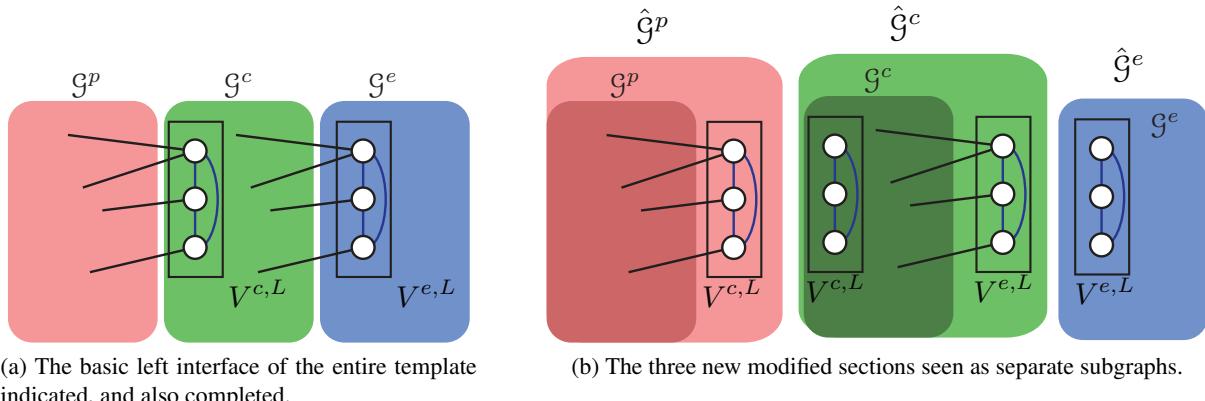


Figure 13.14: Basic left interface and modified sections. In the case of the modified section, the word “left” might here be a bit confusing. Note that the “left” in left interface corresponds to the vertices that are incident to the section on the left. Once we modify a section, however, we augment it with the left interface of the section on the right. Hence, in the left interface method, a section gets augmented with vertices on the right, and those vertices on the right are the left interface of the neighboring section on the right.

Note that $V^{e,L}$ is in $\hat{\mathcal{G}}^c$. This is because once we eliminate all of \mathcal{G}^c , we'll have all of $V^{e,L}$ completed. For this template to make sense, we must have that

$$V^{e,L} = V_{\Delta(T(\mathcal{G}^c))}^{c,L}. \quad (13.6)$$

The reason is that in an unrolling, a chunk never knows if a chunk or if an epilogue is to the immediate right of it.

After the above completion operations are done, and the modified sections are formed, we are free to triangulate each of $\hat{\mathcal{G}}^p$, $\hat{\mathcal{G}}^c$, $\hat{\mathcal{G}}^e$, since such a process only adds edges (the interfaces will still be cliques once any additional edges are added). In fact, any static graphical model inference method (exact or approximate) is applicable that starts by receiving information in the form of a message at the left interface on the left of the modified section, and propagates that information to the left interface on the right of the modified

section. The interfaces, moreover, become minimal separators in the resulting triangulated graph — each interface renders the past and future (relative to the interface) independent.

13.3.2.3 Right Interface

Symmetrically, let's consider next a left-to-right slice-by-slice elimination order. After we eliminate vertices at time slice $t + 1$, the vertices in slice t that are compulsorily completed due to Rose's theorem are an interface to slice t — they form a separator that separates any vertices at or before time t from any vertices at time $t + 1$ or above. It could be that the compulsorily completed vertices consists of the entire left slice at time t as in factorial HMM Figure 13.8, but it might also be a subset of the left slice as shown in Figure ?? or Figure 13.10. Again, it is easy to see that the compulsory completed nodes in slice t consist of all vertices that are incident to an edge going between slice t and slice $t + 1$.

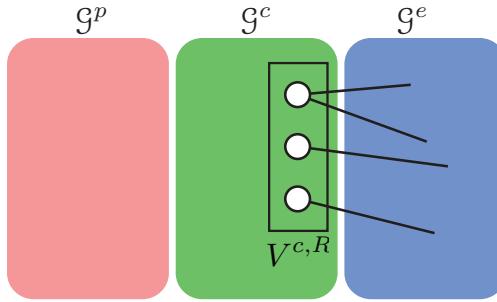


Figure 13.15: Basic right interface. The vertices marked $V^{c,R}$ is the right interface since they consist of the set vertices within \mathcal{G}^c that interface between \mathcal{G}^c and \mathcal{G}^e . In an unrolling, moreover, $V^{c,R}$ consists of the set of vertices that interface between a section and its right adjacent section. If we were to do a strictly section-by-section right-to-left elimination order, then for all but trivial graphs, the vertices within $V^{c,R}$ would be completed.

More formally, let's consider the nodes of \mathcal{G}^c that interface on the right. Define $V^{c,R} = \partial^<(E^{ce} \cup E^{ec})$ to be the “right interface” of the chunk. These are the vertices of the chunk that interface to the chunk’s right. If we do strictly right-to-left, section-by-section elimination, then for all but trivial graphs, $V^{c,R}$ will become a clique, again by Theorem 149. We can see a template and the left interface within a chunk in Figure 13.15. These are the nodes in section t that have neighbors in section $t + 1$. When sections correspond to time slices, the right interface has been called the *forward interface* [448, 104] or *outgoing interface* [312]. We will call this simply the “right interface” and is again a separator in the DGM rendering its leftward and rightward variables conditionally independent.

Definition 151. *The set of vertices in a section (either a chunk or an epilogue) that are neighbors to a section on the right is connected is called the section’s right interface.*

The reason the set is called the right interface is because it consists of vertices that connect to the section on the right.

We can define a modified chunk that consists of a chunk and its left neighbor’s right interface. That is, define modified subsets of G that consist of each section of the template along with the right interface of its left section. Define $\hat{G} = (\hat{\mathcal{G}}^p, \hat{\mathcal{G}}^c, \hat{\mathcal{G}}^e)$ where

$$\hat{\mathcal{G}}^p = \mathcal{G}^p \cup \text{comp } \mathcal{G}^p[\partial^<(E^{cp} \cup E^{pc})] \quad (13.7)$$

$$\hat{\mathcal{G}}^c = \mathcal{G}^c \cup \text{comp } \mathcal{G}^c[\partial^<(E^{ce} \cup E^{ec})] \cup \text{comp } \mathcal{G}^p[\partial^<(E^{pc} \cup E^{cp})] \quad (13.8)$$

$$\hat{\mathcal{G}}^e = \mathcal{G}^e \cup \text{comp } \mathcal{G}^e[\partial^<(E^{ce} \cup E^{ec})] \quad (13.9)$$

The right interface idea is therefore entirely symmetric to the left interface idea. The completed sections of the template are shown in Figure 13.16a which also shows $V^{p,R}$ the right interface of \mathcal{G}^p . The three modified sections $\hat{\mathcal{G}}^p$, $\hat{\mathcal{G}}^c$, and $\hat{\mathcal{G}}^e$ (with completed vertices and vertex augmentations) are shown in Figure 13.16b, and again no longer partition the template.

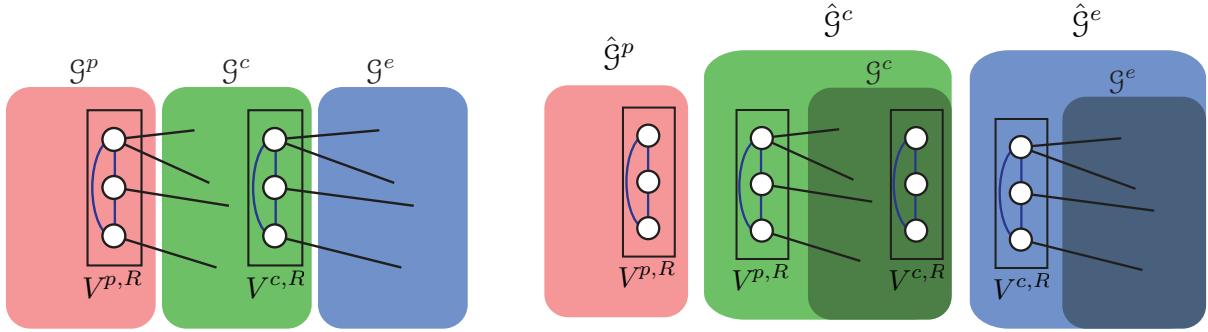


Figure 13.16: Basic right interface and modified sections. See Figure 13.14 for a symmetric discussion on how we use the word “right” in the case of the modified sections.

13.3.2.4 Left or Right Interface, which to use?

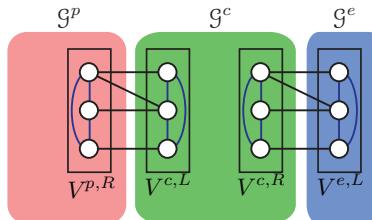


Figure 13.17: Template with both the left and the right interfaces indicated.

The above suggests that we can use either the left or the right interface. We give a number of examples of left and right interface in the following sections, and in particular show when they are different in §13.3.2.6. The question we address in this section is, which one do we use?

First of all, it is important to realize that we can still do left-to-right inference/elimination even if we use the right-interface method, and can still do right-to-left inference/elimination even if we do the left-interface method, so this doesn't influence our choice. The reason is that for each kind of interface, the interface consists of the compulsorily completed vertices that arise when we do a section-by-section elimination. When we form the modified sections, the interfaces are the conduit through which messages are sent both into a modified section and then are sent back out of a section. Since in each case, it is the interface which is a separator (creating a temporal Markov property) each resulting modified partition can be treated separately.

We have the following theorem.

Theorem 152. *Either of the interfaces (either the left or the right interface) render, when conditioned on, those variables on the left and on the right conditionally independent.*

Indeed, the fact that the interface is a separator should help us determine which interface method, left or right, should be used. We should use the interface that is smaller or cheaper in some way, since that is a bound on the memory usage, and is also a lower bound on computational cost.

Firstly, we address the question why the interface gives us the exact memory cost — the interfaces are the units that should be stored when moving over the sequence in any implementation. Recall that in a DGM, we wish to do a forward pass and then a backwards pass. When we do the backwards pass, we need information from the forward pass to compute certain constructs (such as clique posteriors). When doing the forward pass, an inference method will typically repeat the following two-step procedure: 1) expand a chunk, 2) project down to interface. To minimize memory, it is only the interface that needs to be stored between sections, not the expanded chunk (and its cliques) within the sections (this can easily be recomputed again at each time when the backwards pass has been performed). We note that this process is a generalization of the standard HMM case where the “section” consists of two successive variables, and the interface consists of only one variable, as shown in Figure 13.18 and Figure 13.19. Moving in the right-to-left direction is the same pattern, and uses the expanded chunk to produce clique potentials.

To make this point more clear, suppose that an interface consists of three variables $\{A_t, B_t, C_t\}$, while the chunk consists of variables $\{A_t, B_t, C_t, D_t, E_t, F_t\}$. The memory required for each chunk when going forward is proportional to the state space of the interface, namely: $|D_{A_t}| \times |D_{B_t}| \times |D_{C_t}|$ rather than the much more expensive $|D_{A_t}| \times |D_{B_t}| \times |D_{C_t}| \times |D_{D_t}| \times |D_{E_t}| \times |D_{F_t}|$. If there is determinism amongst the set of random variables, or many zeros in tables over probability scores, then these memory cost estimates are actually upper bounds. In any event, it is best to choose the interface that results in the smallest upper bound on memory usage.

Regarding computational complexity, when we complete the interfaces, we are adding edges, reducing structure, and potentially increasing computation. Completing the smaller of the two interfaces leads to a smaller compulsory clique, thereby reducing a *lower bound* on inference cost.

Hence, a reasonable heuristic is to choose the interface method, left or right, that is smaller, or one that has a smaller state space. Indeed, the interface heuristic is a very important part of the process of deriving a method for inference in DGMs — like with a triangulation, the choice between a good and a poor interface can mean the difference between many orders of magnitude higher or lower computational and memory costs.

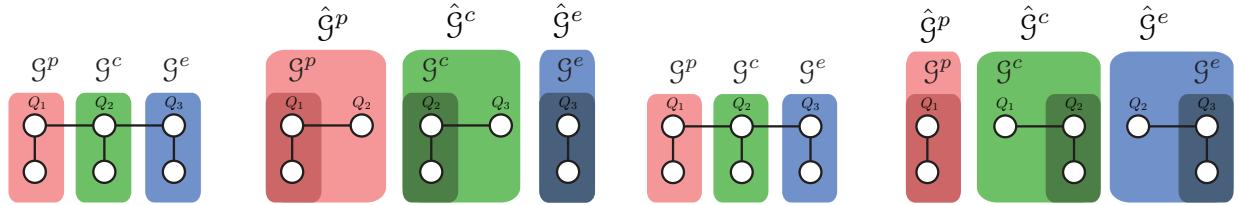
13.3.2.5 Examples of left and right Interfaces

Our first example is for the left interface with the simple HMM. An HMM’s left interface in fact consists of a single vertex, and that is the state variable to the right of the current frame. In this case, there is no completion of nodes (since the interface is a single node). Thus, the left interface method is optimal since it results in no additional compulsory edges. The resulting chunk \tilde{G}^c has two state variables in it as shown in Figure 13.18a

We next consider the right interface case, again with a simple HMM. This is shown in Figure 13.18b. Again, the chunk \hat{G}^c has two state variables in it. Also, there is again no completion of nodes (since the interface is a single node).

Thus, in the case of the HMM, there is no reason to prefer the left or the right interface method as they are identical to each other both in terms of memory and compute costs. In each case, we can easily optimally triangulate each modified section (each sections still has a tree-width of one), form a junction tree from each section, and glue them together as needed in an unrolling. That is, in an HMM, we take a vertex in a time slice and the nodes that connect to it either from the right (Figure 13.18a, the *left interface* method) or from the left (Figure 13.18b, the *right interface* method) to form the periodic section — this is shown in Figure 13.19. The left and right interface methods are identical, and leads to the same junction tree segment.

We also note that memory costs in an HMM is exponential in the size of the interface (namely, $O(TN^1)$)



(a) Left interface for the basic HMM. On the left is the template for an HMM. On the right is shown the modified partitions $\hat{\mathcal{G}}^p$, $\hat{\mathcal{G}}^c$, and $\hat{\mathcal{G}}^e$ for the left interface case. Here, the interface is a single node so the compulsory completion of the interface does not lead to any further edges being added. Therefore, the left-interface method is optimal for an HMM.

(b) The right interface case for the HMM. Note the template (on the left) is the same as in the left interface case. Here, the modified section is the original section augmented with the right interface that is within the section on the left.

Figure 13.18: Basic left and right interface case for the HMM, and the set of modified sections.

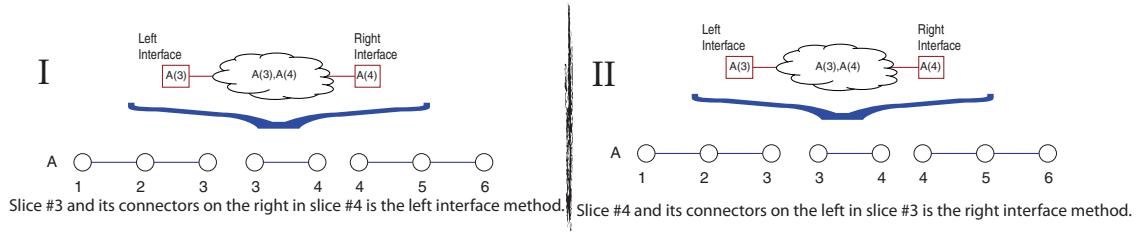


Figure 13.19: The HMM interface. On the left we have the left interface case, where we see the junction tree resulting from the modified section $\hat{\mathcal{G}}^c$ as well as the interfaces on the left and right side of $\hat{\mathcal{G}}^c$. The right interface case (shown on the right) is clearly identical. Compare this with Figure 8.44 as described in the text.

since the interface is of size one) and the compute cost is exponential in the size of the modified chunk (namely, $O(TN^2)$ since there are two vertices in the modified section). In fact, it is important to compare the example in Figure 13.19 with the HMM junction tree that is shown aligned to the HMM state trellis in Figure 8.44 in §8.4.5. Indeed, the modified section will correspond to the section that is repeated in the trellis. The interfaces of the modified section (or equivalently, the size of the separators in that junction tree, which is of size one), correspond to the exponent of the memory requirement of the method (the exponent is one in the HMM case leading to the $O(TN^1)$ memory requirement). The separator in the HMM junction tree (which is a single state variable) corresponds precisely to the interface in the DGM junction tree. In fact the interface is a separator in the underlying MRF, rendering those variables on the left and right conditionally independent, analogous to how a state variable in an HMM renders the left and right conditionally independent. We will sometimes use the phrases *interface separator* or just *interface* interchangeably. The complexity of doing inference at each time step is determined by the complexity of doing inference within the modified section (which in the HMM case consists only of a clique of size two). In the HMM case, the clique size of two in Figure 13.19 corresponds to the cliques in the junction tree in Figure 8.44 and corresponds also to the cross-edges in the trellis grid in Figure 8.44.

This will be a general property of these methods, where the exponent of the memory cost will exactly correspond to the interface size (or state space) and the exponent of the compute cost will be lower bounded by the interface size and upper bounded by the number of variables in the modified sections. We say “upper bound” here as it will in some cases be possible to triangulate the modified sections so that it turns not into one monolithic clique (see below).

Our next example is the factorial HMM (cf. §8.9.1) and is the left interface case. The resulting modified

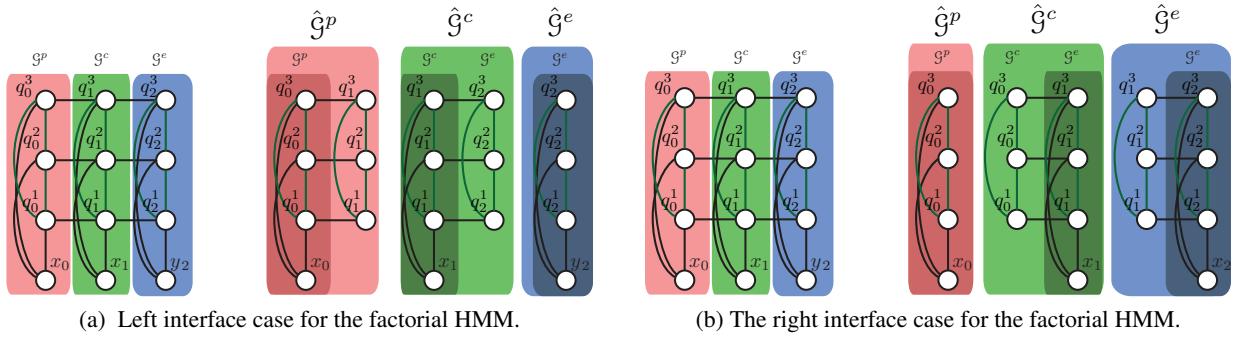


Figure 13.20: Basic left and right interface case for the factorial HMM. The edges due to moralization are shown in red. The compulsory edges are not seen since in this case, they correspond to edges that are already moralization edges. Moralization edges are shown as green.

chunk $\hat{\mathcal{G}}^c$ contains two successive state vectors corresponding to two successive time steps. The left interface in this case is a vector of vertices (in fact, all of the state variables). For example, in Figure 13.20a, we see that $\hat{\mathcal{G}}^c$ consists of \mathcal{G}^c along with the variables $\{q_0^1, q_1^2, q_2^3\}$. The interface variables, and a version of them shifted in time to the left by one frame, must be completed due to Rose's theorem (Theorem 149). Fortunately, the interface nodes are already complete due to the moralization step of the FHMM, so the compulsory completion has not hurt us in any way. Hence, like in the HMM, there is no additional edge fill-in. We can optimally triangulate each modified section, and once glued back together, we will recover the optimal $O(\ell r^{\ell+1} T)$ inference. That is, the optimal section triangulation, when pieced together with the rest of the model leads to an optimal exact inference algorithm for factorial HMMs. This is good news since it means that, for this model, there is no penalty for creating the modified sections. Stated another way, the best possible elimination order obtainable (however laboriously) via the unroll-and-compute scheme mentioned in §13.3.1 is still possible to obtain by finding the optimal elimination order within the modified sections.

We note that the right-interface case (shown in Figure 13.20b) is analogous. That is, in right interface case, we have $\hat{\mathcal{G}}^c$ which has its nodes and the right interface on the left section of the graph. These interfaces are again necessarily completed. Once again, however, since the interfaces are already complete (due to the FHMM's moralization), so no additional fill-in edges are added. The modified right-interface chunk can then be optimally triangulated (or some form of inference derived). Hence, the right-interface FHMM can also lead to an optimal algorithm. So like the HMM, for the FHMM, there is no difference in left- or right-interface, either in terms of memory or compute cost. In fact, this is true with for a factorial HMM with any number of hidden chains.

13.3.2.6 Cases where left and right interface are different

Can it be the case that there is a difference between the left and right interface, either in terms of memory or compute costs (or both)? If not, then clearly there is no reason to chose one over the other. In this section, we show that one can easily produce a model where the left and right interfaces are different from each other.

Our first example where the left and right interfaces are different is with a Bayesian network template, as shown in Figure 13.21. We see that the left interface size is one, while the right interface size is two. While the sizes of the two interfaces are different, that in both cases the interfaces are completed — in the left interface case there is only one vertex, and in the right interface case the two vertices are connected by an edge. Hence, in this case there is no reason to prefer one interface over the other, even though they are

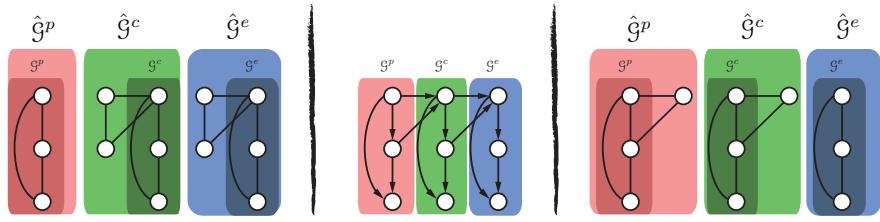


Figure 13.21: Example where the left and right interfaces are different. The template is shown in the middle. The left interface case is shown on the right. The right interface case is shown on the left. (TODO: swap to put left interface on the left of the page and the right interface method on the right of the page, unless there is some reason not to).

of different size. We also can see this by noting that the treewidth of the left- and right- interface modified templates is identical.

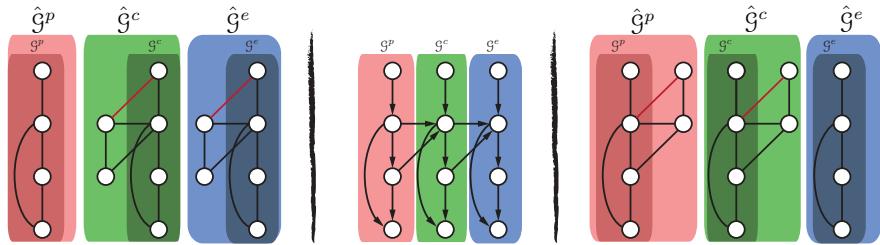


Figure 13.22: An example where the left and the right interface are the same size but where they are a different set of variables. The template is shown in the middle. The left interface case and modified sections is shown on the right. The right interface case modified sections is shown on the left.

Modifying this template a little bit by adding an additional parent, as shown in Figure 13.22, leads again to the case where the sizes of left and right interface are the same (both 2). In this case, however, the interfaces do not correspond to the same set of variables. If it was the case, for example, that the cardinality of the joint alphabet of one set of variables was different than the other set, and if the compulsory completed edges caused actual additional fill-in, then there would be reason to prefer the left vs. the right interface. However, in the present case, since there is no additional fill-in, once again the left and right interface methods have the same cost since both interfaces are already complete.

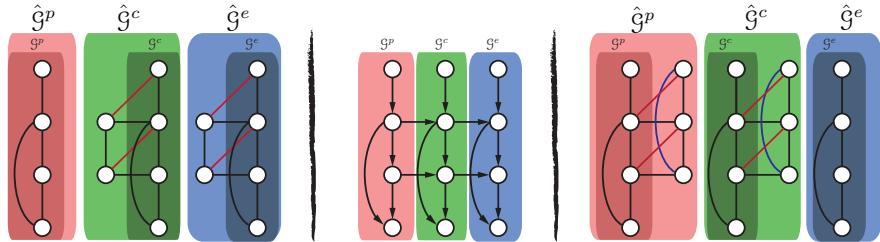


Figure 13.23: Example where the left and the right interface cases are neither identical nor have identical cost. The template is shown in the middle. The left interface case and modified sections is shown on the right. The right interface case modified sections is shown on the left. The left interface case leads to one compulsory fill-in edge, shown in blue.

The next examples shows when left and right interfaces are indeed not identical. Consider Figure 13.23. We see clearly that the size of the left and the right interface is different. However, in this case, so is the

cost. The left interface produces an interface clique of size 3, leading to a maximum maxclique of size 4. The right interface case produces an interface clique only of size 2, leading to a maximum maxclique of size 3. Hence, the left and the right interface's modified sections have different treewidths, and this is owing to the (blue) compulsory completed edge in the left-interface case. Thus, left and right interfaces can sometimes be different, and choosing the right one can mean the difference between finding the optimal inference algorithm and not finding it.

The previous case showed an example where the larger interface case was more expensive since it resulted in a compulsory fill-in edge. The next example shows that the right and left interface cases can be equally good, even when one case results in a compulsory fill-in edge and the other case does not.

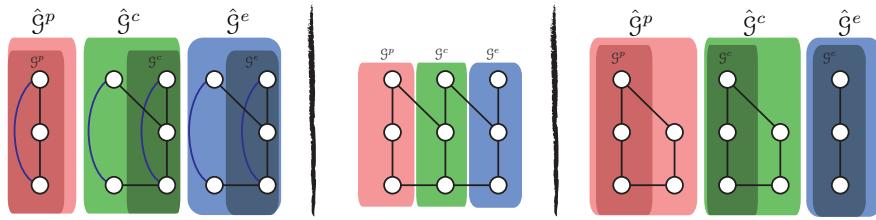


Figure 13.24: A case where the left and right interface sizes are same, where the right-interface method requires one compulsory fill-in edge (shown in blue), but where still the overall cost is the same (when the random variables have the same domains size). The template is shown in the middle. The left interface case and modified sections is shown on the right. The right interface case modified sections is shown on the left. The left interface case leads to one compulsory fill-in edge, shown in blue.

Figure 13.24 shows that the sizes of left and right interface are same (both size 2). The left interface already complete, so no edges are added. Resulting maximum maxclique size is 3. The right interface is not complete, additional edge needed. But resulting maximum maxclique size is 3. Thus, even though the right interface case needed a fill-in edge, the cost of left and right interface cases are the same. Of course, again, if the domain sizes of the interface is different, then the cost might not be the same. For example, the left-interface method might involve a clique with overall lower cardinality than the right-interface method. In this case, the real cost of the left- vs. right- interface method depends on how well one can derive an inference algorithm for the different modified sections. I.e., while the treewidth for the modified sections on both the left and the right of Figure 13.24 are the same, in practice, there can be tricks that, depending on the nature of the random variables, can lead to one being better than the other.

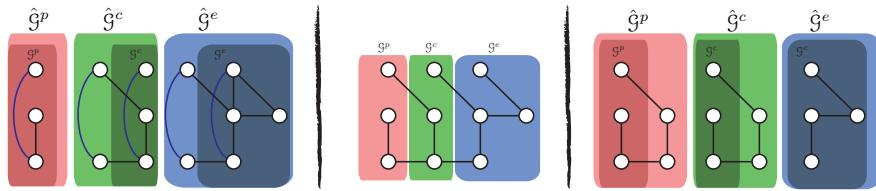


Figure 13.25: Another case where the interface sizes are the same but the cost of the left and the right interface is different.

Figure 13.25 shows yet another example. In this case, the interface sizes are the same (two for both left and right interface case). The left interface case, however, has preserved the inherent 1-tree property of the template, and thus the maximum maxclique is of size 2. The right interface case, however, has lost the 1-tree property, and it maximum maxclique size of 3, which can be significantly worse.

Yet another example showing the left- and right- interfaces, in this case for multi-frame sections and a bidirectional Bayesian network, is shown in Figure 13.26. In this case, we have edges pointing both in the

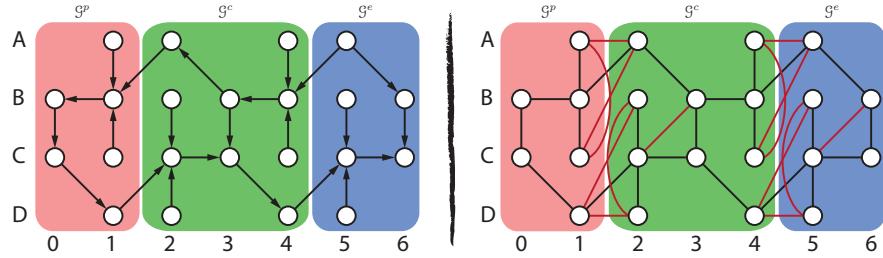


Figure 13.26: Another example. On the left we have a template where each section consists of multiple frames, two for \mathcal{G}^p , three for \mathcal{G}^c , and two for \mathcal{G}^e . Note that this is a bidirectional Bayesian network template, meaning arrows go both forward and backwards in time. This is still a valid dynamic Bayesian network template, however, since neither in the template nor in any unrolling are there any directed cycles. On the right we show the moralized template (moral fill-in edges are shown as red) which then makes the interfaces very clear. The left interface of \mathcal{G}^c consists of variables $\{A_2, B_2, C_2, D_2\}$, and where 2 is the frame number. The left interface of \mathcal{G}^e is then $\{A_5, B_5, C_5, D_5\}$. The right interface of \mathcal{G}^p is the set $\{A_1, B_1, C_1, D_1\}$ and the right interface of \mathcal{G}^c is $\{A_4, B_4, C_4, D_4\}$. Hence, the left and right interfaces are the same size in this case.

direction of forward time and backwards time.

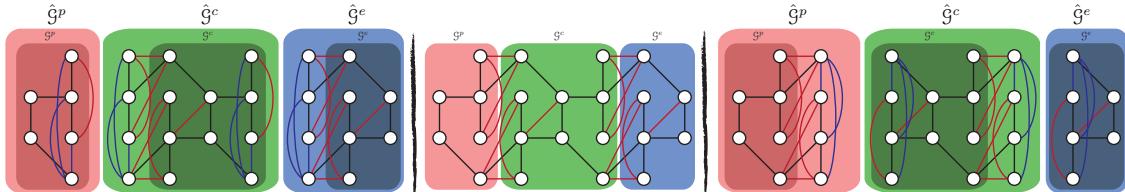


Figure 13.27: Another example. The example from Figure 13.26 showing the modified sections as well. The template is shown in the middle, in this case the moralized DBN template, where the moral fill-in edges are shown as red. The left interface case and modified sections is shown on the right. The right interface case modified sections is shown on the left. The compulsory fill-in edges for the interfaces are shown in blue.

Figure 13.27 shows the example template from Figure 13.26 in terms of the modified sections for both the left- and the right- interface. As can be seen from Figure 13.27, the interface sizes are the same (four in both case). It can be shown, however, that neither the right nor left interface is preferable. The reason for this is that the treewidth of $\hat{\mathcal{G}}^p$, $\hat{\mathcal{G}}^c$, $\hat{\mathcal{G}}^e$ is 4,4, and 3 under the left interface case, but is 3,4, and 4 under the right interface case. Hence, in this case, the left and the right interface are identical computationally.

Summarizing the different cases between deciding between the left and the right interface, we have the following cases:

- Do the left and right interfaces correspond to the same variables shifted in time? Yes (for the HMM in Figure 13.18, the factorial HMM in Figure 13.20, and Figure 13.27) and no (Figure 13.22, Figure 13.24, Figure 13.25, Figure 13.21, and Figure 13.23).
- Is the size of the left- and the right-interface the same or different? Yes (for the HMM in Figure 13.18, the factorial HMM in Figure 13.20, Figure 13.22, Figure 13.24, Figure 13.25, and Figure 13.27) and no (in Figure 13.21 and Figure 13.23).
- Do either of the left- or the right- interfaces require additional compulsory (blue) fill-in edges? No (for the HMM in Figure 13.18, the factorial HMM in Figure 13.20, Figure 13.21, Figure 13.22,

Figure 13.24 left-interface case, Figure 13.25 left-interface case, and Figure 13.23 right-interface case) and yes (Figure 13.24 right-interface case, Figure 13.25 right-interface case, Figure 13.23 left-interface case, and Figure 13.27).

- Is the resulting cost (in terms of the tree-width of the resultant modified section) the same or different between the left- vs. the right interface case? Same (for the HMM in Figure 13.18, the factorial HMM in Figure 13.20, Figure 13.21, Figure 13.22, Figure 13.24, and Figure 13.27) and different (Figure 13.25 and Figure 13.23).
- It is also worth considering cases where the set of random variables have the same size for the left and right interface case, but the cost of one is lower than the other, due to the domain size of the corresponding random variables. We can obtain examples of this using the above examples once we give the random variables domain sizes.

13.3.2.7 When neither the left nor right interface is optimal, but another “interface” is.

In §13.3.2.6, we saw that there are cases where one should prefer the left over the right interface method, or vice versa. In this section, we ask if it is always the case that at least one of the left and the right interface methods are optimal.

Firstly, what do we mean by optimal? We remind the reader that if we were to unroll the template to a given desired length, say T , and then find the optimal elimination order in that unrolled model, then this would be best for a length T model. This was called the unroll and compute method in §13.3.1. By an optimal method, we mean to say that an inference procedure derived from the template (and perhaps a modest and bounded amount of unrolling) can be used to derive an inference strategy that is just as good as if we were to find the optimal method via the (obviously impractical) unroll and compute approach.

So getting back to our question, is left or right interface always optimal, or might there be some “interface” between the left and right interface that is optimal (or at least “more” optimal)?

To further motivate this question, consider the template in Figure 13.26. From the figure (and from Figure 13.27), we see that the left interface size is 4 and the right interface size is also 4. This means that at best, the maximum clique size for each of these is at least 5 and indeed, as suggested above, it is possible to find a triangulation in each case with a clique size of no more than 5.

There is, however, an alternative separator in the graph that could be used and that is much better. The separator consists of the nodes $\{B_3, C_3\}$ and constitutes a vertex cut in the graph. The only problem however is that this separator is neither the left nor the right interface, and therefore it seems to be inaccessible.

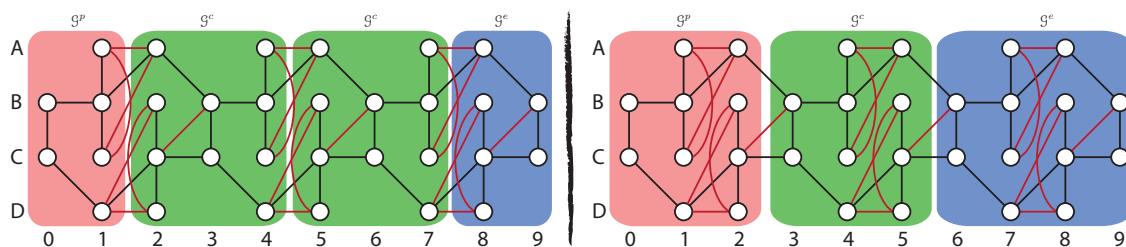


Figure 13.28: On the left, the example from Figure 13.26 unrolled one time. On the right, a redefinition of the three sections G^P , G^c , and G^e , where the chunk is delineated by interfaces that are now of size two rather than of size four. Indeed, because in this form there are no compulsory fill-in edges in either the left or right interface method relative to this modified template, this reduces the treewidth of the sections from four in Figure 13.26 to three.

If we were, however, to unroll the template one time, then we could consider the portion of the graph between these two small vertex separators as a new chunk, and this new chunk could then be repeated as necessary for any unrolling. Anything before this new chunk can be considered a new prologue, and anything after this new chunk can be considered as a new epilogue. In such case, we see that since the separator is smaller (only two vertices rather than four) and moreover since it is already complete (there is an edge between B_3 and C_3), there are no additional compulsory fill-in edges.

This is shown in Figure 13.28. On the left, we see the original template that has been unrolled once so that the graph is ten frames long consisting of a prologue, two chunks, and an epilogue. On the right of the figure, we have split the two successive chunks at the point of the minimal separator $\{B_3, C_3\}$ (and its version shifted by the length of the chunk, namely $\{B_6, C_6\}$). The right of the figure, therefore, can be seen as a new template with a bigger prologue and epilogue, and with a chunk that has a different set of interfaces. This new chunk itself has a left interface and a right interface that is quite different than the old — one should compare Figure 13.29 with Figure 13.27 to see how different the interfaces are. Since the interface is of size two, the lower bound on the maximum maxclique is three. Indeed, the tree-width of the left interface case is now three, much better than before. In fact, this recovers the best possible treewidth of the unrolled graph.

In some sense, in producing this new template we have identified a new start of the periodic section of a graph that gets repeated as mentioned in §13.3.2. In this case, however, the start of the period determines which vertices must be compulsorily completed. Searching for a start of period that requires the fewest compulsorily completed vertices could possibly lead to a strategy to ensure that one can recover the optimal possible inference strategy using the unroll and compute approach.

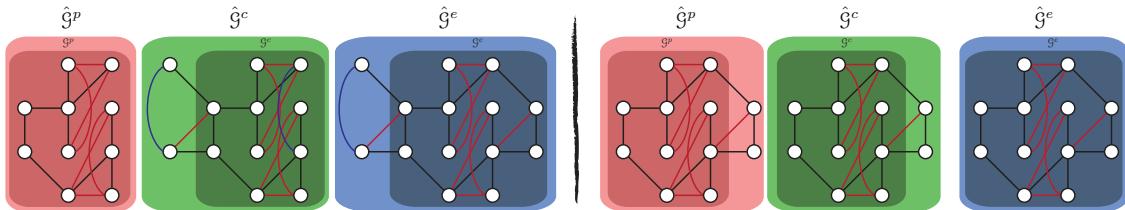


Figure 13.29: The left and the right interface modified sections (i.e., the sections as well as the interfaces) for the new template in Figure 13.28. The template is shown in Figure 13.28. The left interface case is shown on the right. The right interface case is shown on the left. We see that the right interface method results in one compulsory fill-in edge, while the left-interface method requires no compulsory full-in edges since the interface (the two nodes $\{B, C\}$) are already complete at that point.

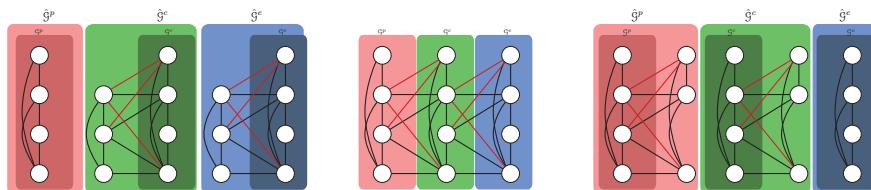


Figure 13.30: The template from Figure 13.7 shown here again in the middle. The left interface case and modified sections is shown on the right. The right interface case modified sections is shown on the left.

Lets next consider the example we saw in Figure 13.7 (which, again, is the same as that of Figure 8.79 and was shown in unrolled form in Figure 13.9). This example template and its left and right interface modified sections is shown in Figure 13.30. We see that both the left and the right interfaces are of size three, and both are already completed so no compulsory fill-in edges are needed. Indeed, the tree-width

for both of these cases is the same, namely four. From looking at §13.3.1.3, however, we know that this is not optimal as there is an elimination order that is not slice-by-slice that results in a treewidth of three. Therefore, like in the case of Figure 13.26, neither the left nor the right interface methods are optimal.

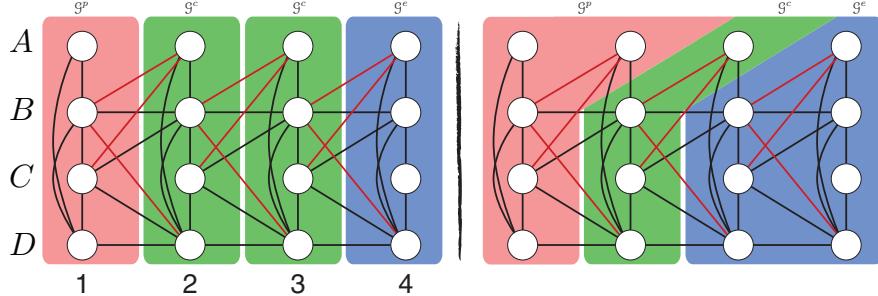


Figure 13.31: The template from Figure 13.7 unrolled by one (on the left) and then re-partitioned to have a new new prologue, chunk, and epilogue (right). This new set of sections, as seen in Figure 13.32 has an interface size better than the size four interfaces shown in Figure 13.30.

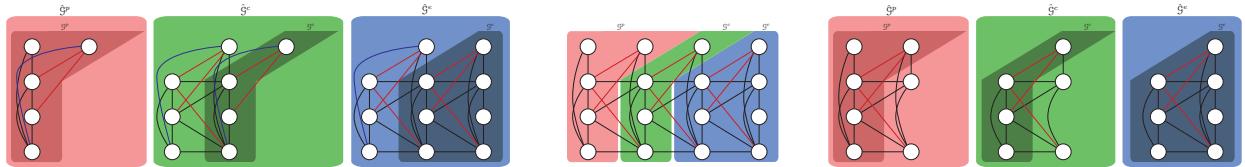


Figure 13.32: The template from Figure 13.7 shown with its new prologue, chunk, and epilogue (middle) based on the template unrolled by one as shown in Figure 13.31. On the left side of the figure, we see that the right interface has four variables and also requires one compulsory fill-in edge (shown as blue). This immediately leads to a clique size of 5, so is clearly not optimal. On the right side of the figure, however, we see that the left interface case has two variables and requires no compulsory fill-in edges. When we triangulate these sections, we can obtain a clique size of 4, giving the optimal tree-width of three as was obtained with the non-slide-by-slice elimination order as shown in §13.3.1.3.

To find the optimal interface, we once again unroll the model one time as shown on the left of Figure 13.31. Once this is done, we can define new modified sections \hat{G}^p , \hat{G}^c , and \hat{G}^e as shown in green on the right of Figure 13.31. We note that the modified sections contain variables from different time slices, unlike the original sections. For example, the modified chunk \hat{G}^c contains $\{A_3, B_2, C_2, D_2\}$.

At this point, one might wonder why this partition of the unrolled-by-one template is of any use. Indeed, we will discuss this in §?? and in §13.3.2.9. For now, let us consider Figure 13.31 as a standard template and consider both the left and the right interface case relative to Figure 13.31.

The left and right interface case, relative to the modified template in Figure 13.31, is shown in Figure 13.32. The left of the figure shows the right interface case, where we see the original sections in the darker shaded regions and the interfaces in the lighter shaded regions. Since the interfaces do not form a clique, we needed to add one compulsory edge and this is shown as blue. Hence, the treewidth of the resulting model can be no lower than four, clearly not optimal considering the non-slide-by-slice elimination order obtained a treewidth of three.

On the right of Figure 13.32, however, the interface sizes are only two and moreover the two interface variables are already connected so no additional compulsory full-in edges need be added. Optimally triangulating each of these sections results in a maximum clique size of four, hence yielding the optimal tree-width three model seen above.

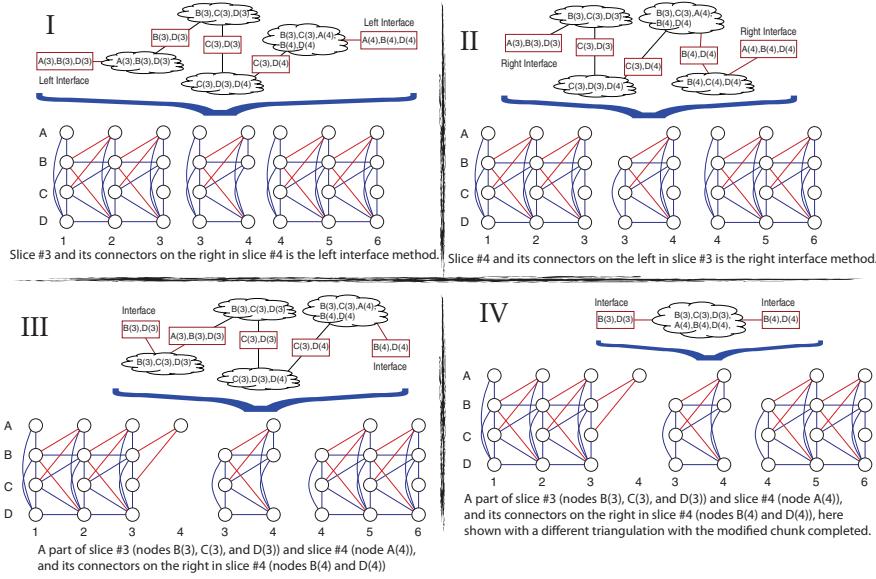


Figure 13.33: This figure shows information analogous to Figure 13.19 but for the template corresponding to Figure 13.7, Figure 8.79, and Figure 13.9. I: when elimination is performed strictly left-to-right in slice-by-sliced constrained order, there are a set of nodes A, B, and D that are forcibly completed. When we create a modified chunk (shown here as the excised slices 3 and 4), those nodes become a clique. A junction tree segment corresponding to this modified chunk is shown on the top. This junction tree can be stitched together in time to create a long junction tree corresponding to any degree of unrolling of the original DGM. II: a similar case when elimination is performed strictly right-to-left and slice-by-slice. In both case I and II, the junction tree segment has interfaces to neighboring junction tree segments of size 3, leading to a memory complexity of $O(N^3T)$ where N is the domain size of the random variables. III: an improved situation where nodes in one slice need not all be eliminated before proceeding to the next slice. An improved modified chunk (shown as the excised slices 3 and 4) can be discovered in low-order polynomial time by any minimum vertex-cut algorithm [317, 35]. Here the junction tree segment has interfaces to neighboring junction tree segments of size 2, leading to a memory complexity of only $O(N^2T)$. IV: A example of the interfaces in case III, but where the junction tree is just one clique with 6 variables, analogous to the HMM junction tree shown in Figure 8.44.

Another way to view the above property is via junction trees associated with a given modified section. Figure 13.33 continues continue the discussion of the template given in Figure 13.7, Figure 8.79, and Figure 13.9, and shows junction trees associated with unrollings of partitionings of the template.

First, the standard left interface case of the original template is shown in Figure 13.33-I, where nodes $A(t)$, $B(t)$, and $D(t)$ for all t are necessarily completed when doing a slice-by-slice left-to-right elimination ordering. In Figure 13.33-I, the size of the corresponding separator between two junction tree segments is size 3, but we can plainly see within the junction tree that if we were to define periodicity starting from a different clique within the junction tree, we could find another interface separator that would be of size 2. Thus, memory complexity could in theory be reduced from $O(N^3T)$ to $O(N^2T)$. The question is how to identify this separator.

The right interface is shown in Figure 13.33-II. We see once again, however, that the size of the separator between two successive junction tree segments is still of size 3. Moreover, if we were to try this strategy on Figure 13.10, it would still produce excessive additional edges thereby increasing the state space of the model.

Viewing the junction tree, many more options than just the standard left or right interface are available

[35], however, since there might be a small separator that lies entirely within a chunk's junction tree. As we can see from Figures 13.33-I or II, the interior (i.e., non-interface) separators of the junction trees are only of size 2, while the interface separators are of size 3. If we define a modified chunk as shown in Figure 13.33-III, then the resulting junction tree is separated by its neighboring junction tree by interfaces of only size 2, thereby achieving the desired $O(N^2T)$ memory cost. This corresponds to the right of Figure 13.32.

Figure 13.33-IV shows another example with the same interface cost (size 2) but with a very different junction tree consisting of only one clique with all variables. This is akin to the junction tree for the HMM shown in Figure 8.44 except that rather than one variable in the separator and two variables in the clique, we have 2 variables in the separator and 6 variables in the clique. It is interesting to note, in passing, that this junction tree corresponds to a “flattening” of the state space mentioned earlier, and begins to appear similar to Figure 13.19. In fact, flattening could occur for any of the interface options in Figure 13.33. In each case, all cliques are merged together into one grand clique which has the interface separators as flanking connectors to successive junction trees segments. We will revisit this example again a bit later in the article.

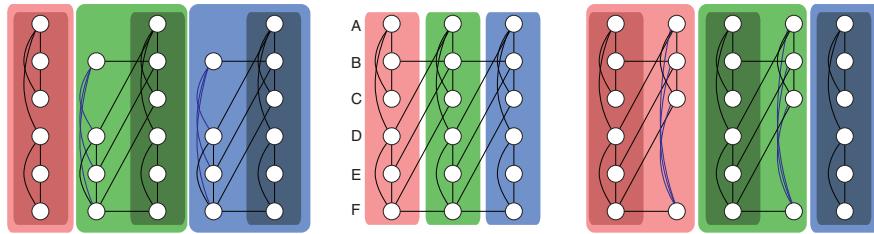


Figure 13.34: Center: the template from Figure 13.10 repeated here again. Left: the standard right interface method’s modified sections. Right: the standard left interface method’s modified sections. Neither of these are optimal.

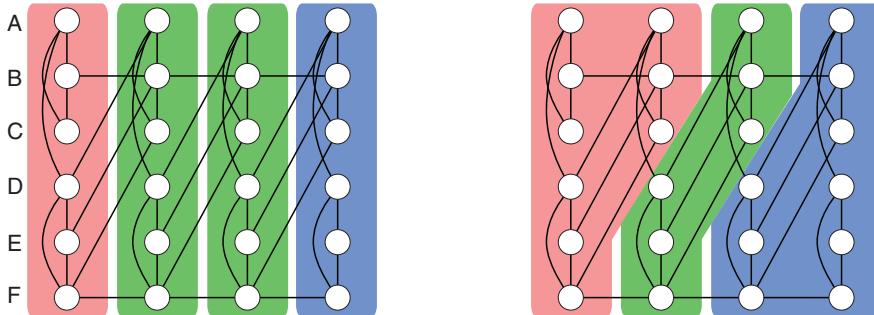


Figure 13.35: Left: the template from Figure 13.10 unrolled one time. Right: modified sections based on this unrolling.

Our last example of this section is based on the template shown in Figure 13.10 where we, once again, find that the standard left and right interface is suboptimal. Indeed, the left and right interface case are shown in Figure 13.34. We can unroll the basic template again one time as shown in Figure 13.35. This allows us to produce either a right or a left interface for this modified template as shown in Figure 13.36, the left interface case of which is optimal.

We end this section by noting the underlying process we have been performing. We unroll the basic template one time, and then identify three new sections from this unrolled template which can then be treated as the original three sections. These three new modified sections, however, might have interface separators (either left or right) which have much better properties than those interfaces associated with the original template.

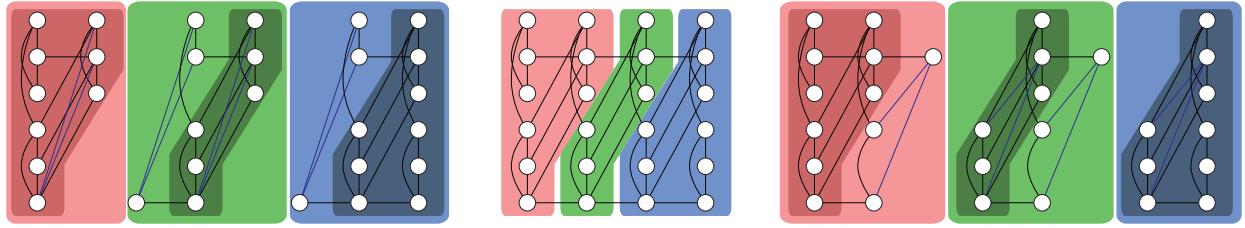


Figure 13.36: Center: the modified template from Figure 13.10. Left: the modified sections for the right interface case based on this modified template. Right: the modified sections for the left interface case based on this modified template. This latter left interface case achieves the optimal treewidth.

13.3.2.8 Where no default interface is optimal

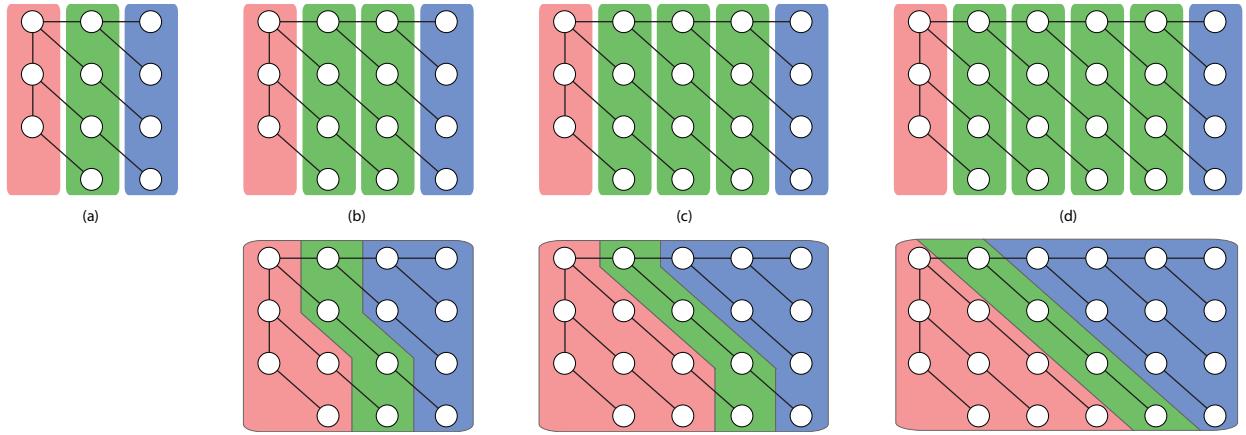


Figure 13.37: The example of Figure 13.12 where we unroll by more than one before finding a new template partition. In (a) we show the basic template again in undirected form. In (b) we unroll the template one time ((b)-above), which is not enough to recover the one-tree property of the template since the compulsory fill-in edges increase the tree-width. One example of a partition in this case is shown in (b)-below. In (c) we unroll the template two times (above), which is still insufficient to retain the one-tree property (example shown below). Finally, in (d) we unroll the template three times (above), which now allows us to form a partition having interface separators of size one, meaning no compulsory edges need be added and thus retaining the one-tree property of the template. This the new partition is formed below. Note that the modified chunk \hat{G}^c now spans four of the frames of the original template. This, therefore, puts a new lower bound on the length of the possible segments that can be represented by this template. Let $\hat{\tau}$ correspond to a segment length. In the original template, we can have segment lengths satisfying $\hat{\tau} = 2 + k$ for integral $k \geq 1$ (and $k \geq 0$ if we allow only a prologue and epilogue). With the new segment in (d), $\hat{\tau}$ must satisfy $\hat{\tau} = 5 + k$.

Let us next consider the template given in Figure 13.12. This template, as you may recall, corresponds to a one-tree and hence has an exact inference algorithm that is efficient for any degree of unrolling. If we perform the strategy suggested in §13.3.2.7 where we first unroll one time, as shown in Figure 13.37-(b)-above, then a simple enumeration shows that there is no interface separator, either left or right, that achieves the tree-width one. In fact, unrolling one time would result in a minimum interface size of three, potentially two orders of magnitude worse than the original one-tree. One example of this is shown in Figure 13.37-(b)-below.

Rather than giving up, let's continue this process a little bit by unrolling once again as shown in Figure 13.37-(c)-above. Unfortunately, here again we see that by simple enumeration that the best interface is

of size two, potentially an order of magnitude worse than the one-tree. An example of a re-partition is shown in Figure 13.37-(c)-below, but any such partition would not retain the one-tree property of the template.

Unrolling the template one additional time, as shown in Figure 13.37-(d)-above finally leads us to a separator that has as interfaces of size one. This can then be used to define a new template as shown in Figure 13.37-(d)-below. Although this template perhaps looks a bit strange, it is well-defined nonetheless and corresponds to a valid template.

The question we must address now is how to we automatically identify how to identify the “optimal” separator, how much to unroll in order to identify it, and moreover, how to determine even what “optimal” really means (how to judge a separator)?

13.3.2.9 The M (vertex cut span) and S (chunk skip) parameters, and template restrictions

In all of the above example sans Figure 13.12, we have unrolled the template one time, and then re-partitioned the unrolled template into three sections, the second section of which became the new “modified” chunk \hat{G}^c , and the first and third sections became the modified prologue \hat{G}^p and epilogue \hat{G}^e respectively. We have also magically come up, in each case, with a new partition of the unrolled template that has ended up enabling the optimal triangulation to occur, and in Figure 13.37 the number of times needed to unroll the template before a re-partitioning of the unrolled template retains the low tree-width property of the template. In this section, we talk about how this can be done more formally and automatically.

In order to do this, we need to define both a graph cut and a vertex cut. Given a graph $G = (V, E)$ a graph cut (or just a cut) is defined as a subset of edges $C \subseteq E$ that separates one set of nodes $A \subset V$ and another $B = V \setminus A$. We can define a cut via a set of vertices, in that given $A \subset V$, this defines a cut via the set of edges $\delta(A) = \{(u, v) = e \in E : u \in A, v \in V \setminus A\}$ incident to the nodes A but not fully contained by A . Given particular vertices $s, t \in V$, an (s, t) -cut is a set of edges that separate s and t .

A vertex cut [317], as the name implies, is a cut that is created by a set of vertices of a graph. That is, given a graph $G = (V, E)$, a subset of vertices $C \subseteq V$ is a vertex cut if there is a set of vertices $A, B \subset V$ such that every path from any vertex in A to any vertex in B must go through a vertex in C . An (s, t) vertex cut is a vertex cut that separates vertices s and t .

As an example of edge and vertex cuts, Figure 13.42 shows an undirected graph where there are many cuts. For example, edges $\{e_1, e_2\}$, $\{e_4, e_3, e_5, e_6\}$ and $\{e_6, e_7, e_8\}$ are (s, t) edge cuts, and $\{a, b\}$, $\{a, b, c\}$, and $\{b, c\}$ are (s, t) vertex cuts. Figure 13.38b shows a directed graph — here, edge sets $\{e_1^\rightarrow, e_2^\rightarrow\}$, $\{e_a^\rightarrow, e_b^\rightarrow\}$, and $\{e_2^\rightarrow, e_3^\rightarrow, e_5^\rightarrow, e_7^\rightarrow, e_8^\rightarrow\}$ are all directed (s, t) cuts. Note that in the directed case, the arrow directions are significant, a cut, in general, means that there is no directed path from s to t that is left uncut, but there still can be uncut paths from t back to s . Note that a cut is called minimal if any subset of a cut is no longer a cut. See [386] for much more discussion on this topic (and others).

We immediately see the correspondence between a separator in a graphical model with a vertex cut in the corresponding undirected graph — indeed, all separators in a undirected graphical models are vertex cuts in the corresponding undirected graph (although in a Bayesian network, a directed separator obtained via d-separation does not correspond to a directed cut, but here directed versions of the graph that we will construct below have nothing to do with Bayesian networks).

It turns out that the problem of finding a separator in a DGM template is identical to finding a vertex cut in a transformation of the partially unrolled template. Recall the notation in §8.11.2 and §8.11.3, and in particular Equations (??) and (??). Suppose we start with template $G = (V, E)$ consisting of timed nodes and edges, and we unroll this graph twice (so $\hat{\tau} = 2$) giving template G_2 and vertex sections V^p , V^c , $V_{\Delta(T(\mathcal{G}^c))}^c$, $V_{\Delta(2T(\mathcal{G}^c))}^c$, and $V_{\Delta(2T(\mathcal{G}^c))}^e$. We can construct a new graph with vertex set $V_{\Delta(T(\mathcal{G}^c))}^c \cup \{s, t\}$ where s, t are new vertices not contained within $V_{\Delta(T(\mathcal{G}^c))}^c$. The edges are constructed to be the same as the vertex induced graph $G_2[V_{\Delta(T(\mathcal{G}^c))}^c]$ except that every vertex in $V_{\Delta(T(\mathcal{G}^c))}^c$ that has a neighbor in $V^p \cup V^c$ (i.e., $V_{\Delta(T(\mathcal{G}^c))}^c$ ’s left interface), is now instead connected to s , and every vertex in $V_{\Delta(T(\mathcal{G}^c))}^c$ that has a neighbor

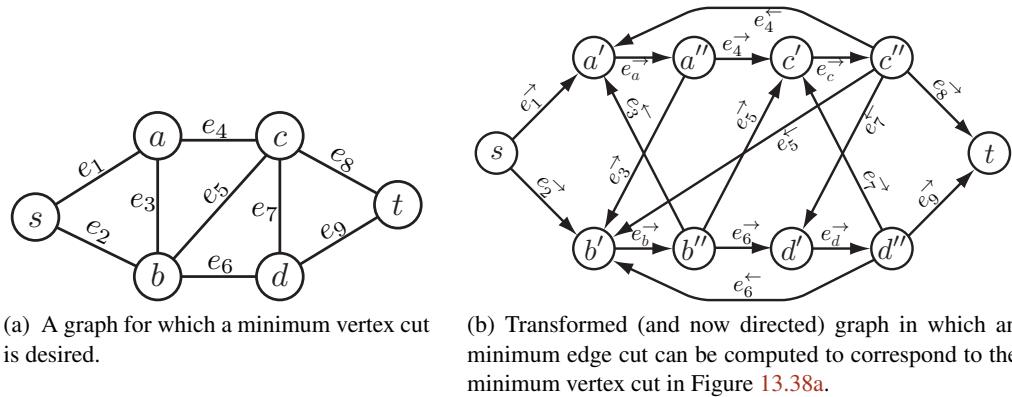


Figure 13.38: A vertex cut can be computed using standard edge cut in a transformed graph as shown in this example. This figure is discussed further in §13.3.2.10.

in $V_{\Delta(2T(\mathcal{G}^c))}^c \cup V_{\Delta(2T(\mathcal{G}^c))}^e$ (i.e., $V_{\Delta(T(\mathcal{G}^c))}^c$'s right interface) is now instead connected to t . Note the reason for unrolling the graph thrice is to ensure that vertices s and t are connected (recall that a template might have either an empty prologue or epilogue).

We then find an (s, t) optimal vertex cut (how to judge the quality of a given vertex cut is discussed fully below) in this graph. Lets say that $C^c \subseteq V_{\Delta(T(\mathcal{G}^c))}^c$ is this vertex cut.

This vertex cut becomes the separator to split the unrolled template into the modified prologue, chunk, and epilogue as we've done above. That is, we take the template and this time we unroll it once giving template G_1 and sections with vertex sets V^p , V^c , $V_{\Delta(T(\mathcal{G}^c))}^c$, and $V_{\Delta(T(\mathcal{G}^c))}^e$. We have that $C^c \subseteq V_{\Delta(T(\mathcal{G}^c))}^c$ and also that $C_{-\Delta(T(\mathcal{G}^c))}^c \subseteq V^c$. Moreover, both $C^c \subseteq V_{\Delta(T(\mathcal{G}^c))}^c$ and $C_{-\Delta(T(\mathcal{G}^c))}^c$ constitute vertex cuts in G_1 and we can use these vertex cuts, along with the portion of G_1 that lies between these two cuts, to define the modified template sections. That is, we use the block of vertices between these two vertex cuts to define the modified chunk $\hat{\mathcal{G}}^c$. Note that there are three non-intersecting sets of vertices, the left vertex cut C_l^c , the right vertex cut C_r^c , and the intervening vertices V^i . With these, we have several choices: If we include within the modified chunk the left vertex cut, namely take $\hat{\mathcal{G}}^c \leftarrow G_2[C_l^c \cup V^i]$, we obtain a section that uses the vertex cut as its left interface, and if the vertex cut is optimal, then so is the corresponding modified section's left interface. If, on the other hand, we include within the modified chunk the right vertex cut, namely take $\hat{\mathcal{G}}^c \leftarrow G_2[V^i \cup C_r^c]$, we obtain a section that uses the vertex cut as its right interface, and if the vertex cut is optimal, then so is the corresponding modified section's right interface.

We next give several examples of this, and in doing so describe an important variant of the above procedure.

First, consider the template initially given in Figure 8.79. For this template, we saw that neither the standard left nor right interface was optimal. If we unrolled the template one time, it was possible to find a partition of the vertices such that the left interface relative to that partition was optimal, as was shown in Figure 13.31 and Figure 13.32.

We can form an (s, t) graph from this template as shown in Figure 13.39 on the left. Note that s is connected to the left interface of the chunk, and t is connected to the right interface of the chunk. Suppose that we judge the quality of a vertex cut by its size (i.e., we ask for the vertex cut consisting of the fewest number of vertices). In such case, the vertex cut is size two as shown either as shaded yellow Figure 13.39-(a).

Once we have the optimal vertex cut, we need to form the modified prologue, chunk, and epilogue based on these partitions. If the best vertex cut consists of the entire chunk, then the standard left and right interface is the only option and we need not proceed any further. If, on the other hand, the best vertex cut consists of

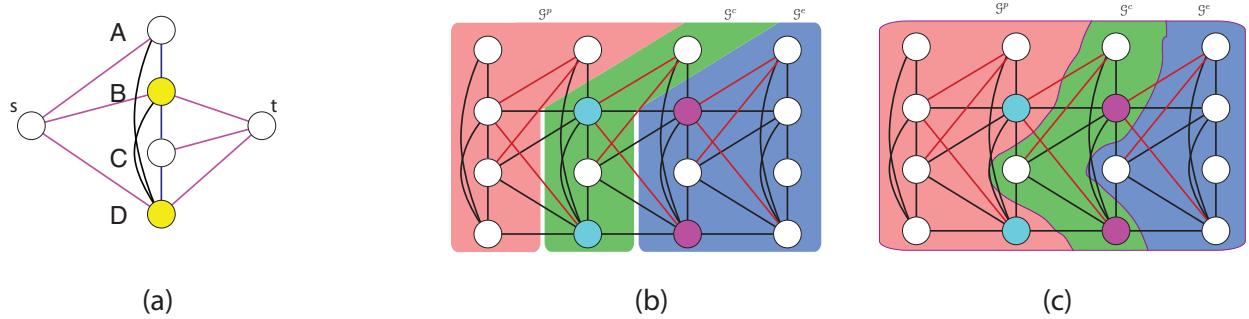


Figure 13.39: (a): An (s, t) graph for the template given in Figure 8.79, unrolled in Figure 13.7, and re-partitioned in Figure 13.32. When the vertex cuts are measured by size, the best (minimum) cuts is shown shaded yellow and has size two. (b): The template of Figure 13.7 unrolled one time ($\hat{\tau} = 1$) and thus containing two of the vertex cuts given in (a), a left one (shaded cyan) and a right one (shaded magenta). In this case, we re-partition the template so that the modified chunk includes the vertices between the two vertex cuts (shown as white with a green background) and also the left vertex cut (cyan). This re-partitioning is identical to the one given in Figure 13.31 and is optimal in the left interface case — recall the left interface relative to this modified template is optimal (in that it leads to no additional fill in and can recover the original tree-width of the template). (c) An $\hat{\tau} = 1$ unrolling of the template and re-partitioning where the modified chunk includes vertices between the two vertex cuts (white with a green background) and also the right vertex cut (magenta). This modified template is optimal in the right interface case.

a subset of \mathcal{G}^c , then we can form a new chunk to be those nodes that lie between two successive vertex cuts, as mentioned above. To do, one must at least unroll the graph one time, which means that there exists two instances of the vertex cut. This is done in Figure 13.39-(b), and the left vertex cut is shown as cyan and the right vertex cut is shown as magenta.

Given this unrolled graph, we still have two choices regarding how to produce modified sections. The choice is determined by which of the vertex cuts are contained within $\hat{\mathcal{G}}^c$ and which act as the interface to a neighboring modified chunk. Indeed, the vertex cut is the interface through which $\hat{\mathcal{G}}^c$ communicates, and moreover renders all variables to the left (into the past) and right (into the future) of the cut independent. If the left vertex cut is contained within $\hat{\mathcal{G}}^c$, then this leads to it being optimal w.r.t. the left interface method relative to the modified template, as shown in Figure 13.39-(b) (we also saw this in Figure 13.31 and Figure 13.32). Recall, however, from Figure 13.32 that the right interface method, for this template, was not optimal. If, on the other hand, the right vertex cut is contained within $\hat{\mathcal{G}}^c$, this leads to it being optimal w.r.t. the right interface method relative to the modified template, as shown in Figure 13.39-(c). The left interface method would not be optimal for the modified template in Figure 13.39-(c) since it would select vertices some of which are not directly connected. Note that in each case, by optimal we mean in terms of tree-width (since no compulsory fill-in edges are required).

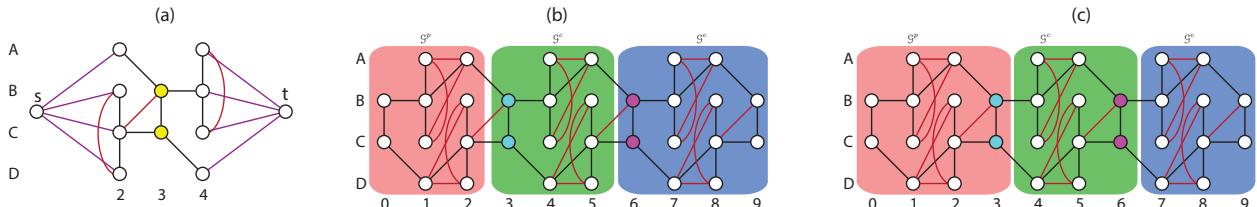


Figure 13.40: (a): An (s, t) -graph for the example Figure 13.28. (b): a modified template for which the left interface is optimal. (c): a modified template for which the right interface is optimal.

For our second example, consider Figure 13.28 which shows both a template unrolled one time and an optimally partitioned version of this unrolled template. Figure 13.40-(a) shows the template chunk with additional vertices s and t constructed as mentioned above. In particular, once again, s is connected to the chunk's left and t is connected to the chunk's right interface. The optimal vertex cut (when size is the objective) is shown as yellow.

Once we unroll the template one time, we have two copies of the vertex cut, a left (cyan) and a right (magenta) copy. As above, there are two choices for partitioning these vertices. Using the left minimum cut (cyan) within $\hat{\mathcal{G}}^c$ leads to the left-interface method being optimal for the template in Figure 13.28 (also shown in Figure 13.40-(b)). Using the right minimum cut (magenta) within $\hat{\mathcal{G}}^c$ leads to the right-interface method being optimal for the template in Figure 13.40-(c).

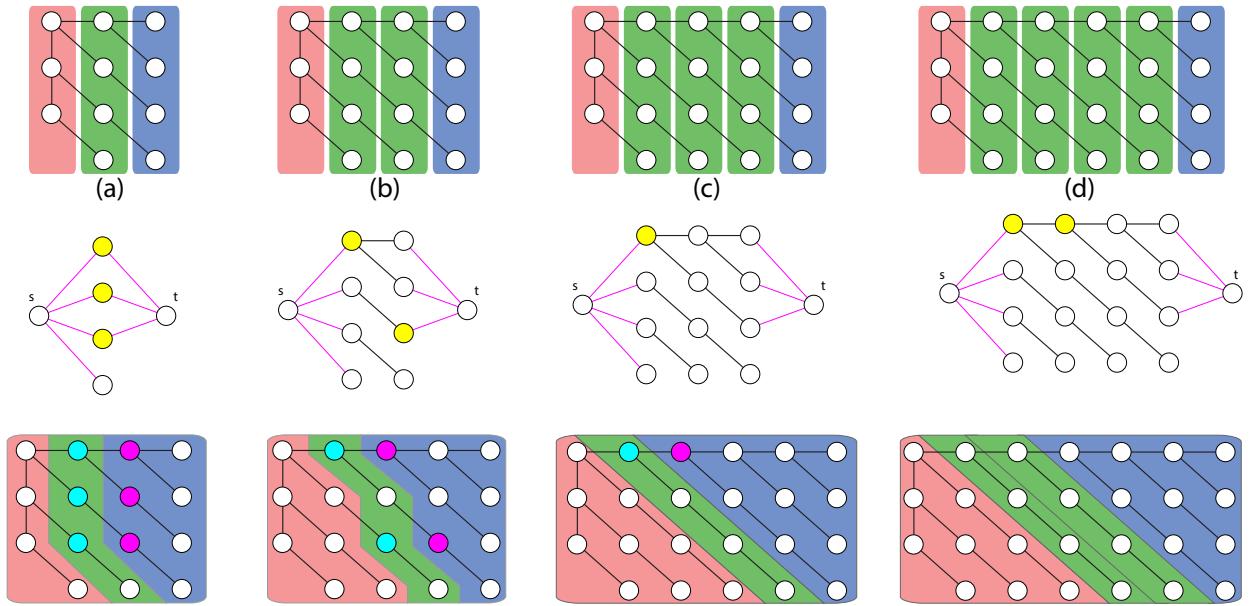


Figure 13.41: The example of Figure 13.12 and Figure 13.37. In each case ((a), (b), (c), and (d)), we have the unrolled template (top), the (s, t) -graph constructed from the chunks within the template (middle) and the resulting vertex cut chosen (in yellow), and a resulting modified template based on the chosen vertex cut (bottom) based on unrolling the template once more relative to the template, and where the corresponding two vertex cuts are in cyan or magenta. In (a), we have the original template which leads to a size three vertex cut. In (b) we have a size two vertex cut. In (c) we have a size one vertex cut which achieves the optimal tree-width one of the graph. In (d) there is no benefit over (c) and in fact we have either of the yellow vertices being an optimal size one vertex cut.

Our third example is for the template in Figure 13.12 which was shown for various unrollings in Figure 13.37, and is given in Figure 13.41. As explained earlier, the template corresponds to a tree under any unrolling so any modified template that does not achieve a tree-width of one is suboptimal. Indeed, when looking at the minimum vertex cut (again judged by size) as we've so far done (and as shown in Figure 13.41-(a) top), the minimum vertex cut is three, leading to modified templates as shown in the bottom which clearly, either for the left or the right interface method, is suboptimal. We can search for a vertex cut in a graph consisting of more than one chunk, as shown on the top in Figure 13.41-(b), but this also results in a vertex cut of size two, also suboptimal as seen on the bottom. It is only when we search for a vertex cut in three successive chunks do we find the optimal size-one vertex cut (shown in yellow in Figure 13.41-(c)). This leads to a very diagonal looking modified chunk shown at the bottom. Moreover, if we attempt to find a vertex cut in more than three chunks, as shown in Figure 13.41-(d), there is no further benefit (in fact, there

are two optimal vertex cuts in this case, both shown as shaded yellow). Hence, searching for a vertex cut in three successive chunks is sufficient for partiality.

In light of the above examples, we describe the procedure for vertex cut finding in Algorithm 16. Note that in the algorithm we introduce two new inputs: first, is M , the vertex cut span and corresponds to the maximum number of chunks that a vertex cut may lie in. We also a bit more formally introduce $J_c(C)$, an objective to judge the quality of a given candidate vertex cut C .

TODO: Relate M to the aforementioned k for periodic signals in Equation (13.2).

Algorithm 16: Vertex cut algorithm for finding an optimal interface separator

Input: DGM template (\mathcal{G}^p , \mathcal{G}^c , and \mathcal{G}^e), M (the vertex cut span) and $J_c(\cdot)$ (an objective to score vertex cuts).

Result: Resulting vertex cut C and modified template sections ($\hat{\mathcal{G}}^p$, $\hat{\mathcal{G}}^c$, and $\hat{\mathcal{G}}^e$)

1 Create a graph $G = (V, E)$ consisting of M copies of \mathcal{G}^c where

$V = \{s, t\} \cup V^c \cup V_{\Delta(T(\mathcal{G}^c))}^c \cup \dots \cup V_{\Delta((M-1)T(\mathcal{G}^c))}^c$, where the edges among the copies of V^c are as in any unrolling, and where s is connected to the left interface of V^c and where t is connected to the right interface of $V_{\Delta((M-1)T(\mathcal{G}^c))}^c$;

2 Find an (s, t) vertex cut in this graph that minimizes $J_c(C)$;

3 Unroll the original template $M + 1$ times (leading to two copies of the vertex cut, C and $C_{\Delta(\mathcal{G}^c)}$) ;

4 Form $\hat{\mathcal{G}}^p$ to include \mathcal{G}^p and anything leading from any vertex in \mathcal{G}^p up to C ;

5 Form $\hat{\mathcal{G}}^e$ to include \mathcal{G}^e and anything leading from any vertex in $\hat{\mathcal{G}}^e$ back to $C_{\Delta(\mathcal{G}^c)}$;

6 Form $\hat{\mathcal{G}}^c$ to include the vertices not included in either $\hat{\mathcal{G}}^p$ nor $\hat{\mathcal{G}}^e$;

7 Note for a left interface method, C is included in $\hat{\mathcal{G}}^c$ and $C_{\Delta(\mathcal{G}^c)}$ is included in $\hat{\mathcal{G}}^e$ — while for a right interface method, C is included in $\hat{\mathcal{G}}^p$ and $C_{\Delta(\mathcal{G}^c)}$ is included in $\hat{\mathcal{G}}^c$;

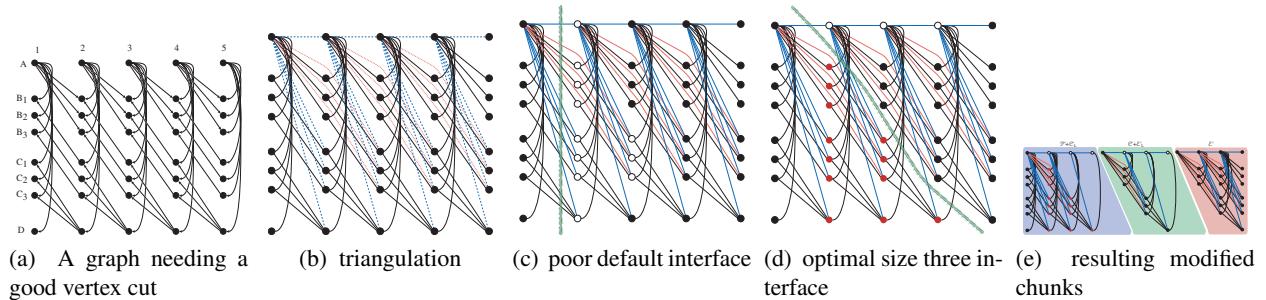


Figure 13.42: Another example of a perhaps surprising vertex cut.

There are several issues raised by the above that should to be discussed further. These are:

1. $J_c(\cdot)$, i.e., how to judge the vertex cut, by size, by weight, fill-in, or something else? Indeed, the separator can be judged in a number of ways, including:
 - (a) the size of the separator
 - (b) the weighted sum of the variables in a separator (log state space)
 - (c) The edge fill-in of the candidate separator
 - (d) the size of the maximum maxclique in the resulting new chunk $\hat{\mathcal{G}}^c$
 - (e) the entire state space of the resulting chunk $\hat{\mathcal{G}}^c$
 - (f) the ability to make approximations within modified chunk (e.g., a variational bound)

- (g) closeness to a particular desired query that is needed (i.e., might be other constraints, need a joint over certain variables not involved in a factor).
2. How to compute the optimal vertex cut for a given vertex cut objective function and vertex cut span? What are the computational implications of these choices?
 3. How many successive chunks might the optimal vertex cut lie in (how to choose M)?

We address these in the next section.

13.3.2.10 Computing the optimal vertex cut

It turns out that the different choices can make a big difference to the computability of the optimal vertex cut.

We first discuss J_c . In many cases, it is possible to reduce the optimal vertex cut problem to a standard minimum edge cut problem, but it depends on how we judge a given cut. Let C be a set of vertices, and let $J_c(C)$ be the cost of the cut. Suppose $J(C) = \sum_{c \in C} J(c)$ where $J(c)$ is the cost of an individual vertex. We note that this form of function is modular. I.e., whenever $J(C) = \sum_{a \in A} w(a)$ with C is a candidate separator, we have a modular weight function $w : V \rightarrow \mathbb{R}_+$ that gives a weight for each vertex.

A modular cost vertex cut includes the case where we judge the cut only by its size, namely $J(C) = |C|$ since we can take $J(c) = 1$ for all c . One instance of this includes measuring the state space of the set of random variables within the vertex cut. Each $c \in C$ corresponds to a random variable and each random variable X_c has a domain or alphabet size $|D_{X_c}|$. Since the interface will be compulsorily completed one useful measure of interface quality is this state space, i.e.,

$$J_c(C) = \log \prod_{c \in C} |D_{X_c}| = \sum_{c \in C} \log |D_{X_c}| \quad (13.10)$$

which has the sum form above. Assuming that C is this minimum vertex cut, then the memory cost of exact inference on a sequence of T chunks becomes $O(T \prod_{v \in C} |D_{X_v}|)$ (as discussed in §13.3.2.4). Moreover, if a modified chunk that lies between two successive vertex cuts, say C_t and C_{t+1} is now $\hat{\mathcal{G}}_t^c$ then the time complexity is at most $O(T \prod_{v \in \hat{\mathcal{G}}_t^c} |D_{X_v}|)$. Therefore, one goal might be to find a vertex cut (interface separator) that minimizes $\prod_{v \in S} |D_{X_v}|$ (to reduce memory requirements) and $\prod_{v \in \hat{\mathcal{G}}_t^c} |D_{X_v}|$ (to reduce an upper bound on computational demands).

It turns out, whenever $J(C) = \sum_{c \in C} J(c)$ (any non-negative modular weight function), it is possible to reduce the vertex cut problem into a standard edge cut problem and use a standard (s, t) maxflow/mincut algorithm (which is quite efficient to find, as described in [386]).

If the separator cost is measured simply in terms of the number of nodes in the separator, or is measured in terms of the weight of the separator (i.e., log state space or $\log \prod_{v \in S} |D_{X_v}|$), then the problem can be solved optimally in polynomial time using a max-flow algorithm. The approach is to transform the graph $(\mathcal{G}_1^c, \mathcal{G}_2^c, \dots, \mathcal{G}_M^c)$ into a flow network by adding two nodes (s, t) and where s is connected to all nodes in I_l , and t is connected to all nodes in I_r . All edges in the graph are set to have infinite capacity and then we solve the max-flow problem in a transformed network where the nodes may have a cost. This is done by duplicating each node and connecting them with an edge corresponding to the cost of the node [259], a standard procedure for performing max-flow with node costs, also known as the vertex cut problem [317]. None of the resulting edges from the original graphical model, therefore, will be selected as part of the cut since they have infinite capacity, and the cut will consist only of node edges, corresponding to the separator with the minimal cost that we desire.

To solve a minimum (modular cost) vertex cut with a standard minimum edge cut procedure, we do the following: We first create new vertices s, t , and connect s to all nodes in the left interface, and connect t to

all nodes in the right interface, as described in the first part of Algorithm 16. An example of this is shown in Figure ???. Next, we transform the vertex cut graph to an edge cut graph as shown in Figure ???, and find directed edge-cut on the transformed graph. The graph transformation is as follows. Each original vertex is transformed into two vertices connected by an edge having the capacity equal to the cost of the vertex. Also, each original graph edge is transformed into two directed edge, each having infinite capacity. We then find a minimum edge cut in this new graph, and the resulting edge cut is guaranteed to correspond to a minimum vertex cut in the original graph.

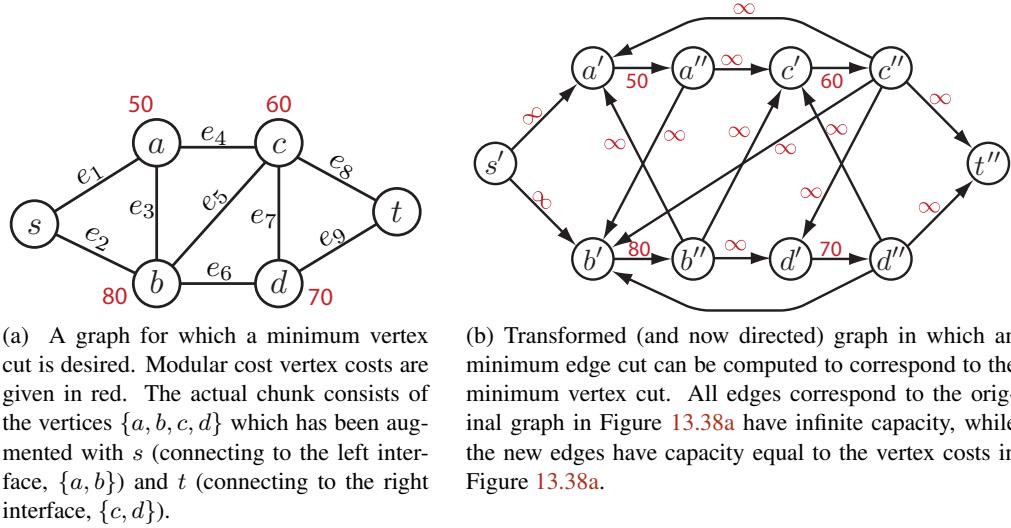


Figure 13.43: A vertex cut with a modular vertex cut cost function can be computed using standard edge cut in a transformed graph as shown in this example.

More formally, given graph $G = (V, E)$, we form a new digraph $G' = (\{s', t''\} \cup V' \cup V'', E_s \cup E_t \cup E' \cup E'')$, where $V' = \{v'\}$ for each $v \in V(G)$ and $V'' = \{v''\}$ for each $v \in V(G)$. Thus, there are two more than twice as many nodes in G' as there are in G . We view each $v' \in V'$ as an *in-node* and each $v'' \in V''$ as an *out-node* which gives a direction between these vertices — namely, we connect each in-node with its corresponding out-node producing edge set $E'' = \{(v', v'')\}$ for each $v \in V(G)\}$. Next, for any connected original graph vertices, we connect their in to out (and out to in) vertices. That is, we create additional directed edges $E' = \{(u'', v')\}$ for each $(u, v) \in E(G)\}$, so if exists edge (u, v) in G , then u 's out-node gets connected to v 's in-node, and v 's out-node gets connected to u 's in-node. We set $E_s = \{(s', v') : v \text{ is a left interface vertex}\}$. Lastly, we set $E_t = \{(v'', t'') : v \text{ is a right interface vertex}\}$. Each such edge $e \in E''$ is given capacity equal to the cost of the corresponding vertex v , and all other edges in G' are given infinite capacity. An example of this is shown in Figure 13.43.

It should be clear that any vertex cut in G corresponds to a directed edge cut in G' , meaning that there is no directed path from s to t that doesn't go through the edges corresponding to the vertex cut in G . Also, note, to prevent any min-cut from including edges in E' , we set the weight of each edge in E' to a large number (e.g., ∞ , but n^2 is sufficient in practice). Finding an (s', t'') cut in G' will find minimum (s, t) vertex-cut (and therefore graph separator) in the original graph G . The algorithm to compute minimum vertex separator $\kappa(G)$ in the graph runs in polynomial time. For example, one algorithm for doing so [164] can compute the cut in time $O(m(n + \min(\kappa(G)^{5/2}, \kappa(G)n^{3/4})))$.

There are a number of possible vertex cut cost objectives that we might wish to use. What happens, however, if this cost function is not modular? In such case, it is no longer possible to reduce the problem to a standard single edge-cut algorithm and get the optimal solution. For example, given candidate vertex

cut C , and the corresponding set of random variables, we might wish to use the entropy of these random variables as their cost:

$$J_c(C) = H(X_C) = - \sum_{x_C} p(x_C) \log p(x_C). \quad (13.11)$$

The entropy prefers cuts whose state space can be highly compressed. An alternative formulation comes when the vertices might have deterministic relationships with each other. Suppose, for example, that for each $v \in V$, there are a set of other vertices π_v that render the random variable X_v deterministic. That is, given the values of X_{π_v} , the value of X_v becomes known, which means that in such case X_v does not contribute to the overall state space of the vertices. This leads to the cost function

$$J_c(C) = \sum_{v \in C \setminus \{u : \pi_u \subseteq C\}} \log |\mathcal{D}_{X_v}| \quad (13.12)$$

which does not count the state space of any variable that can be deterministically inferred by the other variables in the cut.

It turns out that the two functions mentioned above (Equations (13.11) and (13.12)) are submodular, and using them to judge the quality of a cut corresponds to the “cooperative cut” problem (a problem first introduced in [35], and then expanded upon in [204]). In [204], moreover, a simple iterative algorithm (that repeatedly computes upper bounds on the submodular function and then solves then using standard graph cut) was found to be very effective for this problem, not to mention having certain theoretical worst-case guarantees on its performance.

Other cost functions, however, are not submodular. For example, suppose we wished to find the vertex cut that leads to the least number of additional fill-in edges. I.e., given a graph $G = (V, E)$ the cost of a cut $C \subseteq V$ would be:

$$J_c(C) = \sum_{i,j \in C} \mathbf{1}_{(i,j) \notin E} \quad (13.13)$$

I.e., we sum up all pairs of vertices within C that are not already an edge. This function is non-negative and supermodular, and was also mentioned in [35], and could be called an instance of a “competitive cut” problem (since edges rather than reducing each others cost and thus cooperate, they increase each others costs and so compete). Thus, an approach very similar to [204] can be used to produce an approximate solution.

If the function $J_c(C)$ is neither submodular nor supermodular, if it is computationally feasible to decompose the function into $J_c(C) = f(C) - g(C)$ where f and g are both submodular (which it often is when the internal structure of the function $J_c(C)$ is known), then it can be possible to run an algorithm to the submodular-supermodular procedure [319, 203] but in this case make a restriction that the solutions must be cuts. While these problems are in general inapproximable, it may be that heuristics work quite well for certain J_c .

We note also that the vertex cut analysis needs to be done only once, and can be used many times for the life of the model. Hence, an offline algorithm can even use a search algorithm to find the optimal cut. That is, it is often feasible even to try all vertex cuts to find optimal one, as was also done in [35].

We note that so far, all of the above objectives have been local in that $J_c(C)$ determines the quality of a cut based only on the vertices C and the subgraph $G[C]$. E.g., considering the modular case (size or weight), entropy, or the necessary fill-in, in each case, the quality measure is local or *intrinsic*, meaning one never looks outside the vertex cut itself to judge its quality. In fact, we may distinguish between different J_c 's based on if they are intrinsic or if instead they are *extrinsic*, meaning that to evaluate $J_c(C)$ one must look beyond either C or $G[C]$.

We discuss extrinsic J_c 's below. In the intrinsic J_c case, interestingly, the quality the best left and best right interface will be identical, as given in the following theorem.

Theorem 153. Left & Right Interface Parity

Suppose we can find the optimal vertex cut based on an intrinsic $J_c()$ function. Then we can form modified sections using this vertex cut either as a left interface or as a right interface, and hence the intrinsic cost of the best left and right interface method is identical.

Proof. Consider a modified partition of the unrolled template using the optimal vertex cut based on the left interface method. We can move the left interface nodes (and their shifted by $\Delta(T(\mathcal{G}^c))$ cohorts) to the other side of the boundary between sections, and we get a partitioning that uses the same vertex cut as a right interface. \square

Note, that the proof of this theorem was used directly in Figure 13.39-(b)-(c) and Figure 13.40-(b)-(c). Hence, this means that when one finds an optimal vertex cut, the left/right interface distinction discussed in §13.3.2.2 and §13.3.2.3 becomes moot.

There are non-intrinsic measures of vertex cut costs not yet mentioned. Indeed, a number of *extrinsic* (or global) cost measures $J_c(C)$ are also possible — they are extrinsic since $J_c()$ is a function of the entire graph. These include where $J_c(C)$ gives: 1) the tree width of the resulting triangulated modified template; 2) the tree width of the resulting modified chunk $\hat{\mathcal{G}}^c$; 3) the state space of the resulting triangulated graph; or 4) the state-space of the modified chunk. This last one is particularly important since this indicates the degree to which complexity grows with unrolling amount k . Within each of the above lie also the different options for triangulating a graph (heuristics, annealing, etc.) or performing some approximate inference method within $\hat{\mathcal{G}}^c$. With an extrinsic measure, therefore, one is not guaranteed that the left and right interface methods are identical unless one can solve the optimum triangulation problem (which is NP-hard). Moreover, there is no known way to transform the graph into one where the optimal (or even an approximate vertex cut can be found). Again, owing to the fact that the cost of finding the vertex cut is amortized over the life of the model, a very simple heuristic algorithm will consider all (exponential number of) vertex cuts, estimate the cost of $J_c(C)$ for the current cut, and then choose the best. For small graphs, this can be practical. But for larger graphs, or as M gets larger, this quickly becomes completely impractical.

13.3.2.11 Finding M , the vertex cut span

As mentioned above, the highest quality vertex cut using Algorithm 16 might only reveal itself if it is allowed to span $M > 1$ successive original graph chunks. In many of the above examples, $M = 1$ sufficed but we also saw an extreme example in Figure 13.41 which required $M = |\mathcal{G}^c| - 1$.

Theorem 154. *For any intrinsic vertex cut cost function J_c , the minimum cost cut for any $M > |\mathcal{G}^c|$ is the same as the minimum cost cut for any $M = |\mathcal{G}^c|$.*

Proof. TODO: Need a few additional assumptions on J_c to prove it generally. \square

The theorem therefore puts an upper bound on the largest M that might be any benefit, at least for intrinsic cost functions.

13.3.2.12 The chunk skip parameter S

We saw above the vertex cut span M parameter for determining how many successive chunks a vertex cut may lie within. Once we have the resulting modified sections $\hat{\mathcal{G}}^p$, $\hat{\mathcal{G}}^c$, and $\hat{\mathcal{G}}^e$, we are free to use any static inference method we wish, either exact or approximate. The only constraint on the inference method used is that (assuming we are doing left-to-right inference), the inference procedure accepts on the left of a section a message in the form of a clique table over a set of vertices consisting of the interface on the left of a section. Also, the inference procedure must produce on the right a message in the form of a clique table

over the vertices of the interface on the right. The final right message of the t^{th} section becomes the initial left message of the $(t + 1)^{\text{st}}$ section.

There is one additional parameter, called the chunk skip parameter S , that allows for a bit more flexibility when choosing or designing the within-section inference method. Given the final vertex cut C spanning M chunks, one still has an option regarding how many modified chunks to skip between each such cut. We call this the *chunk skip* parameter $S \geq 1$. Given a DGM template, we may partially unroll it $M + S - 1$ times thus allowing room enough for two vertex cuts spaced S chunks apart. That is, the number of original chunks between the two a separator and the next is S . The first boundary is “laid across” the first M chunks, and the second boundary is laid across chunks $S + 1$ through $S + M$. To summarize the M and S parameters: M is the parameter that determines the number of chunks within which a vertex cut can lie. And S is the chunk skip parameter, the number of chunks between the resulting separators that are found.

For example, applying the S parameter to an HMM leads to the possibility of triangulating with a larger clique. An HMM with $S = 2$ corresponds to Figure 13.5 and HMM with $S = 3$ corresponds to Figure 13.6. Now as mentioned earlier, normally one would not want to have such larger cliques, but if one wishes to do a limited-extent asynchronous inference procedure where the clique (or more generally the modified section) limits the extent of the asynchrony, then the S parameter can allow this.

For various M and S values, Figure 13.44 gives some examples of both unmodified chunks (before optimal vertex separator is found) and after (how the modified sections might look) an optimal M -frame spanning vertex cut is found. This figure shows the left interface case, but a symmetric description corresponds to the right interface case. Note that in the basic left interface (when we use the original chunk’s left interface) means that $\hat{\mathcal{G}}^c$ has S chunks and $\hat{\mathcal{G}}^e$ contains both an original epilogue \mathcal{G}^e and an extra M chunks. The reason for the extra M chunks in $\hat{\mathcal{G}}^e$ is that the vertex cut may span M chunks into $\hat{\mathcal{G}}^e$ as shown in each of the examples in Figure 13.44.

In general, why might we want $S > 1$? The reason for the S parameter is that it creates a bigger chunk of graph on which one can perform static inference and this may be advantageous in a number of situations.

Firstly, with a larger modified chunk, certain approximate message passing schemes contained entirely within a modified chunk might have a better chance of converging, or might converge to a better solution since the information available to the chunk is larger (since it spans a larger amount of time).

Secondly, we can use the modified chunk to determine the limits of a given search procedure. Recall in §13.1 the discussion of synchronous vs. asynchronous search in HMM-based systems. In that section, the idea of hybrid synchronous/asynchronous search was discussed, where the junction tree might place limits on the degree of asynchrony. Indeed, rather than a clique placing the limits, a modified chunk can place such limits on the degree of asynchrony. That is, $\hat{\mathcal{G}}^c$ might cover many time frames, and the larger the S parameter the more time frames $\hat{\mathcal{G}}^c$ covers. Within $\hat{\mathcal{G}}^c$, we may perform asynchronous search (which can conceivably be any search method used in either AI, SAT, or constraint satisfaction problems). The S parameter hence gives more flexibility to the degree of asynchrony.

Third, one may wish to perform a do a style of inference corresponding to [61, 62], where every so often (in time) the messages are projected down onto a factored form which allows subsequent messages to continue effectively with lower treewidth, thus bypassing the entanglement problem mentioned in Question 2 and Theorem 149, at the expense of inference accuracy. If it is the case that the projection occurs at every vertex cut, then a larger S parameter allows for a greater “recovery time” between these projections and sources of approximation. The S parameter may be a perfect way to control for the tradeoff between inference quality and computational cost.

13.3.3 Further Template Restrictions

The reader is encouraged to read §8.11.4 before reading this section — indeed, some of the material in section §8.11.4 explains the same concept, although here we are able to justify the restrictions made in



Figure 13.44: Examples of the vertex cut span (M) and the chunk skip (S) parameter for the left interface case, for various values of M and S (each row). On the left column is the template unrolled $M + S - 1$ times, shown as the prologue (in red), chunk (in green), and epilogue (in blue). In the center column is the modified sections using the chunk's basic left interface (i.e., without finding an optimal vertex cut). On the right column is how the modified sections might look after finding a vertex cut that spans M chunks. Note how increasing S leads to the modified chunk spanning larger stretches of time, and therefore allows for more flexibility when designing a limited temporal extent static inference method for the chunk. The right interface case is an exact mirror image of the left interface case (e.g., for $M = 1, S = 1$, the initial modified prologue contains the original prologue and one chunk).

§8.11.4 via an understanding of the M and S parameters.

With a vertex cut spanning M and skipping S chunks, there are additional constraints placed on the frame length of a segment.⁵ Specifically, the number of time slices T in unrolled graphs must satisfy the constraint

$$T = T(\mathcal{G}^p) + T(\mathcal{G}^e) + (M + kS)T(\mathcal{G}^c) \quad (13.14)$$

for some $k \in \{0, 1, 2, 3, \dots\}$.⁶ Therefore, making M and/or S larger reduces the number of valid possible segment lengths. In particular, increasing S also decreases the granularity of T since each length increment is by $ST(\mathcal{G}^c)$.

There are additional constraints, however, that the aforementioned assumptions (can and perhaps should) require on the interfaces themselves.

Firstly, note that even when $M = 1$ and $S = 1$, the first step in Algorithm 16 is to unroll the template one time as is also shown in Figure 13.44-(a). Since the modified template (either using the default interface or the optimal vertex cut interface) is based on the template unrolled one time, any time signal with length

⁵The reader is encouraged to review §8.11.4 regarding DGM template validity.

⁶Note that it is also possible to append extra subgraphs at the end in order to allow for any number of slices.

equal to the original template can no longer be represented. In other words, we must have $T = T(\mathcal{G}^p) + T(\mathcal{G}^e) + 2T(\mathcal{G}^c)$. In general, the initial configuration (using the default interface) with general M and S parameters has $\hat{\mathcal{G}}^c$ consisting of S original chunks and $\hat{\mathcal{G}}^e$ consisting of an original \mathcal{G}^e and an extra M chunks. The reason for the extra M chunks in $\hat{\mathcal{G}}^e$ is that the vertex cut may span M chunks into $\hat{\mathcal{G}}^e$. Hence, the problem becomes more acute as the M and S parameters increase, and indeed we have the restriction $T = T(\mathcal{G}^p) + T(\mathcal{G}^e) + (M + kS)T(\mathcal{G}^c)$ for $k > 0$ (meaning $k = 0$ would be disallowed).

Suppose we are processing a DGM in left-to-right fashion. The next few paragraphs will be written from the perspective of a modified chunk, and we will use the words “we” and “our” from this perspective. Our underlying goal is to be able to have a section of model, say $\hat{\mathcal{G}}^c$, where we may receive some information from the left in the form of a message, do some additional processing (in the form of an exact or approximate inference method), and send it out to the right. The form of message we receive on the left is something that is sent from a section on our left, and hence we need to agree with our left neighbor what we will be receiving. What we expect to receive must be compatible with what our left neighbor will send. Moreover, we need to agree with our right neighbor — what we send must be compatible with what our right neighbor expects to receive. Moreover, Since we can have successive modified chunks ($\hat{\mathcal{G}}^c\hat{\mathcal{G}}^c$) this means that what we receive from our left neighbor is a modified chunk, and what we will send to on the right is also a modified chunk.

Hence, what we receive into a $\hat{\mathcal{G}}^c$ is the same form as the information we send out to our right $\hat{\mathcal{G}}^c$. The only difference, however, is either at the beginning or at the end of time. At the beginning, it is a $\hat{\mathcal{G}}^p$ that sends information to us. Hence unless we special case this message (which would add complexity) we must ensure that what a $\hat{\mathcal{G}}^p$ sends out is compatible with what an $\hat{\mathcal{G}}^c$ sends out (thus implying that what an $\hat{\mathcal{G}}^p$ sends out is compatible with what an $\hat{\mathcal{G}}^c$ receives). Similarly, at the end of time, our last message is sent to an $\hat{\mathcal{G}}^e$. When we send out a message, we never know if what we are sending to is going to be a $\hat{\mathcal{G}}^c$ or a $\hat{\mathcal{G}}^e$ (which only happens once, if at all⁷). Hence, we need to ensure that an $\hat{\mathcal{G}}^e$ receives a message that is compatible with what an $\hat{\mathcal{G}}^c$ sends out. We call this requirement the section interface message compatibility.

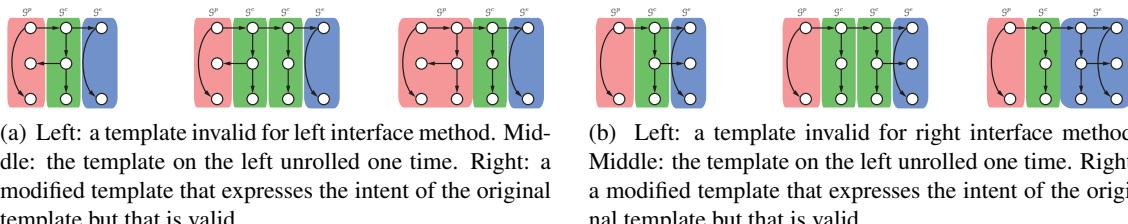


Figure 13.45: Templates can be specified that are invalid for the left or right interface method. When this happens, one must use a slightly wider prologue (in the left interface case) or epilogue (in the right interface case) than what is shown in each example on the right.

This means that any interface between unmodified or modified sections must be compatible. It is possible to specify a template that does not have such a compatibility. For example, consider the templates shown in Figure 13.45. We can see that, in the $M = 1, S = 1$ case, owing to the semantics of where a factor “lives” in the template (see §8.11.2), Figure 13.45a specifies a different left interfaces for the first and second instance of the chunk when the template is unrolled one time. A similar situation occurs in Figure 13.45b where the right interface is different in the first and second chunks of the unrolling. These template, therefore, do not work with our section interface message compatibility requirement above. In each case, we see modified forms of the template that expresses the original intent of the template (i.e., to have an extra child variable

⁷In online or streaming inference, there might never be an $\hat{\mathcal{G}}^e$.

in the prologue in Figure 13.45a or epilogue in Figure 13.45b) but that is valid for the left and right interface method.

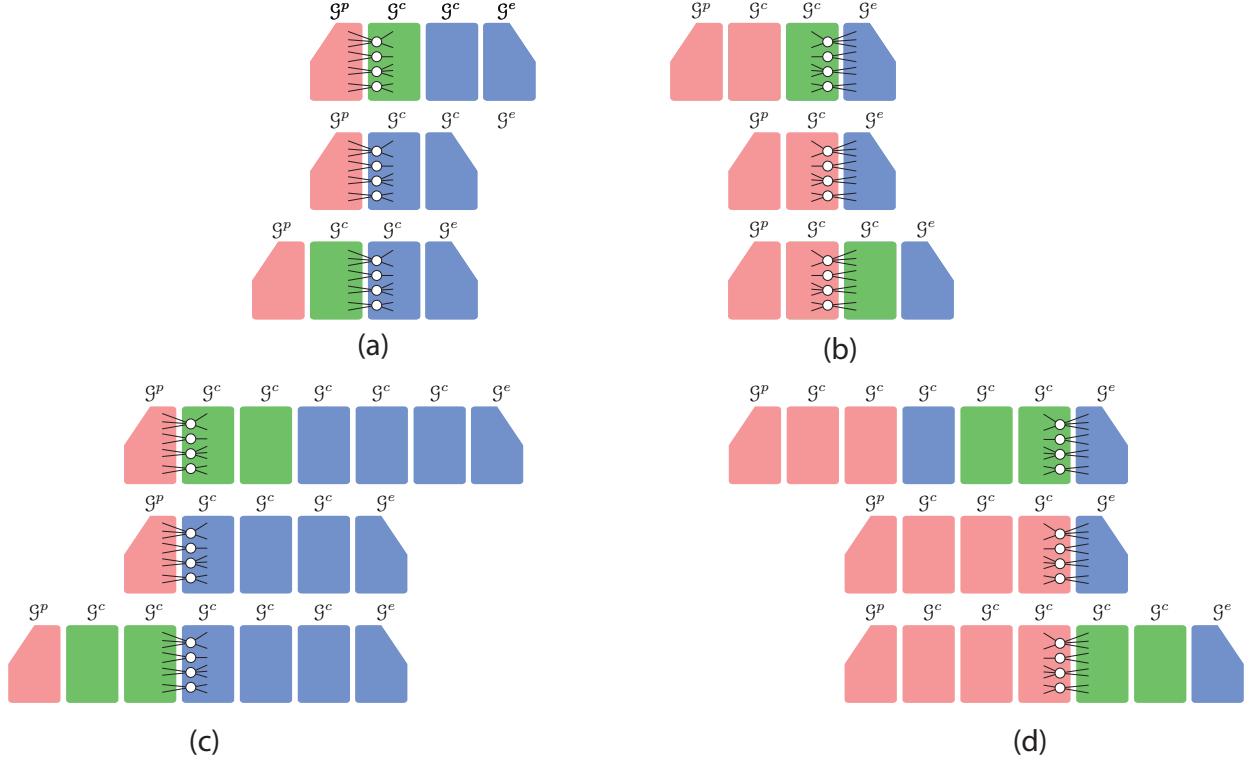


Figure 13.46: Template restrictions based on the vertex cut. In each case, we show the unmodified sections (\mathcal{G}^p , \mathcal{G}^c , \mathcal{G}^e) unrolled a certain number of times, and in each case a vertex cut (the vertices shown) that must be identical (up to at most a time shift) for different amounts of unrolling. (a) the left interface case with $M = 1, S = 1$; (b) the right interface case with $M = 1, S = 1$; (c) the left interface case with $M = 3, S = 2$; (d) the right interface case with $M = 3, S = 2$.

In order to allow for $k = 0$ and to ensure the aforementioned section interface message compatibility requirements are met, therefore, we must place an additional constraint on the template. In order to describe it, assume that $\hat{\mathcal{G}}^p$, $\hat{\mathcal{G}}^c$, and $\hat{\mathcal{G}}^e$ is at its initial state, as shown in the center column in Figure 13.44-(a), before an alternate vertex cut has been found (recall, that the left interface for a chunk also constitutes a vertex cut, if not the optimal one). This means that $\hat{\mathcal{G}}^p = \mathcal{G}^p$, $\hat{\mathcal{G}}^c = \mathcal{G}^c$ and $\hat{\mathcal{G}}^e = \mathcal{G}_{\Delta(-T(\mathcal{G}^c))}^c \mathcal{G}_{\Delta(-T(\mathcal{G}^c))}^e$, where we use the notation AB to denote concatenation, where A and B are sections (or sets thereof). Moreover, define $\mathcal{C}(A, |B|) \subseteq B$ to be the vertex cut between sections A and B contained within B and $\mathcal{C}(A|, B) \subseteq A$ to be the vertex cut between sections A and B contained within A . Then the template restriction, for the left interface, is as follows:

$$\mathcal{C}(\hat{\mathcal{G}}^p, |\hat{\mathcal{G}}^c \hat{\mathcal{G}}^e|) = \mathcal{C}(\hat{\mathcal{G}}^p, |\hat{\mathcal{G}}_{\Delta(-T(\mathcal{G}^c))}^e|) = \mathcal{C}(\hat{\mathcal{G}}^p \hat{\mathcal{G}}^c, |\hat{\mathcal{G}}^e|_{\Delta(-T(\mathcal{G}^c))}), \quad (13.15)$$

and for the right interface case is:

$$\mathcal{C}(\hat{\mathcal{G}}^p \hat{\mathcal{G}}^c |, \hat{\mathcal{G}}^e)_{\Delta(-T(\mathcal{G}^c))}, = \mathcal{C}(\hat{\mathcal{G}}^p |, \hat{\mathcal{G}}_{\Delta(-T(\mathcal{G}^c))}^e) = \mathcal{C}(\hat{\mathcal{G}}^p |, \hat{\mathcal{G}}^c \hat{\mathcal{G}}^e) \quad (13.16)$$

An example of this constraint (for the case where $M = 1, S = 1$ and for the case where $M = 3, S = 2$) is shown in Figure 13.46. In Figure 13.46, while edges incident to the vertex cut is shown only as living in the current unmodified section, we note that these edges might reach beyond the current section in, say, the case

of higher order models. This is further discussed in §13.3.4. We also see that Figure 13.45a-left violates (13.15) and Figure 13.45b-left violates (13.16).

We have the following theorem.

Theorem 155. *Suppose the interface check expressed in (13.15) and (13.16) is satisfied. Then such an interface is compatible with all unrollings.*

Note that once optimal vertex cut is found, as in the right column of Figure 13.44, the vertex cut restrictions are automatically satisfied since the modified sections are now based on the partition of the graph that lies between two vertex cuts.

Also note, it is only in the $M = 1, S = 1$ case above that can we recover the basic template $\mathcal{G}^p, \mathcal{G}^c, \mathcal{G}^e$ by concatenating the sections $\hat{\mathcal{G}}^p\hat{\mathcal{G}}^e$ (i.e., by not using $\hat{\mathcal{G}}^c$). This is the only graphical restriction placed on the template.

13.3.4 Higher order models: recovering the lost Markov property

The reader is encouraged to read §8.11.4 before reading this section.

The temporal Markov property requires that there is a window of time (say from time t to $t + \tau$, for any given t) that renders the past independent of the future. Notationally, for a stochastic process X_t , this becomes $X_{1:t-1} \perp\!\!\!\perp X_{t+\tau+1:T} | X_{t:t+\tau}$ for all t . In many cases, we have $\tau = 0$, so that one *frame* or *slice* of the model is sufficient to render the past and future independent. For example, one of the properties of an HMM (e.g., see Figure 8.44) is that it is simple to see how the past and future are independent given the present — the state Q_t renders past (times before t) and future (times after t) independent. Indeed, Q_t is a vertex cut (separator) in the graph (graphical model).

In a generalized DGM template (§8.11.3), suppose that: 1) \mathcal{G}^p has edges only into the immediate adjacent chunk on the right; 2) \mathcal{G}^e has edges only into the immediate adjacent chunk on the left; and 3) each chunk has neighbors only at most one chunk to the left or one chunk to the right. If this is the case, then the chunk itself, in any unrolled model, will render (when conditioned on) any vertices to the left independent of any vertices to the right. Hence, we retain that the past and future are rendered independent given (at least some notion of) the present which in this case is the chunk. We say that in such case, it is a first order chunk model, since edges never connect more than two successive chunks. A first order chunk model does not require a first order model, however, as within a chunk we can have more than one time frame worth of variables. E.g., if Q_t is the Markov chain variable, we could have that $\mathcal{G}_t^c = \{Q_{2t}, Q_{2t+1}\}$, and even if Q_t has Q_{t-1}, Q_{t-2} as parents, it never reaches back more than one chunk.

In some cases, however, the chunk alone is not sufficient to render past and future independent. This happens when a template specifies a chunk that, once it is unrolled, has incident edges that either reach back (respectively forward) more than one chunk into the past (respectively future). For example, consider the template in Figure 8.86 which has an edge between the prologue and the chunk. When we unroll by $\hat{\tau} = 0$, then the chunk does not render the prologue and epilogue independent. If we unroll by $\hat{\tau} = 1$, as shown on the right in Figure 8.86, then neither of the chunks \mathcal{G}_1^c and \mathcal{G}_2^c separate the past on the future. On the other hand, the two chunks \mathcal{G}_1^c and \mathcal{G}_2^c when grouped together separate the prologue and epilogue (and indeed, in any unrolling in this model, two successive chunks constitute a separator). In general, however, unrolling one time and then considering two successive chunks might not be sufficient to ensure that it contains a temporal separator. This can be seen in Figure 13.47, where it is necessary to unroll twice in order to see a chunk-based Markov property.

The problem is in the flexible definition of a template, where there may be vertices that live (see §8.11.2) in a chunk and which might have incident edges whose other side may be an arbitrary number of chunks into the future or past. Such a template might violate the temporal Markov property, conditioning on a chunk

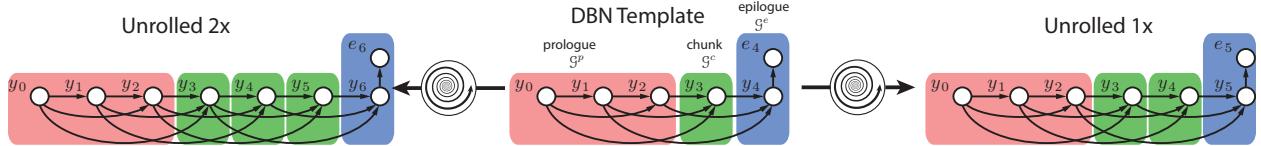


Figure 13.47: A third order model with an epilogue. When we unroll one time (right) there is still an edge between the prologue and epilogue and thus the two chunks $\mathcal{G}_0^c, \mathcal{G}_1^c$ do not render the past and future independent. We don't see a Markov property again until we unroll by at least $\hat{\tau} = 2$ times, where $\mathcal{G}_0^c, \mathcal{G}_1^c, \mathcal{G}_2^c$ is sufficient to render past and future independent. Our goal is to modify the chunk so that it once again renders past and future independent.

will not render the left and right independent since there are edges that can reach over the chunk being purportedly conditioned on.

One option is to restrict the template definition and allow only connections between successive chunks. With such a restriction, there are several options for expressing higher order models. The first option is to use multi-frame chunks. The problem with this, however, is that by forcing a $T(\mathcal{G}^c)$ to be larger, the template represents a smaller class of models. Indeed, since a segment of length T must satisfy $T = T(\mathcal{G}^p) + T(\mathcal{G}^e) + (M + kS)T(\mathcal{G}^c)$ for some $k > 0$, the larger $T(\mathcal{G}^c)$ becomes, the granularity of valid T values becomes coarser. Another issue with this approach is that since the chunk contains more random variables, inferring an inference procedure for the chunk becomes more difficult as well.

The second option is to use larger domain sized random variables, very similar to how a higher order Markov chain can be represented by a lower order Markov chain (described in §8.3.0.1). This is so that a suitably rich history or future context can be represented.

Ideally, we'd like to express higher order models directly in the template without such restrictions and/or requirements of the model designer. Moreover, it may be that only a small set of random variables within the chunk reaches beyond an adjacent chunk, with most of the variables having neighbors living either entirely within the chunk or residing in an adjacent chunk.

We next discuss a third option, which is an operation that will recover the chunk-based Markov property but that doesn't further restrict T and does not require a larger domain size for the random variables. That is, we will take the three initial sections (lets call them \mathcal{G}^p , \mathcal{G}^c , and \mathcal{G}^e) and modify them to become the sections $\hat{\mathcal{G}}^p$, $\hat{\mathcal{G}}^c$, and $\hat{\mathcal{G}}^e$. Note that we've used this notation before, initially described in §13.3.2.1, where we modify the initial sections in order to include the basic left and/or right interface (e.g., see Figure 13.14b, Figure 13.16b, Figure 13.18, and other similar figures in §13.3.2.1), or where we modify the sections of a template based on having found the optimal vertex separator in some way (e.g., see §13.3.2.7 and §13.3.2.9). Note that in the original template, the three sections are really a partition — there is no vertex overlap between them, while after the interfaces have been included, the intersection between successive sections is exactly the interface. As mentioned in §13.3.2.1, however, there might be multiple stages of transformation from original sections of a template to modified sections before we reach the final set of modified sections that are used for inference. The modification of the template we next discuss is one that occurs *before* any of the subsequent modifications mentioned in §13.3.2.1.

The modified template will consist of the three modified sections, and an unrolling will be based on these modified (or further modified as per §13.3.2.1) sections. In any unrolling, the Markov property will be true w.r.t. these newly modified sections, i.e., $\hat{\mathcal{G}}^c$ will render the past and future independent given any unrolling. Clearly, we will not simply repeat \mathcal{G}^c to get $\hat{\mathcal{G}}^c$ as that has the very same granularity increase problem mentioned above.

Before defining the transformation, lets consider an issue that will turn out to be important in forming

the modified sections. Given two random variables X and Y , consider the independence statement

$$X \perp\!\!\!\perp Y | Y. \quad (13.17)$$

At first, it might seem superfluous (or worse incorrect) to make such a statement. Not only, however, does the statement make sense but it is always true, and in fact can be useful in the current context to recover the Markov property. To make sense of this statement, consider the following:

$$p(X = x, Y = y | Y = y) = p(X = x | Y = y, Y = y)p(Y = y | Y = y) \quad (13.18)$$

$$= p(X = x | Y = y)p(Y = y | Y = y) \quad (13.19)$$

where the first equality comes from the definition of conditional probability, and the second since conditioning on a redundantly stated event $\{Y = y, Y = y\}$ is the same as conditioning on the non-redundantly stated event $\{Y = y\}$. The right hand side is the factorization corresponding to the independence statement (in general if $A \perp\!\!\!\perp B | C$ then $p(a, b | c) = p(a | c)p(b | c)$). Intuitively, (13.17) follows since once we know everything about a random variable Y (by conditioning on Y), then no other source (such as X) can be informative about Y — once we know everything about Y , X can not further reduce the entropy of Y , and hence conditional independence should hold.

We next show how we can use this property to recover the Markov property for chunks. That is, unlike the three sections of the original template, which are really blocks of a partition and hence do not intersect, the newly modified sections will (if there is a Markov property to be recovered) have an intersection, and these intersections are called the section separators $\dot{\mathcal{G}}^p \cap \dot{\mathcal{G}}^c$ and $\dot{\mathcal{G}}^c \cap \dot{\mathcal{G}}^e$ but there is no $\dot{\mathcal{G}}^p \cap \dot{\mathcal{G}}^e$ unless an unrolling by 0 (which corresponds to an original template). Note that the section separators will also render the past and future independent. As mentioned above, this process is done before any optimal vertex cut is discovered.

Whenever possible, we wish not to increase the size of the chunk. While for higher order models, increasing its size will be inevitable, in most cases it will not increase as much as the aforementioned naïve way of accomplishing this goal (namely, duplicating the entire chunk).

In order to render the graph first-order Markov at the granularity of chunks, there has to be a separation property (past and future separated by present). The way this is done is to consider a template unrolled one time, resulting in sections $(\mathcal{G}^p, \mathcal{G}_1^c, \mathcal{G}_2^c, \mathcal{G}^e)$. There could then be vertices in \mathcal{G}_1^c with neighbors in \mathcal{G}^e or vertices in \mathcal{G}_2^c with neighbors in \mathcal{G}^p . There could also even still be variables in \mathcal{G}^p with neighbors in \mathcal{G}^e , and vice versa if the model order is high enough.

Based on Equation (13.17), we can essentially copy variables from one section to another and treat them as if they resided in the destination section of the copy. Consider next the following process to recover the temporal Markov property.

- 1) Unroll template $1 \times$ resulting in a template $(\dot{\mathcal{G}}^p, \dot{\mathcal{G}}^c, \ddot{\mathcal{G}}^c, \dot{\mathcal{G}}^e)$
- 2) variables in $\dot{\mathcal{G}}^e$ adjacent to variables within $\dot{\mathcal{G}}^c$ or $\dot{\mathcal{G}}^p$ are copied into $\ddot{\mathcal{G}}^c$, i.e.:

$$\ddot{\mathcal{G}}^c \leftarrow \ddot{\mathcal{G}}^c \cup \left[(\Gamma(\dot{\mathcal{G}}^c) \cup \Gamma(\dot{\mathcal{G}}^p)) \cap \dot{\mathcal{G}}^e \right] \quad (13.20)$$

- 3) variables in $\dot{\mathcal{G}}^p$ adjacent to variables within $\ddot{\mathcal{G}}^c$ or $\dot{\mathcal{G}}^e$ are copied into $\dot{\mathcal{G}}^c$, i.e.:

$$\dot{\mathcal{G}}^c \leftarrow \dot{\mathcal{G}}^c \cup \left[(\Gamma(\ddot{\mathcal{G}}^c) \cup \Gamma(\dot{\mathcal{G}}^e)) \cap \dot{\mathcal{G}}^p \right] \quad (13.21)$$

- 4) any additions to $\dot{\mathcal{G}}^c$ (resp. $\ddot{\mathcal{G}}^c$) are placed in $\ddot{\mathcal{G}}^c$ (resp. $\dot{\mathcal{G}}^c$) with the appropriate time-frame adjustment, so that the final resulting $\dot{\mathcal{G}}^c$ and $\ddot{\mathcal{G}}^c$ are again identical except for a time shift. I.e.

$$\dot{\mathcal{G}}^c \leftarrow \dot{\mathcal{G}}^c \cup \ddot{\mathcal{G}}_{-\Delta(T(\mathcal{G}^c))} \text{ and } \ddot{\mathcal{G}}^c \leftarrow \ddot{\mathcal{G}}^c \cup \dot{\mathcal{G}}_{\Delta(T(\mathcal{G}^c))}^c \quad (13.22)$$

- 5) Modified template becomes

$$(\hat{\mathcal{G}}^p, \hat{\mathcal{G}}^c, \hat{\mathcal{G}}^e) \leftarrow (\dot{\mathcal{G}}^p, \dot{\mathcal{G}}^c, \dot{\mathcal{G}}^e_{\Delta(-T(\mathcal{G}^c))}) \quad (13.23)$$

which might no longer be a partition.

In a junction tree procedure, such variable copying would correspond to variables expanding out in the tree, but without violating the running intersection property — any sub-forest corresponding to the variables being copied would still be connected and hence would still be a tree. Any message passing algorithm would treat such variables in the subtree as a copy. Once the above procedure is done, therefore, we are certain that in any unrolling, $\mathcal{G}_{1:t-1}^c \perp\!\!\!\perp \mathcal{G}_{t+1:T}^c | \mathcal{G}_t^c$.

We next give an example. Suppose $\mathcal{G}^p = \{Y_0, Y_1\}$, $\mathcal{G}^c = \{Y_2\}$, and $\mathcal{G}^e = \{Y_3, E_3\}$, and where the model involves the factor $p(y_t | y_{t-1}, y_{t-2})$, as shown in Figure 8.86. Thus, there is an edge between \mathcal{G}^p and \mathcal{G}^e .

The template, unrolled one time, becomes $(\dot{\mathcal{G}}^p, \dot{\mathcal{G}}^c, \ddot{\mathcal{G}}^c, \dot{\mathcal{G}}^e) = (\{Y_0, Y_1\}, \{Y_2\}, \{Y_3\}, \{Y_4, E_4\})$ and we can see that the chunks do not render the past and future independent. Once we change the template according to the above procedure, however, the modified and unrolled template becomes $(\dot{\mathcal{G}}^p, \dot{\mathcal{G}}^c, \ddot{\mathcal{G}}^c, \dot{\mathcal{G}}^e) = (\{Y_0, Y_1\}, \{Y_1, Y_2, Y_3\}, \{Y_2, Y_3, Y_4\}, \{Y_4, E_4\})$ where $\mathcal{G}_t^c = \{Y_{t-1}, Y_t, Y_{t+1}\}$. Conditioning the past and future on the chunk, we get:

$$\{Y_0, Y_1, \dots, Y_t\} \perp\!\!\!\perp \{Y_t, Y_{t+1}, \dots, Y_T, E_T\} | \{Y_{t-1}, Y_t, Y_{t+1}\} \quad (13.24)$$

which holds by Equation (13.17) and by the properties of the template. Note that the separator $\{Y_1, Y_2, Y_3\} \cap \{Y_2, Y_3, Y_4\} = \{Y_2, Y_3\}$ between two sections also renders past and future independent (analogous to an HMM).

Theorem 156. *Above operation guarantees that in any unrolling, $\hat{\mathcal{G}}_{1:t-1}^c \perp\!\!\!\perp \hat{\mathcal{G}}_{t+1:T}^c | \hat{\mathcal{G}}_t^c$.*

Proof. Consider only one direction (other is symmetric). Any variable to right of following chunk is now in following chunk, so given the following chunk, anything in subsequent chunks is independent, which follows from Equation (13.17). \square

Moreover, any so modified template is Markov w.r.t. the separators as well. I.e., with $\mathcal{S}_t = \mathcal{G}_t^c \cap \mathcal{G}_{t+1}^c$, we see that $\hat{\mathcal{G}}_{1:t-1}^c \perp\!\!\!\perp \hat{\mathcal{G}}_{t+1:T}^c | \mathcal{S}_t$.

One last point of discussion regarding the above is minimality. That is, are we certain that the addition to the chunk is the least that can be added while ensuring the temporal Markov property holds? Indeed, in the above example, a modified template that takes the form $(\dot{\mathcal{G}}^p, \dot{\mathcal{G}}^c, \ddot{\mathcal{G}}^c, \dot{\mathcal{G}}^e) = (\{Y_0, Y_1\}, \{Y_1, Y_2\}, \{Y_2, Y_3\}, \{Y_4, E_4\})$ where $\mathcal{G}_t^c = \{Y_t, Y_{t+1}\}$ would also have the temporal Markov property. While any message passing scheme involving cliques in a junction tree that have many common variables would still be valid, those variable copies essentially on any implementation be copied from clique to clique, thereby leading to wasted computation.

In order to minimally augment the chunk to restore the Markov property, we must use the factor and edge's residence (cf. §8.11.2) to determine which vertices to copy. Recall, that an edge has a residence corresponding to one of the residences of its incident vertices. To do this, we modify steps three and four above to instead read:

- 2) variables in $\dot{\mathcal{G}}^e$ adjacent to variables within $\dot{\mathcal{G}}^c$ or $\dot{\mathcal{G}}^p$ via incident edges that reside in $\dot{\mathcal{G}}^c$ or $\dot{\mathcal{G}}^p$ are copied into $\dot{\mathcal{G}}^c$.
- 3) variables in $\dot{\mathcal{G}}^p$ adjacent to variables within $\ddot{\mathcal{G}}^c$ or $\dot{\mathcal{G}}^e$ via incident edges that reside in $\dot{\mathcal{G}}^c$ are copied into $\dot{\mathcal{G}}^c$.

It can be seen that if we use this prescription, the above example will achieve the two variable modified chunk $\hat{\mathcal{G}}_t^c = \{Y_t, Y_{t+1}\}$ which minimally separates past and future.

Theorem 157. *The modified operation guarantees that in any unrolling, $\hat{\mathcal{G}}_{1:t-1}^c \perp\!\!\!\perp \hat{\mathcal{G}}_{t+1:T}^c | \hat{\mathcal{G}}_t^c$. Moreover, the operation minimally achieves this.*

13.3.5 Online inference, Kalman smoothing and filtering and sections and interfaces

The GMTK standard: Definition of zero additional lag (for the moment not considering the epilogue, infinite segment length).

HMM case: as soon as x_t is available, it is possible to produce $p(y_{t-\tau}|x_0, \dots, x_t)$ immediately (not counting any inference cost), as well as (in the Viterbi case), produce $y_{t-\tau}^* \in \text{argmax}_{y_{t-\tau}} p(y_{t-\tau}|x_0, \dots, x_t)$.

General case, lag is defined in terms of modified sections. I.e., we must have all of the observed variables in a section before we can compute the posterior probability of any variable in the current section, or any previous section. But once all the observed variables in the t^{th} section are available, we can compute the posterior of any variable in the current or previous section conditioned on all of the observations we've so far received.

13.3.6 Cases where no monolithic interface is optimal for original template: factored interfaces

So far, we have considered an interface that not only is connected but that ends up being completed (and so is a clique or a subclique). There are some valid templates, however, where this completion process will always lead to suboptimality. Moreover, there are a number of DGM specific approximate inference methods whereby the interface itself is deliberately approximated via a factored form in order to improve computational properties.

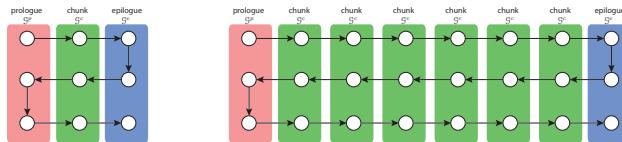


Figure 13.48: A model where no completed interface is optimal. The template (left), and any amount of unrolling of the template (e.g., right), is a model with treewidth one. If we completed the interface, however, we would increase the treewidth of the model to three.

Consider the template shown in Figure 13.48. Since each vertex is adjacent to itself in neighboring frames, each vertex is incident to a persistent edge. Hence, the interface (either left or right, or any optimal vertex cut using any criterion) will be of size three. Thus, the template model, which clearly corresponds to a tree-width graphical model for any unrolling, can't be optimally dealt with because of the compulsory completion process.

One strategy is to have factorizable interfaces. I.e., an interface could itself consist of a group of sub-interfaces each of which are completed (and each of which might even consist of a single vertex), but there is no notion of compulsory completion as described in §13.3.1.3 across every vertex of the interface. Indeed, such a strategy would account for models such as Figure 13.48 since each of the variables in a chunk would be one of the sub-interfaces, and there would be, say, three separate messages sent between neighboring section.⁸

⁸Technically, a strictly left-to-right or right-to-left inference strategy for Figure 13.48 would still be suboptimal since eliminating variables in such a pattern would create fill-in edges. The only strategy that would be optimal for this model would be to zig-zag,

Another class of models that could utilize factored interfaces would be a set of independent parallel Markov chains. Since each Markov chain is independent, it would be wasteful to compulsorily complete their interface between successive sections.

We note also at this point that the approximate inference strategies described in [61, 62] correspond precisely to factored default interfaces (further discussed in §13.4.5.3). Namely, in this work, one takes the default left or right interface (§13.3.2.2 and §13.3.2.3), and “project” it down to a factored interface which is then passed as messages to a neighboring section⁹. This then allows the next section to proceed with an interface method that has fewer constraints, since its incoming interface has not necessarily been completed.

13.3.7 Summary

To summarize this section, we first take the default template, add variables to the chunk to ensure that it satisfies the temporal Markov property at the level of the modified chunk, and then further unroll and partition the template in order to find the best interface.

13.4 Inference strategies for DGM sections

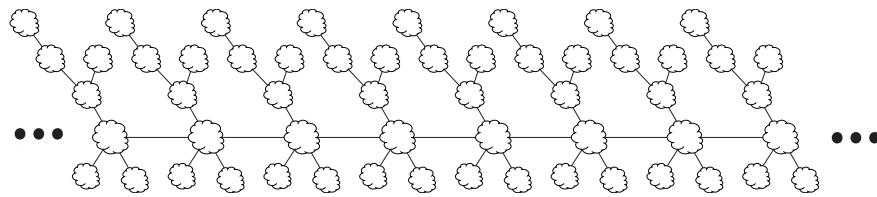


Figure 13.49: An example of what a possible junction tree for a DGM might look like once expanded.

An exact DGM junction tree has a distinct looking structure, typically where there is one backbone chain with sub-trees hanging off the side, like a very tall tree that never gets skinnier as one goes higher (see Figure 13.49). Sometimes, there is only one backbone chain with no sub-trees (i.e., a chain of cliques), an HMM is one example, but there are other structures where a chain of cliques is suboptimal. In general, this unique shape can be exploited by certain practical and theoretical techniques to make inference fast. Many of the the techniques rely on a vertex-cut based graph partitioning mentioned in earlier sections.

In general, what makes DGM inference hard? For some applications, DGM inference is hard because the state space is large. E.g., in speech recognition, we might have a 200,000 word vocabulary, one to two pronunciations per word, about five states per word pronunciation, three sub-states per state, leading to an overall state space in an HMM system of approximately $200,000 \times 1.5 \times 5 \times 3 = 4.5M$, and $(4.5M)^2 \approx 2 \times 10^{13}$. This makes a HMM queries hard to compute despite the fact that its maxclique size is only two. For other applications (e.g., genomics), the state spaces might be relatively small (perhaps only a few hundred or a thousand at most) but the sequence is very long (length of human genome is more than 3 billion base pairs). Hence, in an HMM, a temporal expansion might last for a very long time. In still other applications (modern/research genomics), both of the above might be true simultaneously. It is in such cases where a more structured state space can be beneficial. Ideally, we would like an inference strategy that can handle all of the above cases.

A main goal of §13.3 was to take an existing template $G = (V, E)$ consisting of three (unmodified) sections \mathcal{G}^c , \mathcal{G}^p , and \mathcal{G}^e , and transform them into modified sections $\hat{\mathcal{G}}^c$, $\hat{\mathcal{G}}^p$, and $\hat{\mathcal{G}}^e$. While the original

first left-to-right, then right-to-left, and then again left-to-right, in each case eliminating only some of the variables in each section. While the template specified in Figure 13.48 is mathematically valid, its practical utility for any real application is yet to be seen.

⁹We note that [61, 62] use very different terminology and moreover do not have the generalized notion of a DGM template or non-default left/right interfaces with which to factor.

sections do not intersect, the modified sections do, and in fact intersect at exactly the separators that allow successive sections to temporally communicate with each other. We've discussed how these separators have to agree (i.e., that the interface on $\hat{\mathcal{G}}^e$'s left must agree with the interface both on $\hat{\mathcal{G}}^c$'s and $\hat{\mathcal{G}}^p$'s right, and that these interfaces are shared with other sections to ensure this).

The resulting three modified DGM sections are otherwise practically identical to any normal static graphical model, they consists of a set of vertices and edges (random variables and factors). Hence, any inference method that can be used for a static graphical model can also be used for each of the sections as long as it respects the additional constraints required at the section interfaces. For example, an exact inference procedure that forms a triangulation (and a resultant junction tree) of each section can be used — since the separators produced via the results of Section 13.3.2 are compulsorily completed, and since any triangulation of a section will not create any new edges beyond the boundaries of a section, the DGM separation property will still hold regardless of the triangulation used, even if the section becomes one large clique (e.g., Figure 13.33-IV). Alternatively, an approximate inference scheme can be used if (assuming, for the moment, a left-to-right interface strategy) it consumes information from the left section via the interface on the left, and produces information for the right section via the interface on the right.

This is where we see how from only the template itself, we can compute the cost of exact inference for any unrolling length. The chunk can be, for example, triangulated using any off-line triangulation scheme [368, 367, 243, 246, 36, 14], and the resulting maximum clique size of the triangulation then provides an upper bound on the cost of inference. That is, analogous to the static case, if the resultant maximum clique size of the chunk is $\omega + 1$ then the time cost of inference will become $O(TN^{\omega+1})$. Moreover, the up-front cost spent performing triangulation is amortized over the life of the DGM, since the computational properties of the triangulation will hold for any T . Moreover, good triangulation (or perhaps even optimal triangulation) is feasible since the modified sections are typically much smaller than is the original template unrolled by an amount needed for a particular segment.

The approach described above (of first finding the optimal interface separator and then optimally triangulating each of the resulting sections) has the potential to recover the optimal inference scheme for most any DGM (with further flexibility granted if we allow for factored interfaces, as mentioned in §13.3.6). The reason can be seen by considering the alternative, unrolling for a fixed T and then finding the optimal inference procedure in that unrolled graph for the given T . Since T can grow unboundedly, it makes sense only to eliminate nodes in some (bounded monotone) left-to-right order (see §13.3.2), but as mentioned above, it need not be the case that one chunk \mathcal{G}^c should be fully eliminated before moving onto the next chunk. However, with a modular score function (§13.3.2.10) and a first-order model, there would never be a utility in a succession of more than $|V(\mathcal{G}^c)|$ partially eliminated chunks since at that point the benefit of eliminating the latest node could be applied to one of the earlier ones.

When an optimal separator (§13.3.2.9) can be found in a sequence of no more than m chunks, then setting the right value of the vertex cut parameter $M \geq m$ allows for an optimal elimination scheme in a pre-unrolled graph. And since the elimination scheme also corresponds to a triangulation, finding first the interface separator and then triangulating the resulting sections (via the same elimination scheme applied internally to the sections) has exactly the same inference cost options as the unroll and triangulate approach.

In the remainder of this section, we assume that the template and its original three sections \mathcal{G}^p , \mathcal{G}^c , and \mathcal{G}^e (which form a partition of the template) have been modified according to the techniques in §13.3 into modified sections $\hat{\mathcal{G}}^p$, $\hat{\mathcal{G}}^c$, and $\hat{\mathcal{G}}^e$ where the modified sections no longer partition the template, what we assume is that the intersections between $\hat{\mathcal{G}}^p$ and $\hat{\mathcal{G}}^c$, between $\hat{\mathcal{G}}^c$ and a time-shifted version of $\hat{\mathcal{G}}^c$, and between $\hat{\mathcal{G}}^c$ and $\hat{\mathcal{G}}^e$ constitute the separators between these sections. We also assume the modified template is first-order (meaning no section has an edge incident to more than one earlier or later section, see §13.3.4).

We start off by describing (and reminding the reader) what generic message passing in a dynamic junction tree looks like in this context.

Next, we give an outline of one method that has been partially inspired from the methods used in large-

vocabulary speech recognition (where state spaces can be in the tens of millions), and that exploit sparsity, and that has been used for GMTK. There are several sides to this approach. The first is the idea of a hybrid synchronous/asynchronous search (see §13.1) in concert with message passing, where the degree of asynchrony is limited by the cliques in a junction tree, as mentioned earlier in Section 13.1. The second idea is also search related, and corresponds to the fact that expanding a clique in a junction tree as the result of a message (i.e., Equation (13.31)) can itself be seen as a search procedure [373], and any of the standard methods for search [373, 108, 8, 7, 289, 458, 459, 457, 221, 220, 456] can be used just within and bounded by a single clique. Using the ideas from search within a clique helps to exploit and preserve sparsity, and value-specific independence — this is helpful if there is any sparsity or determinism in the model, but is also particularly useful in combination with the approximate inference schemes mentioned below in Section 13.4.5.1.

Third, since chunks are being repeatedly expanded in a left-to-right fashion, and since it is a very related set of cliques that are expanded and then projected (namely, it is a set of random variables that have the same origin in the template, but are shifted in time by an amount $T(\hat{\mathcal{G}}^c)$), whatever happens at chunk t can be used to help perform the expansion at chunk $t + 1$. In other words, there is often a form of temporal locality (or smoothness), even at the level of doing inference. For example, certain sparse factors or pruning decisions that are determined at chunk t can be useful at chunk $t + 1$. This last point is particular to DGMs and is one of the things that makes performing inference on models as large as Figure 9.8 possible. We describe each of these in the next several sections.

13.4.1 Generic Messages in a DGM Junction Tree

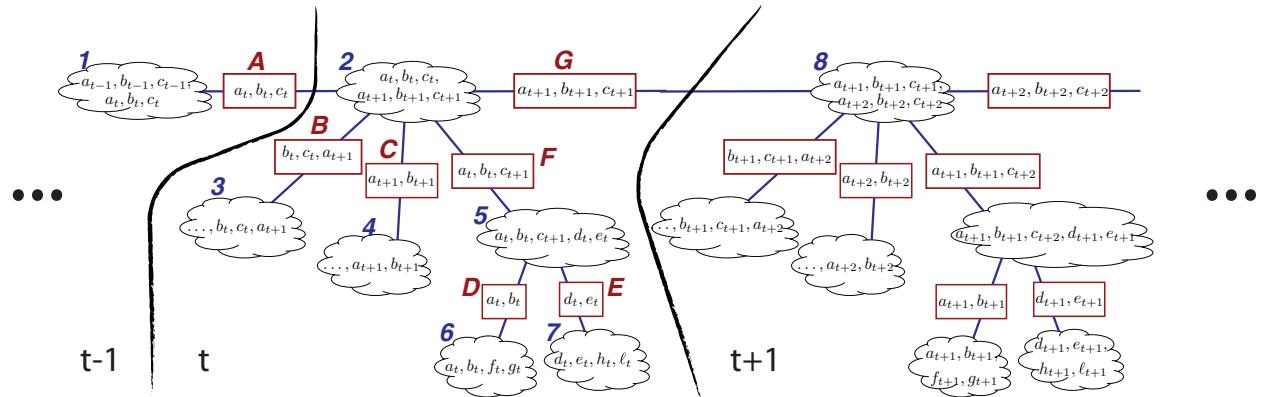


Figure 13.50: Example of a temporal junction tree corresponding to one section at time t and bits of previous and following junction tree sections. Each cliques is given a number (in blue) and each separator is given a letter (in red). Separator A is the interface separator on the left of section t and separator G is the interface on the right of the section. Note that G is shared both as the right separator of section t and the left separator in section $t + 1$.

Once we have strung together these bits of junction trees (e.g., see any of Figure 13.33-I through IV), we may perform a forward and then backward set of messages, very similar to an HMM.

The cliques within a section form a sub-junction tree of what the full junction tree of a fully unrolled model would look like (Figure 13.49). To pass messages according to the message-passing protocol (MPP) we must choose an message ordering of the cliques in each section that is sure to obey MPP once the template has been expanded. The interface to the left is a separator between a clique in the current section and a clique in current section (e.g., separator A in Figure 13.50) — this current section’s clique receives the first message of the section (e.g., clique 2 in Figure 13.50). The interface on the right is a separator (e.g.,

separator G Figure 13.50) between a clique in the current section and a clique in the right section, and the corresponding current section's clique becomes the root clique of the section (clique 2 is the root clique of the section at time t in Figure 13.50). Once a root is designated, the local section's junction tree becomes directed and we may choose any MPP-respecting order of these cliques (e.g., an in-order, pre-order, or post-order of the cliques relative to clique 2 in Figure 13.50). Messages are then collected to this local root clique, and then projected down to the “output” interface separator of the section. For example, in Figure 13.50, messages might proceed in the order $1 \rightarrow A, 6 \rightarrow D, 7 \rightarrow E, D, E \rightarrow 5, 5 \rightarrow F, 3 \rightarrow B, 4 \rightarrow C$, and then $A, B, C, F \rightarrow 2$. Once things have collected into this “output” interface separator (message $2 \rightarrow G$), it becomes an additional factor to be used as the “input” interface at the beginning of the junction tree segment corresponding to the next section on the right (i.e., the message involving G to 8).

When a junction tree corresponding to a DGM is one long chain (meaning there is only one clique per section), the forward messages (unsurprisingly, given §6.2) are very similar to the α -recursion of an HMM (see Equation (8.115) or (8.140)). Let C_t be the clique variables corresponding to the section at time t and let $S_t = C_t \cap C_{t+1}$ be the separator variables between successive cliques, and let X be our set of random variables. Thus, the generic DGM generalization of the forward message in this case is:

$$\phi(x_{S_{t+1}}) = \sum_{x_{C_{t+1} \setminus S_{t+1}}} \psi(x_{C_{t+1}}) \phi(x_{S_t}) \quad (13.25)$$

Note that this is really a direct separator-to-separator message, as the clique table is never expanded in memory (similar to the HMM α recursion).

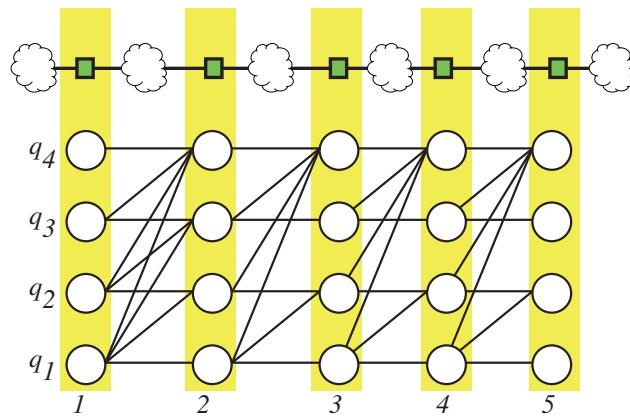


Figure 13.51: An HMM trellis (below) and its junction chain of cliques and separators above.

As an analogy, consider the HMM where the interface separator consists of the variable Q_t , the clique consists of the pair Q_t, Q_{t+1} , and the outgoing separator consists of Q_{t+1} . The forward (α recursion) message from Equation (8.115) (and repeated below) takes the following form

$$\alpha_t(q) = \sum_r p(x_t|Q_t = q)p(Q_t = q|Q_{t-1} = r)\alpha_{t-1}(r). \quad (13.26)$$

The previous separator table is the alpha array $\alpha_{t-1}(r)$ and is of size $|D_Q|$, the clique expansion is $p(x_t|Q_t = q)p(Q_t = q|Q_{t-1} = r)\alpha_{t-1}(r)$ and is of size $|D_Q|^2$, and the outgoing separator is $\alpha_t(q)$ and is of size $|D_Q|$. See Figure 13.51. But the form of Equation (13.26) is from separator directly to separator, and the clique is only implicitly expanded — there is never a table of size $|D_Q|^2$ needing to be stored in memory.

In a junction tree, however, often the cliques are the objects collecting from separators. For example, we

may consider passing a message from cliques C_t directly to C_{t+1} which would be performed as follows:

$$\phi(x_{S_t}) = \sum_{x_{C_t \setminus S_t}} \psi(x_{C_t}) \quad (13.27a)$$

$$\psi(x_{C_{t+1}}) = \psi(x_{C_{t+1}})\phi(x_{S_t}) \quad (13.27b)$$

The separator-to-separator message better memory properties (since separators are generally smaller) and corresponds to the $O(NT)$ memory achieved by an HMM. Using clique-to-clique messages in an HMM would require $O(N^2T)$ memory as well as compute requirements. Indeed, this is precisely why the vertex-cut algorithm of §13.3.2.9 is memory optimal.

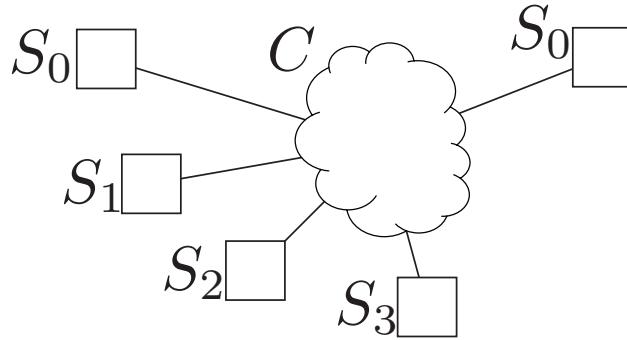


Figure 13.52: Separator-to-separator message with four incoming separators S_1 through S_4 and one outgoing separator S_0 .

In a junction tree, the separator-to-separator message idea can be generalized to an arbitrary junction tree clique node where clique tables are never explicitly stored, only separators. Rather than only one incoming separator as in an HMM, there can be an arbitrary number of incoming separators, depending on the degree of the junction tree node (specifically, if the degree is d , then there are $d - 1$ incoming separators and one outgoing separator). Suppose we have a clique C with a set of four incoming separators S_1, S_2, S_3 , and S_4 , and one outgoing separator S_0 . This is shown in Figure 13.52. A standard message would first collect from the separators into a clique table as follows:

$$\psi'(x_C) = \psi(x_C)\phi(x_{S_1})\phi(x_{S_2})\phi(x_{S_3})\phi(x_{S_4}) \quad (13.28)$$

an operation that would require $O(|D_{X_C}|)$ operations. This would then be used to project down to the outgoing separator as follows:

$$\phi(x_{S_o}) = \sum_{x_{C \setminus S_o}} \psi'(x_C) \quad (13.29)$$

A more direct separator-to-separator message would be formed as follows:

$$\phi(x_{S_o}) = \sum_{x_{C \setminus S_o}} \psi(x_C)\phi(x_{S_1})\phi(x_{S_2})\phi(x_{S_3})\phi(x_{S_4}) \quad (13.30)$$

where the clique table $\psi'(x_C)$ is never stored in its entirety. If the separators are smaller than the clique, this can be a memory savings (as is the case with an HMM).

In order to ensure that it is clear what is meant by the aforementioned messages, we next give an example where the variables being used are given explicitly. where we assume all variables are discrete. We

use clique 2 in Figure 13.50 for our example. We say that say that $\psi(a_t, b_t, c_t, a_{t+1}, b_{t+1}, c_{t+1})$ is a clique “function”, which really means it is a table of scores indexed by the variables a_t through c_{t+1} . During the collect evidence phase, we have the separator functions $\phi(a_t, b_t, c_t)$, $\phi(b_t, c_t, a_{t+1})$, $\phi(a_{t+1}, b_{t+1})$, and $\phi(a_t, b_t, c_{t+1})$. The overall expanded clique function corresponding to clique 2, with all separator information incorporated into the clique table then becomes the following quantity:

$$\psi'(a_t, b_t, c_t, a_{t+1}, b_{t+1}, c_{t+1}) \triangleq \psi(a_t, b_t, c_t, a_{t+1}, b_{t+1}, c_{t+1})\phi(a_t, b_t, c_t)\phi(b_t, c_t, a_{t+1})\phi(a_{t+1}, b_{t+1})\phi(a_t, b_t, c_{t+1}) \quad (13.31)$$

Once this clique table is constructed, we need to “project” down to the outgoing separator of the clique, and this is done (where the outgoing separator involves the variables $a_{t+1}, b_{t+1}, c_{t+1}$) as follows:

$$\phi(a_{t+1}, b_{t+1}, c_{t+1}) \leftarrow \sum_{a_t, b_t, c_t} \psi'(a_t, b_t, c_t, a_{t+1}, b_{t+1}, c_{t+1}) \quad (13.32)$$

Note that the clique table function $\psi'(a_t, b_t, c_t, a_{t+1}, b_{t+1}, c_{t+1})$ may have a large number of possible values of the tuple $(a_t, b_t, c_t, a_{t+1}, b_{t+1}, c_{t+1})$ that have zero score. It is hugely advantages computationally not to store these zeros, and moreover it is advantageous to figure out that large swaths of values have zero score as soon as possible (so as to not waste computation time computing zero) — this is discussed further in §13.4.3. Moreover, in some cases it may be that the model is internally inconsistent so that there is no value of $(a_t, b_t, c_t, a_{t+1}, b_{t+1}, c_{t+1})$ yields a non-zero score — when this happens, there is either an error in the specification of the model or too aggressive approximate inference is being used, and it is called a zero clique error.

The process of gathering from ones incoming separators of a clique, expanding the clique table, and then projecting down to the outgoing separator continues within each section from left to right, culminating in the “root” clique of the final junction tree segment on the right if there is one (in online inference, this process continuous indefinitely). The procedure may also be seen as a collect evidence stage [207, 208].

In the backwards (or distribute evidence) case, messages are sent out of the root clique in the junction tree. When messages reach the interface separator, they are used as the new root at the right of the previous junction tree segment. This continues back until the first junction tree segment.

Most graphical model inference procedures, however, do not simply run an elimination algorithm to find a triangulation, produce a junction tree, and then run a naïve junction tree procedure. Indeed, such methods often do not exploit much of the structure that can exist not only graphically, but also within the parameters of the factors themselves. These include exact inference strategies such as exploiting sparsity and zero compression, context specific independence, and search and caching strategies, and also approximate strategies such as sampling, state pruning, assumed density filtering, and variational procedures.

13.4.2 Clique-based Limited Extent Asynchrony

In Section 13.1, we discussed synchronous vs. asynchronous decoding for HMMs. In this section, we discuss how there can be hybrid approaches that straddle the fence between these two extremes. For example, a constraint can be placed on the maximum temporal distance between the end-points of any two partially expanded hypotheses. This, in fact, is something that can easily be guided by a junction tree and its set of maxcliques. Normally in a junction tree, the form of message passing is based either on the Hugin or the Shenoy-Shafer architectures [290]. The Hugin style message passing is summarized in Figure 2.6-II. Each of these approaches, however, assume that the cost of a single message is itself tractable, which is often not the case in applications such as speech recognition due to the very large number of possible random variable values (e.g., the typical vocabulary size of a large vocabulary system might be more than 500,000 which means that a single random variable itself might have that many possible values, and a factor involving k

of those random variables might have $(500,000)^k$ possible non-zero entries¹⁰). Therefore, even individual exact messages in a junction tree might be computationally infeasible and this is where search methods become useful.

We consider the case where a search procedure is used to compute the junction tree message by expanding the clique. The expansion is constrained to occur only within the clique so that the decoding is “synchronous” but at the level of maximal cliques. This means we are not allowed to expand a clique C_{t+1} until clique C_t has been expanded. Within a maximal clique, however, the expansion can occur in any variable order, much like an asynchronous decoding procedure. The cliques in the junction tree may effectively limit the extent of the hypothesis expansion. For example, consider the junction tree for the HMM given in Figure 8.44, where each clique consists only of successive variables $\{Q_t, Q_{t+1}\}$. If we were to perform constrained asynchronous search in such a junction tree, where cliques are expanded one after another but expansion may be asynchronous within a clique, then we have recovered synchronous search in HMMs.

Cliques can consist of any number of random variables, however, and even span over short stretches of more than two time steps [35]. For example, consider the alternative junction tree for an HMM, shown in Figure 13.5, which can be achieved with $S = 2$. In this case, each clique consists of three rather than two random variables. In the triangulated graph corresponding to this junction tree, the triangulation is no longer minimal (meaning that some of the fill-in edges may be removed and the triangulation property still holds). As mentioned above, in some cases non-minimal triangulations can be quite useful [14]. In this case, in fact, the non-minimal triangulation is used to restrict the degree of asynchrony in an HMM expansion. In Figure 13.6, we see an example where the cliques group four successive Markov chain variables together, further relaxing the constraints on the asynchrony (and which can be achieved with $S = 4$). In general, the degree of asynchrony in HMM search can be controlled implicitly by the triangulation of the graph that yields the junction tree. By forming various triangulations (and corresponding junction trees) we can experiment with an even wider variety of different search expansion constraints for an HMM.

This flexibility is greater in the general DGM case, and the chunk skip parameter S (§13.3.2.9) being greater than one adds to this flexibility. We will explain this using Figure 13.33-IV. As given, the standard way to perform such a message would be to execute the following computation

$$\phi(b_4, d_4) = \sum_{b_3, c_3, d_3, a_4, b_4, d_4} \phi(b_3, d_3) \psi(b_3, c_3, d_3) \psi(c_3, d_3, d_4) \psi(b_3, c_3, a_4, b_4, d_4) \quad (13.33)$$

where $\phi(b_3, d_3)$ is the initial incoming and $\phi(b_4, d_4)$ is the final outgoing interface separator factor, and where $\psi(b_3, c_3, d_3)$, $\psi(c_3, d_3, d_4)$, and $\psi(b_3, c_3, a_4, b_4, d_4)$ are functions corresponding to the cliques of the modified chunk. Such an approach can be wasteful, however, since it is oblivious to any sparsity that might exist in the factors and moreover it does not involve any reasonable approximation.

The way a potentially asynchronous search procedure would approach the problem would be to perform a tree-expansion of the variables B(3), C(3), D(3), A(4), B(4), and D(4) with the use of factors $\phi(b_3, d_3)$, $\psi(b_3, c_3, d_3)$, $\psi(c_3, d_3, d_4)$, and $\psi(b_3, c_3, a_4, b_4, d_4)$ as a form of soft constraints to produce partial score hypotheses. This variable expansion becomes a clique’s search tree (it is important to realize that this is entirely distinct from the junction tree of cliques). The traditional notion of synchrony would mean that the variables a_4 , b_4 , and d_4 are not expanded upon until variables b_3 , c_3 , and d_3 are instantiated. A clique-based limited-extent asynchronous approach would allow the variables to expand in any order, and could even be value specific [373, 108, 8, 7, 289, 458, 459, 457, 221, 220, 456]. This means that the order that the variables are expanded might depend on the values of earlier instantiated variables. This can sometimes yield enormous speedups.

¹⁰Note that such a table would never be actually stored in a dense format, rather only values that are used are computed on an as-needed basis. One example of this are backoff-based N-grams and heterogeneous backoff based N-gram models, and other examples are log-linear models where the number of parameters is on the order of a number of features, rather than the number of non-zero values of a CPT

What turns this into a form of search-based message is the following: at the leaf nodes of the clique expansion’s search tree, all of the clique’s variables are instantiated. Once this is done, this instantiation could be inserted into the clique table (assuming the score is non-zero) corresponding to the clique-to-clique message idea mentioned above. When all insertions have occurred, one could sum or compute the maximum entry of the clique.

On the other hand, rather than computing the maximum, or summing the resulting scores in the clique table, the set of values can be used as an index into the outgoing separator, and the resultant score is accumulated into that indexed entry. For example, with variables $b_3, c_3, d_3, a_4, b_4, d_4$ instantiated with score s , we would accumulate into the outgoing separator table, involving the subset of variables b_4, d_4 , as follows:

$$\phi(b_4, d_4) \leftarrow \phi(b_4, d_4) + s \quad (13.34)$$

This could occur at each search-tree leaf node visitation.

As can be imagined, the clique may span multiple time slices and if this happens so may the degree of asynchrony. The clique might span multiple time frames if $S > 1$. Also, if $M > 1$, the interface separator itself may span multiple time slices. Alternatively, the clique might just correspond to a grouping of more than one modified chunk — consider, as an example, a modification of Figure 13.33-IV, where the clique in the junction tree contains twelve rather than six random variables, namely: $\bigcup_{t \in \{3,4\}} \{B(t), C(t), D(t), A(t+1), B(t+1), D(t+1)\}$ achievable with $S = 2$. The limiting case is when the clique expands to the entire length of the segment, and we have recovered the fully asynchronous search procedures mentioned in Section 13.1.

13.4.3 Sparsity Preserving Search-based Message Passing



Figure 13.53: Long burn in example.

As mentioned above, many DGM models involve sparsity in some form, meaning that the corresponding $p(x)$ is not everywhere strictly positive. In the log-linear case, some values of the feature functions may be negative infinity. Indeed, many of the factors of the model might have contain an abundance of zeros and the pattern of these zeros should not be ignored. For example, even in the HMM case, there is sparsity during the burn in that occurs at the beginning of inference. An pictorial example is shown on the left of Figure ??, but in practice, burn-in can last for many frames — sometimes sparsity may persist even through the entire segment (Figure 13.53) which can occur when the segment is short relative to the number of states (so this can occur for long segments when there are many of states). In the HMM case, an alpha recursion more efficient than Equation (??) would be to use the HMM push strategy mentioned in §8.4.4.7. This can be generalized to DGMs as we will see below.

In the Bayesian network case, not only might there be a plethora of zeros at the beginning of a segment, but such sparsity might persist throughout. This means that for some (or all) parent values in a conditional probability table (CPT), there are only a few (or only one if the CPT is deterministic) child values that have non-zero probability (see §8.12.3 for some discussion of deterministic CPTs). Many of the structures used in such temporal models employ sparse factors. For example, the segment model in Figure 13.7 used a number of deterministic conditional probability tables to express counters and fixed length durations. A third reason for sparsity is during approximate inference — under either sampling schemes or “beam pruning” methods mentioned in Section 13.4.5.1 (and other approximation methods as well), the amount of sparsity can significantly increase. Basically, certain combinations of variable values that while not zero-valued have a low score or a low probability are such that their score is “floored” to zero. Hence, we need

inference methods that recognize and efficiently exploit such structures. Not to do so would mean to spend computation on unnecessarily summing together vectors of zeros.

As mentioned in §6.8.2 (recall Figure ?? and see [14]), standard triangulation schemes that span only the space of minimal triangulations are sometimes sub-optimal and therefore it can be beneficial to consider non-minimal triangulations schemes. These are triangulations that cannot be obtained with any elimination order. One approach is to add extra edges, called *ancestral edges*, to the graph before triangulation. These are edges chosen to ensure that nodes which are beneficially related to each other when they exist in a single clique are such that they never end up in separate cliques under a standard triangulation scheme. As an example, while the triangulation of the chunk shown in Figure 13.33-IV can be achieved by an elimination order (i.e., first eliminate node D(4)), in many cases there is no elimination order that can transform the entire segment into one single clique.

Assume we are in the context where a message between junction tree nodes takes the standard marginalize-and-then-multiply form of Hugin propagation (Figure ??). Assume we wish to pass a message between potentials $\psi(X_A)$ and $\psi(X_B)$ where A and B are indexes of variables within cliques (see Figure 2.6). Assuming that $S = A \cap B$ is the index set of the separator variables between cliques. The message from A to B proceeds [207, 208, 219] as follows:

$$\phi(x_S) = \sum_{x_{A \setminus S}} \psi(x_A); \quad \psi^*(x_B) = \psi(x_B) \phi(x_S)$$

This message passing procedure, given in its naïve generality, is amenable neither to the methods in Section 13.4.5.1 nor to the case when there are many sparse dependencies in the models since it suggests a blind summation over all $x_{V \setminus S}$.

It is possible, however, to deploy a form of message passing which can be termed “separator driven” and that is somewhat akin to a hybrid of cutset conditioning [336, 104], the search based approaches mentioned in Section 13.1 and Figure 13.2, and standard message passing approaches summarized in Figure 2.6. As mentioned earlier, when the cost of individual messages becomes prohibitive (as is the case with speech recognition problems), it is necessary to expand the message in a more intelligent way, and search is one way that has been quite successful on large problem instances. This hybrid message passing procedure, moreover, starts from a set of separators, passes through a clique, and results in an outgoing separator, and therefore corresponds to the separator-to-separator messages mentioned in §13.4.1. Therefore, the cliques never need to be stored, only the separator, and will therefore take advantage of the interface separator minimizing algorithm described in Section 13.3.2, thereby directly minimizing memory usage.

Algorithm 17: Separator driven sparse joins in clique expansion. The residual values of each separator, i.e., $x_{S_m \setminus S_{\{m\}}}$, are iterated only if they lead to a non-zero probability. The sparsity within each of these separator tables, moreover, has been augmented with any previous sparsity coming from the interface separator (see Figure 13.54-right).

```

1 for  $x_{S_1} \in \{x_{S_1} : \phi(x_{S_1}) > 0\}$  do
2   for  $x_{S_2 \setminus S_1} \in \{x_{S_2 \setminus S_1} : \phi(x_{S_2 \cap S_1}; x_{S_2 \setminus S_1}) > 0\}$  do
3     ...
4     for  $x_{S_m \setminus S_{\{m\}}} \in \{x_{S_m \setminus S_{\{m\}}} : \phi(x_{S_m \cap S_{\{m\}}}; x_{S_m \setminus S_{\{m\}}}) > 0\}$  do
5       ...
6       expand clique with value  $x_{S_{1:M}}$  and project to outgoing separator.

```

Now, how does this idea help when sparsity abounds? The essential idea is to deploy what could be

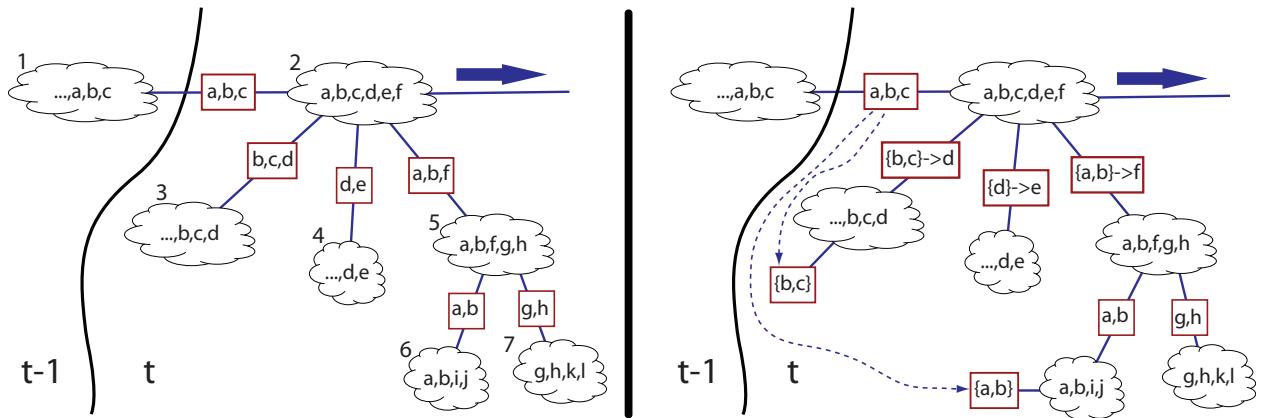


Figure 13.54: Separator driven inference. The left of the graph shows the junction tree along with connecting separators. The goal is to expand factor $f(a, b, c, d, e) = \phi_1(a, b, c)\phi_2(b, c, d)\phi_3(d, e)\phi_4(a, b, f)\psi(a, b, c, d, e, f)$ which depends on factor $g(a, b, f, g, h) = \phi_5(a, b)\phi_6(g, h)\psi(a, b, f, g, h)$ utilizing the fact that any of the factors might be sparse, and moreover that the factor $\phi_1(a, b, c)$ corresponds to the clique in the previous chunk separated by the discovered chunk separator (note other variables might be involved, shown as ... in the figure, but are not notated here for simplicity). Therefore, $\phi_1(a, b, c)$ might possess hard-fought sparsity that was acquired either by expanding earlier portions of the model in time, or by any of the state pruning methods mentioned in this article. The right figure shows how the cliques and separators are organized. The structure on the right shows two things. First, the separators incoming to clique (a, b, c, d, e, f) are organized so that once they are instantiated, only the non-zero entries are iterated, and the non-zero entries of one separator are used to index directly into the entries of the neighboring separator that are guaranteed to have non-zero entries. For example, once (a, b, c) has value, the values (b, c) are used to look up the values of d (if any) that can co-exist with the values (b, c) . Second, the sparsity in the intra-chunk separator (a, b, c) is used to provide real scores to the clique (a, b, c, d, e, f) but only 0/1 sparsity information to other cliques in the chunk (namely (\dots, b, c, d) and (a, b, i, j)).

called separator driven sparse joins. Sparsity that has previously been formed is preserved in the set of separators, and this joint sparsity ensures that it is not the case that variable assignments in the clique are unnecessarily expanded only to be annihilated later by the separators.

Consider a junction tree where a clique is waiting for messages from its neighbors via its incoming separators farther away from the root. Before any message propagation takes place, however, all of the separators that are incoming to a given clique can be ordered and their factors are reorganized into two parts:

1. an accumulated intersection between variables of the separator and any previous separator variables, and
2. a residual set of separator variables

Notationally, assuming that there are M incoming separators with variable index sets, S_1, \dots, S_M , the

separator potential functions (and their associated data structures) are organized as follows:

$$\phi(S_1) \quad (13.35a)$$

$$\phi(S_2 \cap S_1; S_2 \setminus S_1) \quad (13.35b)$$

$$\dots \quad (13.35c)$$

$$\phi(S_m \cap S_{\{m\}}; S_m \setminus S_{\{m\}}) \quad (13.35d)$$

$$\dots \quad (13.35e)$$

where $S_{\{m\}} \triangleq \cup_{i=1}^{m-1} S_i$. The accumulated intersection $S_m \cap S_{\{m\}}$ at position m corresponds to the intersection between the current separator S_m and the accumulated union of the previous separators $S_{\{m\}}$. The residual set of variables corresponds to the new variables $S_m \setminus S_{\{m\}}$ not previously encountered in earlier separators.

Now, what does the organization of the separator factor $\phi(x_{S_m \cap S_{\{m\}}}; x_{S_m \setminus S_{\{m\}}})$ buy us? Given a set of values of the variables $x_{S_{\{m\}}}$ then $\phi(x_{S_m \cap S_{\{m\}}}; x_{S_m \setminus S_{\{m\}}})$ is so organized that we can quickly look up (and iterate over) only those values of $x_{S_m \setminus S_{\{m\}}}$ that, for a given value of $x_{S_{\{m\}}}$ are compatible (i.e., make $\phi(x_{S_m \cap S_{\{m\}}}; x_{S_m \setminus S_{\{m\}}}) > 0$). Since $x_{S_{\{m\}}}$ is already set in in Algorithm 17, any values of $x_{S_m \setminus S_{\{m\}}}$ when combined with $x_{S_{\{m\}}}$ that lead to a zero should never be further considered. But if $x_{S_{\{m\}}}$ is set, it can only effect separator m via the accumulated intersection $x_{S_m \cap S_{\{m\}}}$.

We thus may think of $\phi(x_{S_m \cap S_{\{m\}}}; x_{S_m \setminus S_{\{m\}}})$ as a sparse tensor, where we can quickly look up one subset of dimensions in the tensor via $x_{S_m \cap S_{\{m\}}}$ and iterate only over those remaining subsets of dimensions in the tensor $x_{S_m \setminus S_{\{m\}}}$ that lead to a non-zero $\phi(x_{S_m \cap S_{\{m\}}}; x_{S_m \setminus S_{\{m\}}}) = \phi(x_{S_m}) > 0$. Any other values would be only wasteful to be considered.

The random variable values corresponding to these separators are then iterated (as in Algorithm 17) in this order. The residual random variables are iterated jointly as a group, rather than one at a time, which can lead to a more efficient algorithm especially when sparsity abounds. Jointly iterating random variable values means that values of random variables are considered in blocks as shown in Algorithm 17, rather than individually. The separator scores are multiplied together before “entering” the next separator and before entering the clique (see Figure 13.54 and Algorithm 17 although this algorithm does not show this multiplication for the sake of clarity). Each separator is then checked to determine if it is compatible with the accumulated intersection of previous variable values, and if so all residual variable values are iterated. This works since at the start of iterating each separator, the variables in the accumulated intersection are instantiated (due to the previous separators). Implementation-wise, this means that each separator may consist of a hash-lookup (ensuring that the new separator in variables $S_m \cap S_{\{m\}}$ give non-zero probability), and then an iteration over the non-zero entries of the remaining separator variable values $S_m \setminus S_{\{m\}}$. The tables in the separators themselves may be sparse and optimized so that only the non-zero entries are stored and iterated.

We note that separator driven inference solves the problem sometimes mentioned regarding DBNs — that time-frame t no longer factorizes when we marginalize out frames earlier than t [61, 62, 312] — with separator driven inference we *condition* on the incoming separator which still yields a factorizable chunk, and each factor can be iterated separately. Finding a good frame-separator then means finding one that these separate factors can easily project onto (which adds then another scoring criterion for the separator procedure described in Section 13.3.2).

Another optimization is that separators may also be seen as 0/1-indicators of non-zero variable combinations even if those separators are not an incoming separator for a given clique. For example, in Figure 13.54, we see that the interface separator from the previous chunk could have useful information about sparsity discovered in that previous chunk (say via pruning or via interaction with other variables), and if the cliques (\dots, b, c, d) and (a, b, i, j) were allowed to expand without that knowledge, then this would lead to extraneous

ous computation. Therefore, separators can be used for variable assignment annihilation even if they do not provide anything further to the score of the clique entries other than the values 0 or 1.

The approach outlined above can be combined with any static search method commonly used to successfully solve large CSP (constraint satisfaction) and SAT problems [373, 108, 8, 7, 289, 458, 459, 457, 221, 220, 456]. More importantly, this approach means that beam-pruning for approximate inference can beneficially be applied, as any beam pruning (see Section 13.4.5.1) will reduce the number of entries in all of the separator tables. Since only the separator entries are iterated and conditioned on, any pruning means that search trees that are of exponential size in the number of remaining clique entries will never be expanded.

13.4.4 Exploiting Commonality

As mentioned above, DGM inference involves performing inference over and over again in a left-to-right fashion on a data structure (the chunk) that tends to have very little difference from time step to time step. The basic idea is that an inference engine can predict certain things in the future based on what happened in the past, in a process that might be seen as an embedded online machine learning algorithm that is operating while DGM inference is being performed.

If there are no observed variables, and if all factors in a chunk are time-homogeneous, then we can see that inference within a chunk is identical from one step to the next, and there is great potential for reuse. For example, any of the standard factor-learning procedures can be used over and over again, including those that deduce large factors which indicate zeros, or those that memorize partial computations [373, 108, 8, 7, 289, 458, 459, 457, 221, 220, 456]. In fact, any learnt factor agglomerations can be reused. This implies that there is a much greater potential for reuse than in the static case. In fact, the standard method for flattening the hierarchical structure of a finite-state transducer [308, 307, 306] corresponds to an offline process of learning the structures that lead to non-zero scores in such a hierarchy and then these structures are re-used over and over again in time. Such flattening methods are similar to the factor learning methods, say of [7, 108] but where, in the DGM case, are repeated for each chunk.

Other than learning and re-using factors, the fact that the chunk is repeated leads to many more opportunities for learning and then reusing good strategies for state expansion, some of which we will discuss as part of the next several sections.

Before doing so, however, we note that when observations are present, or when factors are time-inhomogeneous, the potential for reuse over time diminishes. For example, factors learnt in \mathcal{G}_t^c might be rendered useless for \mathcal{G}_{t+1}^c since the observations that caused the zeros at time t are no longer present at time $t + 1$. On the other hand, it is typical in natural and/or real-world signals (such as speech) for the observations to vary slowly enough so that re-use is still possible. Moreover, in cases where we are unable to assume that the observations are the same, then even if factor reuse is diminished, that does not mean that it has no utility at all.

13.4.5 Approximate inference in DGMs

There are many ways to perform approximate inference in DGMs, some of which are described in the sections below.

13.4.5.1 Exploiting sparsity via pruning

Sparsity and beam pruning methods for approximate inference have been extremely successful in the areas of speech recognition and statistical machine translation. The essential idea is that when one is expanding a search, one will annihilate partial hypotheses if their score is looking much worse in some way than the best current hypotheses. In the speech recognition community, such beam pruning methods have been utilized almost exclusively in the context of HMMs, but given the above separator-driven message passing scheme in

a junction tree, such sparsity increasing operations can be used in any graphical model, and are particularly useful for DGM inference.



Figure 13.55: Simple temporal clique chain.

For much of the next discussion, it is sufficient to consider a junction chain. We use the following notation. Let C_t be the clique variables at time t , and let $S_t = C_t \cap C_{t+1}$ be the separator between successive cliques, and X is our set of random variables. Such a clique chain is shown in Figure 13.55. Thus, X_{C_t} and $X_{C_{t+1}}$ are the same random variables but shifted by one section (usually a chunk), and the generic separator-to-separator message is

$$\phi(x_{S_{t+1}}) = \sum_{x_{C_{t+1} \setminus S_{t+1}}} \psi(x_{C_{t+1}}) \phi(x_{S_t}) \quad (13.36)$$

As before, we assume that X_{S_t} is a clique in the chunk, and that the summation in this equation is free to use any method it wants (e.g., sparse junction tree message passing, etc.), but that the naïve summation as given is too computationally expensive. The reason, again, S_t corresponds to a clique is a consequence of Rose's theorem (Theorem 149): even if there is nice initial factorization over S at the beginning of the model, these factorization properties quickly vanish as we eliminate it from the left to the right. We also assume we've spent effort using a minimum vertex-cut procedure to find the best possible S , but due to a large state space, we are still left with a message that is too expensive.

The above described a sparsity preserving message passing routine, so we might be able to “add sparsity” to the model, at the cost of some approximation error. The factor $\phi(x_{S_t})$ might have some inherent sparsity in it already, but we can remove entries that fall below some threshold. That is, the normal state space is the set of tuples $\{x_S : \phi(x_S) > 0\}$, but this can be removed by defining minimum threshold γ and using only those that meet this threshold

$$\{x_S : \phi(x_S) > \gamma\}. \quad (13.37)$$

Such methods are collectively called “beam” pruning, where a narrow beam implies a great deal of computational savings occurs at the expense of large inaccuracy due to approximation, and a wide beam corresponds to a modest amount of computational savings and also a small degree of inaccuracy.

Second, once the clique has been expanded, we can remove entries of the clique, thereby leading to a sparse clique, which can be exploited by the aforementioned separator-driven message passing. This is similar to compression methods that has been employed in Bayesian network systems [105], but is different in that decisions are made dynamically.

In static localized pruning, pruning decisions are made locally based on information only within a current section (or maxclique) and its incident separators. There are two forms.

First, we view the process of listing the clique entries within a clique as a standard tree search procedure, and then portions of the search space are pruned off based on a given partial path looking unfavorable. This is similar to standard approaches for beam-pruning search in static models, and is widely used for SAT/CSP solvers and for search procedures in AI solvers. A key difference, however, is that here, the search is limited to be within a section, and possibly even within a maxclique within a section. Also, here search uses scores that involve the internal clique scores along with the temporal scores up to the current section.

In this approach, anything outside the beam is removed. In the context of the above threshold method, there are a number of ways to determine γ . For example, the beam can be defined as a fixed (log) ratio away

from the current maximum scoring clique hypothesis as follows:

$$\gamma = \left(\max_{x_S} \phi(x_S) \right) / b \quad (13.38)$$

where b is a constant known as the “beam width”, set by the user and then fixed for all time. Alternatively, there can be a time-dependent beam width b_t . This strategy works well if the local distributions do not tend to experience phases of high concentration. In other words, if we were to consider a distribution formed as

$$p(x_S) = \frac{\phi(x_S)}{\sum_{x'_S} \phi(x'_S)} \quad (13.39)$$

and then compute its entropy $H(p(x_S))$, high-entropy cases would tend to work well with this method, since the scores are fairly spread out. If the distribution becomes very concentrated near the top, however, then this strategy fails, in that it stops pruning entirely (almost all scores are close to the maximum score).

Another approach, state-space based pruning, is to keep the state space no larger than k , meaning a fixed value k is chosen and only the top scoring k clique entries are preserved. We order the values so that

$$\phi(x_S^{(1)}) \geq \phi(x_S^{(2)}) \geq \dots \geq \phi(x_S^{(N)}) \quad (13.40)$$

Then, then state space becomes

$$\left\{ x_S : \phi(x_S) > \phi(x_S^{(k)}) \right\} \quad (13.41)$$

Note that we only keep the top k , even if there ties at value $\phi(x_S^{(k)})$, this is to keep the number of entries bound by k . This can useful as it bounds the computation based on k , regardless of the cost of doing so. This approach, moreover, is immune to the concentration of the scores. Moreover, in practice, using a variant of quicksort we select the top k in $O(n)$ time for any k where n is the number of entries, although we don't get the total order in this case (we can get a total order of the top k entries in $O(n + k \log k)$ time). This is important as we do not want the time it takes to compute the pruning to dwarf the amount of computational savings we get by pruning. On the other hand, if the scores are very spread out (or worse, if the scores are bi-modal with two groups of highly concentrated scores, one very high and one very low), this approach might be less effective if k is not set exactly right.

A third approach keeps a fixed percentage of the clique entries regardless of their score. Here, we keep the top, say p percent of the entries. That is, we again order the entries as in Equation (13.40), and choose k in Equation (13.41) such that $100 \times p = k/n$.

A fourth approach, called beam mass percentage pruning, is such that a fixed percentage of the total clique mass is preserved [332], where after sorting the entries, the top k is chosen where k is the smallest number that meets a given mass percentage. Let κ be the fraction of mass to retain. We then order the values so that

$$\phi(x_S^{(1)}) \geq \phi(x_S^{(2)}) \geq \dots \geq \phi(x_S^{(N)}) \quad (13.42)$$

We next find the smallest k such that

$$\frac{\sum_{i=1}^k \phi(x_S^{(i)})}{\sum_{i=1}^N \phi(x_S^{(i)})} \geq \kappa \quad (13.43)$$

It is moreover often useful to place a lower bound k_{\min} on the state space (i.e., if one state already is above threshold), and then take $k = \max(k, k_{\min})$. Then, the state space becomes

$$\left\{ x_S : \phi(x_S) > \phi(x_S^{(k)}) \right\} \quad (13.44)$$

This can be useful to ensure local cost of pruning is not too bad.

A fifth approach, called diversity pruning, uses a diversity criterion that ensures that entries are not removed if it reduces the diversity of the clique by a certain amount, where diversity is determined not by the scores but by the composition of the random variable values making up the clique table entries. For example, one might perform pruning in clusters of clique entries, where each cluster consists of relatively homogeneous sets of clique table entries. The justification for this is that a table entry should be preserved even if it has a low score if its composition is very different than the composition of other table entries. This is to ensure that the diverse composition of the table is preserved in order to account for any events that, while perhaps seeming improbable locally in time, might lead to the only way of avoiding a zero-clique error later in time. Indeed, these pruning methods, along with sparsity preservation, enable massive models to be utilized [269].

In general, it is not always guaranteed that one pruning method is preferred over another method, although a good rule-of-thumb is that the state-space pruning works well in general and also puts a bound on the computation.

In general, we are approximating the DGM by using a (sparser) potential function $\tilde{\phi}(x_S)$ in place of $\phi(x_S)$. With our sparse message passing scheme described above, this can significantly reduce the computation.

We should note that these pruning methods can also be applied to the cliques as well as the separators, including within the chunk to produce $\tilde{\psi}(x_{C_{t+1}})$ in place of $\psi(x_{C_{t+1}})$. There may in fact even be an advantage to doing so as while the maxcliques are typically larger than the separators, since more variables are involved, the pruning strategies might be more nuanced than when things are projected into a separator and the pruning might be more of a blunt hammer. On the other hand, purely separator-to-separator messages are amenable only to pruning at the separator level (as is typical with an HMM system).

Pruning can fail if pruning is too aggressive. The way this is typically manifest is that inference gets to the point where no remaining entry in a clique has non-zero probability. For example, suppose we have a factor at time t of the form $p(o_t|a_t, b_t)$ and the only way of explaining the value of $\bar{o}_t = 3$ is with $a_t = 2, b_t = 1$. If, based on an earlier pruning decision, there is no table entry having value $a_t = 2, b_t = 1$, then there is no way to explain the observation $\bar{o}_t = 3$ and the entire table will contain zeros (or will be empty if zero scores are not stored). This is a classic zero-clique problem, and the only way to avoid it is to perform less aggressive pruning.

On the other hand, different forms of backtracking can recover from such errors. For example, if we run into a brick wall zero-clique error at time t , we could backtrack to $t - 1$, expand the beam, and then move forward again to time t and hope that the zero-clique error has vanished. If it persists, we can backtrack to time $t - 2$, and expand the beam at both time $t - 2$ and $t - 1$ and then move back to t . Each time we backtrack we would expand the beam relative to what it has been set to previously (by some factor say). This kind of backtracking can repeat until we have identified the “cause” of the pruning error. The difficulty in this approach is we do not know how far back to go to identify a problem, nor do we know how much to expand the beam in order to include the table entry that leads to removing the zero-clique error. An alternate strategy could be to backtrack using the pattern $t - 1, t, t - 2, t, t - 4, t, t - 8, t, t - 16, t$ and so on, a form of exponential backoff. The key point is we don’t know where precisely the pruning decision that caused problems is, so we want to get to that point quickly to increase the beam. This would quickly get to the point where the pruning decision lead us to the zero-clique error, although the problem of how much to expand the beam persists. One could use the same approach, and at each beam expansion, increase the beam exponentially fast. The idea tradeoff here, again, is that the amount of time saved by performing pruning and then backtracking and expanding the state space should not be more than the cost of not doing pruning in the first place.

Pruning does not enable factorization - all interactions between the variables are potentially retained in any pruned table. That is, with pruning, there is no $S_1, S_2 \subset S$ such that $S_1 \cup S_2 = S, S_1 \cap S_2 = \emptyset$ where

$\tilde{\phi}(x_S) = \tilde{\phi}(x_{S_1})\tilde{\phi}(x_{S_2})$. Therefore, loss of structural properties of the factored state (Rose's entanglement theorem) space is still present. On the other hand, this might not be a disadvantage for the same reason that big cliques can in fact facilitate pruning. As the clique size increases, the approximation made by pruning can only get better. The reason is that with larger cliques, more information is used to make a pruning decision, so pruning decisions are made more accurately. In the limit, as all variables are in one big clique, pruning is exact (in that it retains only the best set of global hypotheses). On the other hand, there is a tradeoff — bigger cliques are more costly to compute, and any benefit obtained by pruning based on a larger clique might be offset by producing that large clique in the first place. In some sense, there is a tradeoff between more vs. less aggressive pruning, and more vs. less aggressive factorization. Factorization, moreover, can span the space of exact factorization (i.e., if we factor with respect to a graphical model) and approximate factorization (once we, for the purposes of fast inference, start making additional factorization assumptions beyond what is given by the graphical model). Indeed, the graphical model itself might only be an approximation of some true natural process. Where we should be within this tradeoff can only be best answered in the context of a given application.

In the context of DGMs, when pruning, it is beneficial to have the option to make such tradeoffs. For example, the chunk-skip parameter S allows a modified section to cover multiple original sections, and hence allows larger cliques than what ordinarily would be with only $S = 1$, and thus allows more flexible pruning decisions.

dynamic pruning: With dynamic pruning, the learnt temporal properties of a section expansion can be re-used for later section expansions. These methods can be thought of an approximate variant of the techniques in §13.4.4. There must be at least some partial time-homogeneity in the dynamics. Let us say that C_t is the state space at time t . We wish to be able to use information about how C_τ for $\tau < t$ was expanded to help in making decisions about how C_t should be expanded. It is crucial, however, that any learning and prediction methods not be more expensive than the actual expansion itself. One approach, therefore, is to use a form of adaptive filter that learns to predict the maximum score within each chunk. Given an estimate of the maximum score at time t , more informed beam pruning methods and/or A* heuristics can be developed.

13.4.5.2 Sampling approaches

There is a long history of Monte Carlo approaches in dynamic models, often called sequential Monte Carlo or particle-filter techniques where distributions are represented by samples. There are many ways to generate such samples, and one way is importance sampling. The essential idea is to utilize a proposal distribution that generates samples relatively inexpensively, and these samples are then re-weighted by the true distribution (evaluating observed variables is cheaper than generating samples from it) and there are often methods that do not even require direct evaluation of the true distribution [172, 127, 364]. In the dynamic case, this often takes the form of particles (values of a set of random variables in a chunk) that are retained or only slightly modified in stepping from time step to time step, a process which works only because each chunk contains a set of random variables that is identical to the previous chunk's set of random variables, except for a shift in the time index variable. The process of resampling, moreover, ensures that particles do not become degenerate, meaning that there should not be an overabundance of representation of a probability distribution where that distribution has low probability. This process, moreover, can be combined with Rao-Blackwellization (a form of marginalization) when the model is tractable for further benefit[313]. Various forms of particle filters have been quite successful in robotic applications [309].

We note that this approach is particularly useful when the state space is inherently continuous, since in such case the resolution of any sampling, as represented by a set of particles, can adjust over time so that the particles are always densest wherever the greatest concentrated of probability lies. Since probability concentration can evolve in a dynamic model, any *a priori* specified discretization would be much less

efficient. As long as the distribution remains within the range of numerical precision, particles can also become more concentrated to accurately reflect the distribution. For discrete state spaces, however, the aforementioned pruning approaches already achieve this effect since at any given time, the entire state space is not represented, rather only a subset of states exist corresponding to those that currently look most promising. As the distribution over a discrete set becomes more concentrated, so would any post-pruning remaining table entries.

One key difference between pruning and sampling, however, is that sampling can represent the entire distribution, while many pruning methods (such as state-space or max-score based pruning) represents only a truncated distribution — we are keeping only the top scoring entries. Sampling methods retain particles with some probability even within regions of low score. Note that some of the other pruning methods, such as diversity pruning, try to rectify this by ensuring representation even within regions of low-probability.

13.4.5.3 Factored interfaces, and assumed-density filtering approaches

factor based approaches: Another approach to approximate inference in dynamic models is factorization. This was discussed in §13.3.6. Here, additional factorization properties are made to break the consequence of Rose’s theorem, that variables in a structured state space are eventually coupled if there is an active path between them consisting only of variables in the past. There are two basic ways to utilize such factorization methods. The first assumes that the structured state space is factored over all time, so that a variational approximation can be used. A better approach is one of [63, 62] where a projection down to a factorized state space is done every so often, even every frame. The key benefit of this latter case is that the temporal process can reduce the error in such a factorization assumption, in that the variables once again become coupled at each step. The method proceeds by iteratively projecting down to a factored space, and then continuing with forward inference. It can be shown that in some cases the accumulated error is actually bounded and, perhaps surprisingly, independent of the length of the sequence [63, 62]. It should be noted that the chunk-skip parameter $S > 1$ can also control how often this projection takes place, which can allow for further time to “recover” from this factored approximation.

Note that most of the approaches in speech recognition and language processing in hierarchical HMM systems have for the most part avoided such factorizations and deployed state pruning on the unfactorized model (as described above). Note that the factorization and sampling approaches may be combined [309].

13.4.5.4 Multi-pass and Coarse-to-fine approaches

Multi-pass strategies: All of the inference methods we have thus far discussed thus far correspond to single-pass algorithms, where one scans from left-to-right across the model, without any knowledge of the future. In many applications, such as online (Kalman-style) filtering, sensor monitoring, or any real-time inference, this is the only choice, since one is deducing a distribution over the state space at time t and perhaps making critical decisions based on this distribution (robot navigation is one example). In such cases, we cannot use the future. In other applications, however, one might be able to obtain information about the future, and there are some crucial advantages to doing this when it is an option. One possibility is a multi-pass strategy, which starts by making inferences using one pass of a simple model, and then refines the inferences with secondary passes using richer, but more expensive operations. Multi-pass strategies allow estimates of future behaviour, based on simple models, to guide inference in the more complicated model of interest.

We attempt to describe such advantages with an analogy. Suppose we wish to use a web search engine to find images containing a particular object (say a tree, or a car). In presenting the results to a user, the engine could on the one hand present a small patch of each image at its highest available resolution (say portion of a wheel of a car or a bit of green leaf of a tree). This would be suboptimal, however, as the chosen segment might exclude the object we are interested in, and worse would be confusable with objects we are

not interested in. On the other hand, the engine could present us with a list of thumbnails, a coarse-grained low-resolution versions of each image. These thumbnails do not show all detail of every but do provide a quick global summary of the entire image, thereby allowing us to zoom in only on those images that appear to contain the objects of interest. The second scheme is a more efficient way to identify images containing the desired object.

This is the principle behind multi-pass coarse-to-fine inference strategies in DGMs. When we perform a left-to-right pass in a DGM, any approximation made at time t (either via pruning or sampling) might render the search ill-prepared for an event that happens down-stream. For example, a pruning away of a hypothesis that looks poor at the current time might be along the only path that leads to the only possibility of scoring certain later observations with non-zero probability, and it might be the only way to avoid a zero-clique error. It is impossible to predict the future without making (perhaps strong and wrong) assumptions about the nature of the time-signals that are being modeled.

An alternative multi-pass approach performs multiple complete passes of inference on the time signal, each one having an influence on the next and each one possessing more inherent temporal resolution and power to identify the goal of inference. This has the effect that any approximations made at time t can make use of information not only from the past, but also from the future, albeit at a coarser granularity. This can be enormously beneficial even if this information from the future only coarse-grained in nature. In fact, the approach is widely used in speech recognition, where an initial simple HMM-based system is used to produce a data structure, typically a trellis as in Figure 8.28, that restricts the state space of a more complex second pass system that, otherwise, would be computationally prohibitive. There is at least some assurance, however, that these structures enumerate hypothesizes that are globally good. The simplest approach is for the first pass system to produce an N -best list of hypotheses which are then re-ranked by later pass systems. At the time of this writing, multi-pass speech recognition systems are state-of-the-art — there is no single pass system that can do as well in error performance as a multi-pass system.

Note that there are many ways to express the reduced state space in a dynamic model. The field of speech recognition use trellises (as described in §8.4.5), which is a state-transition graph that lists the allowable trajectories through the search space. Sometimes the trellises are quite imprecise, for example “confusion networks” are trellises that are pinched at particular time points in a way that at these pinch points, there is only one possible hypothesis path being represented. Such trellises, nevertheless, are often sufficient to limit the search space sufficiently so that more powerful and accurate second-pass systems can perform quite well. Other structures, however, could equally well be used such as binary [2] or algebraic [9] decision diagrams, or case-factor diagrams [296]. These are all essentially data structures that allow for efficient procedures to produce restricted decoding algorithms in probabilistic models, and could easily be extended to temporal models.

13.4.5.5 Static inference

As each of the sections is essentially a static graphical model, any technique that results in a marginal (or an estimate thereof) distribution over the right interface separator the chunk can be used. In the past there have been variational, mean field, expectation propagation, LP-relaxation, and polyhedral approaches all of which can be applied and most of which are nicely summarized in [434].

It’s worth pointing out that mean field and expectation-propagation methods have been framed as message passing algorithms so it’s not surprising that these ideas can be applied here as well. In theory, the LP-relaxations can also be used. The problem is that in practice they are sometimes impractical due to poor conditioning, or a quadratic dependence on the number of states a variable can take.

Of course, many of the above methods are not incompatible and there is no reason that they could not be usefully utilized simultaneously in the same system.

13.5 Mixed max/sum decoding: semi-ring mixing

It is often of interest to compute not only the Viterbi assignment of a set of variables, but moreover to compute the Viterbi assignment when some of the variables have been summed over. This in the past has been called “maximum likelihood decoding”, or “maximum a-posteriori hypothesis” (or MAP) decoding, which we define fully below.

It is well known that MAP decoding in general is intractable. In this section, we show that in some cases, MAP is not intractable and in fact is no more expensive than decoding using a given triangulation of a graphical model. At the end of this section, we will offer a form of MAP decoding approximation based on a two-pass approach, where posterior probabilities and a marginal projection is used in the first pass, and where a second pass uses this marginal projection in a way that is easy to compute.

First a bit of notation. We have a set of N random variables X_1, \dots, X_N and partition the index set $[N] = 1 \dots N$ into $E \cup H \cup F = \{1, \dots, N\}$ (i.e., E, F and H are complete and mutually disjoint). We can reason about subsets of random variables X_E , X_F , and X_H . The set E constitute the evidence (or observed) variables, H are hidden variables we wish to integrate out, and F indexes the variables that we wish to query.

We have a joint probability $p(x_{1:N}) = p(x_E, x_F, x_H)$. Since E are evidence nodes (or observed), we write this as \bar{x}_E and $p(\bar{x}_E, x_F, x_H)$ to indicate that \bar{x}_E are constant/fixed.

The goal of MAP decoding is to find the most probable or likely assignments to the variables X_F . I.e., the ultimate goal is to form:

$$p(\bar{x}_E, x_F) = \sum_{x_H} p(\bar{x}_E, x_F, x_H) \quad (13.45)$$

and then compute

$$x_F^* = \operatorname{argmax}_{x_F} p(\bar{x}_E, x_F) \quad (13.46)$$

This is really a mix of semi-rings (see §8.4.4.5), namely the sum-product semiring over the variables x_H and the max-product semiring over the variables x_F .

The Viterbi approximation is to use only the max-product semiring over all variables, and to compute instead the following quantity:

$$(\hat{x}_H^*, \hat{x}_F^*) = \operatorname{argmax}_{x_F, x_H} p(\bar{x}_E, x_H, x_F) \quad (13.47)$$

and then we take \hat{x}_F^* to be an approximation to x_F^* . It is well known that \hat{x}_F^* may not be a good approximation to x_F^* . Also, we call the following quantity the “Viterbi score”

$$p(\bar{x}_E, x_H^*, x_F^*) \quad (13.48)$$

Viterbi decoding is in general much easier to do than MAP decoding, and in fact Viterbi decoding is a specific instance of MAP decoding (MAP when $H = \emptyset$ is Viterbi).

The first question we address in this section is the following: is MAP decoding with a non-empty H (an in general semiring mixing) always hard or only sometimes?

What we attempt to answer here is to say that there are cases that MAP decoding is no more difficult than the complexity implied by a given triangulation of the model. If you have a triangulation you are happy with in terms of its complexity, and your MAP decoding obeys certain constraints with respect to that triangulation, then MAP decoding is no more difficult (in general complexity) than Viterbi decoding.

First, recall from §6.8 that a triangulation of a graph is a clustering of the variables into cliques such that the set of clusters obey the running intersection property (rip). Specifically, we have a set of cliques

$\{C_1, C_2, \dots, C_M\} = \mathcal{C}$ where $C_i \subseteq \{1, \dots, N\}$, and where there is an ordering σ of the cliques so that rip holds, specifically, for each i there exists a $j < i$ such that:

$$(C_{\sigma_1} \cup C_{\sigma_2} \cup \dots \cup C_{\sigma_{i-1}}) \cap C_{\sigma_i} = C_{\sigma_j} \cap C_{\sigma_i}$$

This also says how to organize the cliques into a junction tree, namely when we wish to attach clique σ_i we attach it to clique σ_j creating a separator with nodes $C_{\sigma_i} \cap C_{\sigma_j}$.

Given such a triangulation/junction-tree/rip-obeying-clique-set, we can write the joint distribution as follows:

$$p(x_{1:N}) = \frac{\prod_{C \in \mathcal{C}} p(x_C)}{\prod_{S \in \mathcal{S}} p(x_S)}$$

where \mathcal{S} is the set of separators in the corresponding junction tree. Any distribution along with a given triangulation can be exactly written in this way in terms of clique and separator marginals. In other words, each of $p(x_C)$ and $p(x_S)$ is itself a proper probability distribution. Moreover, in complexity given by the corresponding triangulation, it is possible to manipulate the junction tree data structure such that each clique and separator is set to that marginal (this is what GMTK does in fact). Note that what we mean by a marginal is that for a given clique C , we have

$$p(x_C) = \sum_{x_{\{1, \dots, N\} \setminus C}} p(x_1, \dots, x_N) \quad (13.49)$$

and where the same thing is true for separator marginals $p(x_S)$.

Given this data structure, then, the question becomes when is it possible to do MAP decoding in complexity no worse than that implied by the junction tree.

Consider the MAP marginalization step mentioned above:

$$p(\bar{x}_E, x_F) = \sum_{x_H} \frac{\prod_{C \in \mathcal{C}} p(x_C)}{\prod_{S \in \mathcal{S}} p(x_S)}$$

Suppose it is the case that for each $i \in H$, there exists only one $j \in \{1, \dots, M\}$ where $i \in C_j$. In other words, each hidden variable lives in one and only one clique. In such case, we see that none of the hidden variables live in any of the separators, and it is possible to form the above computation in the following way:

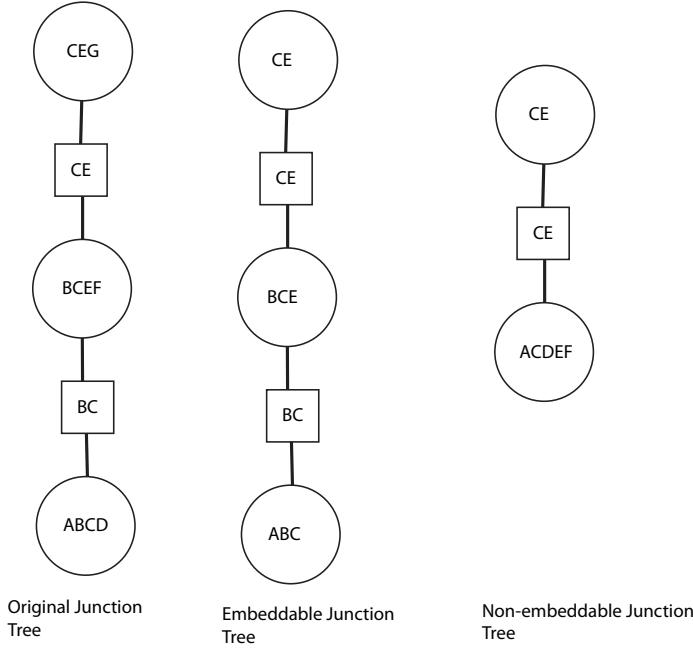
$$\frac{\prod_{C \in \mathcal{C}} \sum_{x_{i(C)}} p(x_C)}{\prod_{S \in \mathcal{S}} p(x_S)}$$

where we use the notation $i(C)$ to indicate the set of $i \in H$ that are uniquely contained in clique C . We then get a new representation of the form

$$p(\bar{x}_E, x_F) = \frac{\prod_{C'} p(x_{C'})}{\prod_{S \in \mathcal{S}} p(x_S)}$$

where each C' is a subset of the corresponding original C , so $C' = C \setminus i(C)$, and the separators are unchanged. Doing this computation can be done with no more complexity than expressed by the original set of cliques. Therefore, MAP decoding can be easily done whenever it is the case that the variables that need to be marginalized away H are such that they each individually live only in one clique of a current triangulation.

As an example, consider the following junction tree on the left:



This figure (on left) corresponds to the distribution

$$p(a, b, c, d, e, f, g) = \frac{p(a, b, c, d)p(b, c, e, f)p(c, e, g)}{p(b, c)p(c, e)}$$

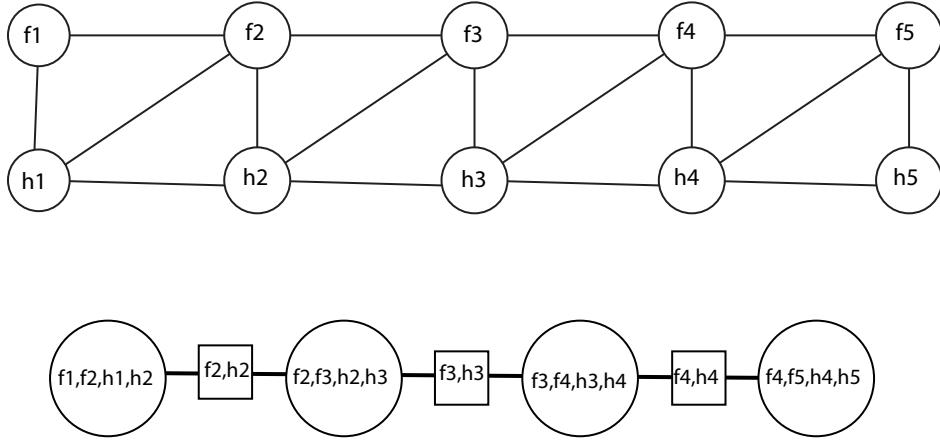
If we were to marginalize out the variables d , f , and g , we would get a new junction tree that can be “embedded” into the original junction tree, and therefore has the same or less complexity. What we mean by one JT being able to be embedded into another is that each clique in the first has a corresponding clique in the other which is a superset of its variables, and the same for the first JT’s separators. The “embeddable junction tree” is shown in the middle of the figure.

When MAP decoding starts to become more difficult is that in order to have a triangulation where the above is true, the complexity of triangulation could increase arbitrarily. I.e., we might need to make the cliques large enough so that the variables we are interested in marginalizing over each live in only one clique.

A similar strategy would be to take the original junction tree, and merge cliques together until the variable that needs to be marginalized away lives in only one clique. This would then end up joining all cliques that contain a given variable (by rip, the cliques constitute a sub-tree within the junction tree). All of the cliques within that sub-tree would then be merged forming a super-clique which then would be used to marginalize. This new junction tree is no longer embeddable into the original junction tree, since it has larger cliques.

For example, suppose that we wish in the above distribution to marginalize over the variable b first. We could merge the cliques BCEF and ABCD forming super-clique ABCDEF from which it would be possible to marginalize away b , but note that the complexity has (potentially significantly) increased since the maximum clique size has gone from 4 to 6.

The next question is, in the DBN case, what scenario are we in? Lets suppose that we have the DBN fragment as follows: It is a two-layer DBN with two Markov chains f_1, f_2, \dots and h_1, h_2, \dots and where we wish to compute MAP over f marginalizing away h . The figure has \bar{x} removed since they are assumed to be only observations and don’t effect the complexity of the computation in this case (i.e., we assume a single child per frame \bar{x}_t , so that h and x form an HMM).



The bottom of the figure shows one possible triangulation of the above graph where we have a chain of cliques and separators. Note that this is of course not an optimal junction tree but it is useful for descriptive purposes.

We see that the variables we wish to marginalize away, namely h_1, h_2, \dots are not in only one clique (except for h_1). If we were to start marginalizing, we'd get the following stages:

- Marginalizing out h_1 is not a problem, since we just get the clique f_1, f_2, h_2 which comes from the added undirected edge between f_1 and h_2 .
- Next, marginalizing out h_2 would merge the cliques f_1, f_2, h_2 and f_2, f_3, h_2, h_3 producing a joint clique f_1, f_2, f_3, h_2, h_3 from which we can marginalize away h_2 producing resultant clique f_1, f_2, f_3, h_3 .
- Next, marginalizing h_3 will merge cliques f_1, f_2, f_3, h_3 and f_3, f_4, h_3, h_4 to produce $f_1, f_2, f_3, f_4, h_3, h_4$ from which when we marginalize away h_3 , which would yield f_1, f_2, f_3, f_4, h_4 .
- Continuing on in this pattern, we see that we would end up producing a clique that contains all of f_1, f_2, \dots and finding the max assignment over that would obviously be intractable.

The above procedure can be seen as a variable elimination process, where we eliminate (marginalize out) the h_i variables in order to form a model on which we can hopefully perform MAP inference on. From the graph above, we can see that there is no ordering of the h_i variables that does not couple together the f_i variables into one large clique. This follows from Rose's entanglement theorem (Theorem 149, recall, if there is a path between two variables that consists of previously eliminated nodes, then they will be coupled). Since in this case, all of the f variables are connected by a path consisting of previously eliminated variables (the h variables), all of the f variables are coupled together into one enormous clique.

13.5.1 A class of approximation algorithms

We consider in this section a possible approach to approximating this problem. We start with each clique marginal $p(f_i, f_{i+1}, h_i, h_{i+1}, \bar{x}_E)$ and marginalize out the variables to give $p(f_i, \bar{x}_E)$ (we've added back in the mention of the evidence \bar{x}_E for clarity). We then add back in what we might call “grammar constraints”, which are factors that place either soft or hard constraints on any Viterbi path over the f variables. Lets call this $p(f_i, f_{i+1})$, and then form the product:

$$\prod_i p(f_i, \bar{x}_E) p(f_i, f_{i+1}).$$

Of course this is not a normalized probability distribution, and to make it so we would need to divide by $\sum_{f_{1:T}} p(f_i, \bar{x}_E) p(f_i, f_{i+1})$. Our goal, however, is to find the most probable assignment, i.e.,:

$$\hat{f}_{1:N}^* = \operatorname{argmax}_{f_{1:N}} \prod_i p(f_i, \bar{x}_E) p(f_i, f_{i+1})$$

and for this, anything proportional to $\prod_i p(f_i, \bar{x}_E) p(f_i, f_{i+1})$ suffices. This is a quantity that is efficiently computable by a second stage decoding run of GMTK by using the initial state outputs $p(f_i, \bar{x}_E)$ or the posteriors $p(f_i | \bar{x}_E)$ as virtual-evidence inputs to the second stage.

There are several points that can be made about this process.

First, based on the above discussion, this clearly is not always going to be an exact MAP. In fact, the approximation can become arbitrarily bad. How good an approximation this approach is will depend on certain regularity assumptions made about the distribution and its projections down to posterior factors. On the other hand, we see that this approximation is quite computationally tractable, costing no more than the original Viterbi approximation.

Second, the above suggests a series of approximation strategies, where rather than marginalizing out over all but one variable, we marginalize out over all but K variables. For example, with $K = 2$, the strategy would become: first marginalize out over all but f_i, f_{i+1} to produce $p(f_i, f_{i+1}, \bar{x}_E)$. Second, we take these clique marginals and form the following:

$$\prod_i p(f_i, f_{i+1}, \bar{x}_E)$$

In this form, as long as the grammar constraints are possible to express over only two frames, then there is no need to include an additional factor that imposes them since anything violating those constraints would still be contained in $p(f_i, f_{i+1}, \bar{x}_E)$. If the grammar constraints are over longer time spans (such as various upper bounds on durations) they could be multiplied in as well.

13.6 Finite Precision Numerics and DGMs

Most of the material on how to deal with inference with very long sequences, and the very large dynamic range one needs with the numerics, is given in §8.4.11.12. In that section, two approaches are mentioned: 1) use of log-arithmetic, where only an exponent of a number is represented rather than its base or mantissa; or 2) where a number is renormalized to be in a safe numerical range at each time frame.

13.7 Discussion

In sum, a variety of methods have been described in the previous pages that can be used to make inference in dynamic graphical models both fast and usable for real-world problems. We described how a DGM can be spliced into a repeatable optimal segment in such a way that one can use many of the methods in static graphical models to solve dynamic graphical models. As can be seen, there is quite some flexibility in producing such a final algorithm, starting from how best to choose the interface separators, how to triangulate the resulting modified chunk, how to organize the separators in the separator driven inference procedure and how to organize the search within the clique, the temporal length limit of each clique which determines the extent of asynchrony, and the resulting approximation methods that become necessary when the state space gets large. The concept of search has been key to the production of messages — while search is not the only means to obtain approximate inference, it works quite well in practice on large real-world problems.

Part VI

GMTK Toolkit

Chapter 14

Toolkit Overview

The Graphical Models Toolkit (GMTK) is a software system for developing graphical-model based time series systems. This set of chapters provides a detailed description of GMTK’s features, including its textual language for specifying graphical structures and probability distributions, and how to specify the parameters that must accompany these structures, and how to train and use them. The graphs themselves may represent everything from types of N-gram language models down to Gaussian components or deep MLPs, and the probabilistic inference mechanism supports first-pass decoding for any of these cases.

14.1 GMTK’s “Hello World”

Perhaps the best way to begin to use a toolkit is to see a simple example. It has become common, since the introduction of the C language, to introduce programming languages by showing an example how to print the string “Hello World!” While GMTK is not a general purpose Turing-complete programming language, there are many things that one can do with it, including printing a string (and in fact there are many ways to print such a string). This section shows one simple way to do so.

There are several components that must be specified when producing input for GMTK, and this includes:

1. A structure file, which gives the graphical model (vertices and edges) in the DGM template.
2. Parameter files, which give the parameters associated with the factors or CPTs associated with the structure file. Some structural aspects are also sometimes specified in a parameter file (e.g., dlinks)
3. Data or observation files. An observation file consists of a list of segments, and each segment is a sequence of vectors (which may be integer or real valued or both).
4. GMTK programs and command line parameters. There are several different GMTK programs each having many different options that can be specified on the command line.

For our hello-world example, we have a choice of many different possible structures.

14.1.1 Single variable “Hello World”

We’ll start with the simplest, a single binary random variable observed to be the value 1. This can be specified in the following structure file.

```
GRAPHICAL_MODEL hw1

frame: 0 {
    variable: hw_hidden {
        type: discrete hidden cardinality 2;
        conditionalparents: nil using DenseCPT("concentrate_at_one");
        symboltable: collection("print_strings");
    }
}
chunk 0:0
```

The structure file specifies only one random variable called `hw_hidden`, which is discrete, hidden, and has a domain size (or cardinality) of two, and hence is binary (all discrete random variables in GMTK range in value from zero up to the cardinality minus one). The variable has no parents (`nil`), uses a dense CPT called `concentrate_at_one`, and has a symbol table called `print_strings` which is a way of associating strings to non-negative integer random variable values. As all GMTK structure files specify a DGM template (§8.11.1), we obtain a partition using a chunk specification `0 : 0`, meaning all frames within this range. We assume that this is kept in a file called `hw1.str`.

So much for the structure. The parameter file is called `params.master` and has the following contents:

```
% this is a comment, as is anything after the % character.

% Dense CPTs
DENSE_CPT_IN_FILE
inline % the CPT is inline rather than comes from a file.
1      % The number of Dense CPTs that follow.
0      % The number of the current CPT (zero indicates this is first).
concentrate_at_one % the name of current CPT.
0      % number of parents (zero means no parents).
2      % num of values (two, so this is for a binary variable).
0 1    % P(Z=1) = 1 = 1 - P(Z=0), where Z represents some variable name.

% Collection of names, used in this case for a symbol table
NAME_COLLECTION_IN_FILE
inline % the collection is inline rather than comes from a file.
1      % the number of collections that follow, one in this case.
0      % The number of the current collection (zero indicates first).
print_strings % The name of the collection of strings.
```

```
2 % collection length, to be followed by two strings.
Goodbye_World % the first string.
Hello_World   % the second string.
```

This file contains two objects. The first is a “dense CPT”, which is a conditional probability table that is dense (not sparse). The comments explain what the content means. The second object gives the symbol table, and it defines two strings `Goodbye_World` and `Hello_World` (note strings can not contain any space characters).

A third component must be specified, namely the observation files. A simple way to do this is to have a list of ASCII files, where each ASCII file corresponds to a GMTK segment. Hence the number of ASCII files in the file list corresponds to the number of segments, and each ASCII file determines the segment length. For our example, we will have a list of length one, and the contents of this one ASCII file will be a single observation (the integer one).

The ASCII file list is a file called `ascii_file_list.txt` and has the following form:

```
ascii_file_1.txt
```

The single frame segment then is in a file called `ascii_file_1.txt` and has the contents:

```
1
```

To run the program, we must first triangulate the structure file and this is done with the `gmtkTriangulate` command as follows:

```
gmtkTriangulate -str hw1.str
```

This will create a file called `hw1.str.trifile` — note that once this file exists, one need not triangulate again as long as the structure file is not modified (in fact, in many cases the triangulation process can take a very long time so it is in general unwise to unnecessarily re-triangulate a structure file on each inference run and better instead to just a pre-computed pre-existing triangulation file).

Once the triangulation file exists, one can run “Viterbi decoding” to compute the Viterbi path (see §8.4.11.10) as follows:

```
gmtkViterbi -fmt1 ascii -of1 ascii_file_list.txt -nf1 0 -ni1 1 \
    -inputMasterFile params.master -str hw1.str \
    -verbosity 0 -vitValsFile -
```

(note this should all be one line) which will produce the following output:

```
Segment 0, number of frames = 1, viterbi-score = 0.000000
Printing random variables from (P,C,E)=(0,0,1) sections
Ptn-1 C: hw_hidden(0)=Hello_World
____ PROGRAM ENDED SUCCESSFULLY WITH STATUS 0 AT Saturday September 07 2013, 04:38
```

where one can see the string “Hello_World” printed. If we were to modify the CPT concentrate_at_one to have value 1 0 rather than 0 1, the output would take the form:

```
Segment 0, number of frames = 1, viterbi-score = 0.000000
Printing random variables from (P,C,E)=(0,0,1) sections
Ptn-1 C: hw_hidden(0)=Goodbye_World
____ PROGRAM ENDED SUCCESSFULLY WITH STATUS 0 AT Saturday September 07 2013, 04:38
```

14.1.2 Two-frame HMM-based “Hello World”

As mentioned above, there are in fact many ways to print a string. The example above is fairly basic as it has only one random variable and a CPT which determines its maximum value.

In this section, we modify the above examples to have GMTK print out the string “Hello World” as the output of two separate variables, rather than one variable using a symbol table containing the monolithic string Hello_World. We do this using a slightly more complex example, a simple HMM.

The structure (lets assume its name is hw2.str) file becomes the following:

```
GRAPHICAL_MODEL hw2

frame: 0 {
    variable: hw_hidden {
        type: discrete hidden cardinality 3;
        conditionalparents: nil using DenseCPT("initial_state");
        symboltable: collection("print_strings");
    }
    variable: hw_observed {
        type: discrete observed 0:0 cardinality 4;
        conditionalparents: hw_hidden(0)
            using DenseCPT("observation_distribution");
    }
}
frame: 1 {
    variable: hw_hidden {
        type: discrete hidden cardinality 3;
        conditionalparents: hw_hidden(-1)
            using DenseCPT("transition_matrix");
        symboltable: collection("print_strings");
    }
    variable: hw_observed {
```

```

    type: discrete observed 0:0 cardinality 4;
    conditionalparents: hw_hidden(0)
        using DenseCPT("observation_distribution");
    }
}
chunk 1:1

```

We see now that we have two frames of random variables specified in the template, which corresponds to the two-frame HMM template given in Figure 8.85-(a). The state variables are called `hw_hidden` and the observations called `hw_observed`. At frame zero, the state has no parents and uses an initial state distribution by the name of `initial_state`. It also is associated with a symbol table called `print_strings`. Again, the symbol table is used to associate a string to each integer random variable value.

The observed variable at frame zero now specifies that it has a parent, as given by the `conditionalparents` attribute of the random variable. It asks for the parent with name `hw_hidden(0)`. The `(0)` says that it wants the version of this variable at offset 0 away from the current frame (meaning it wants the variable `hw_hidden` in the same frame as the variable being defined. Since the variable being defined is at frame 0, it asks also for `hw_hidden` in frame zero. It should be noted that the notation `foo(j)` for variable `foo` and value `j`, `j` is an relative offset not an absolute position.

The observation at frame zero also asks for a set of parameters, uses a distribution named `observation_distribution`.

In frame one, the hidden state is the same except now it specifies its parent as `hw_hidden(-1)`, in this case an negative offset is used, specifying it wants itself as a parent from one previous frame. The observation at frame one is the same as in frame zero, thanks to the use of parent specification using relative offsets.

Lets again call the parameter file `params.master` which must define all of the CPTs used in the structure file, this time three of them. It has the following contents:

```

%%%%%
% Dense CPTs
%%%%%
DENSE_CPT_IN_FILE
inline % the CPT is inline rather than comes from a file.
3      % The number of Dense CPTs that follow.
0      % The number of the current CPT (zero indicates this is first).
initial_state % The name of the CPT.
0      % number of parents (zero means no parents).
3      % num of values (so this is for a ternary variable).
0.9 0.05 0.05 % distribution, high probably of starting in first state.
%
1      % The number of the current CPT (one indicates this is second).
transition_matrix % The name of the CPT.
1      % Number of parents (zero means no parents).
3 3    % Parent cardinality (three) and self cardinality (also three).
      % For each value of the parent, we have a row of three values.
0.1 0.8 0.1 % a 3x3 table of entries corresponding to p(child/parent).

```

```

0.1 0.1 0.8 % Columns correspond to child values from 0 to child card - 1.
0.8 0.1 0.1 % Rows correspond to parent values, 0 to parent card - 1.
%
2      % The number of the current CPT (two indicates this is third).
observation_distribution % name of CPT.
1      % number of parents (zero means no parents).
3 4    % parent cardinality (three) and self (child) cardinality (four).
0.8  0.066 0.066 0.066 % a 3x4 table of values.
0.066 0.8   0.066 0.066
0.066 0.066 0.8   0.066

%%%%%%%%%%%%%
% Collection of names, used in this case for a symbol table
%%%%%%%%%%%%%
NAME_COLLECTION_IN_FILE
inline % the collection is inline rather than comes from a file.
1      % the number of collections that follow, one in this case.
0      % The number of the current collection (zero indicates first).
print_strings % name of the collection.
3 % collection length, to be followed by four strings.
Hello
Goodbye
World

```

The parameters in `params.master` for the CPTs (namely `initial_state`, `transition_matrix`, and `observation_distribution`) are set in a particular way so that the appropriate Viterbi path will occur. Perhaps the only tricky aspect of the parameter file is when multi-dimensional matrices are specified. This will be further fleshed out in §17.2.1 — for now, suffice it to know that in the two-dimensional case, rows correspond to parent values and columns to child values, as documented in the comments in `params.master` itself.

Once again, a third observation component must be specified again, in this case, a list of file names each of which must contain an ASCII version of the observations. In this case, the list of files is of length three, corresponding to three segments. The ASCII file list is a file called `ascii_file_list.txt` and has the following form:

```

ascii_file_1.txt
ascii_file_2.txt
ascii_file_3.txt

```

The contents of `ascii_file_1.txt` is:

0
2

the contents of `ascii_file_2.txt` is:

```
1  
2
```

and the contents of `ascii_file_3.txt` is:

```
0  
1  
2  
3
```

One must next triangulate the structure file using a command such as:

```
gmtkTriangulate -str hw2.str
```

We then compute the Viterbi path (§8.4.11.10) using the following command:

```
gmtkViterbi -fmt1 ascii -of1 ascii_file_list.txt -nf1 0 -nil 1 \  
-inputMasterFile params.master -str hw2.str -vitValsFile -
```

which produces the following output:

```
=====  
Segment 0, number of frames = 2, viterbi-score = -2.854233  
Printing random variables from (P,C,E)=(1,1,0) sections  
Ptn-0 P: hw_hidden(0)=Hello  
Ptn-1 C: hw_hidden(1)=World  
=====  
Segment 1, number of frames = 2, viterbi-score = -3.665163  
Printing random variables from (P,C,E)=(1,1,0) sections  
Ptn-0 P: hw_hidden(0)=Goodbye  
Ptn-1 C: hw_hidden(1)=World  
=====  
Segment 2, number of frames = 4, viterbi-score = -4.162322  
Printing random variables from (P,C,E)=(1,3,0) sections  
Ptn-0 P: hw_hidden(0)=Hello  
Ptn-1 C: hw_hidden(1)=Goodbye  
Ptn-2 C: hw_hidden(2)=World  
Ptn-3 C: hw_hidden(3)=Hello  
____ PROGRAM ENDED SUCCESSFULLY WITH STATUS 0 AT Saturday September ...
```

There is output for each of the three segments. The first segment prints out our desired “Hello World” as the Viterbi path corresponds to integer values 0 and 1 which are mapped to the appropriate symbols. The second segment has a different Viterbi path and so prints out a different string, “Goodbye World” in this case. The third segment is of length four, so the template is unrolled twice in this case. The observations in this segment, and the parameters of the model, are such that the shown string corresponds to the Viterbi path.

Exercise 158. Modify the above so that even in the length-four segment, only the string “Hello World” is printed.

There are an unlimited number of ways of producing a model that prints “Hello World.” At this point, however, we move from this simple example to describing the GMTK overall process for producing and then using a DGM.

14.2 Using GMTK: The process of specifying, training, and then using a model

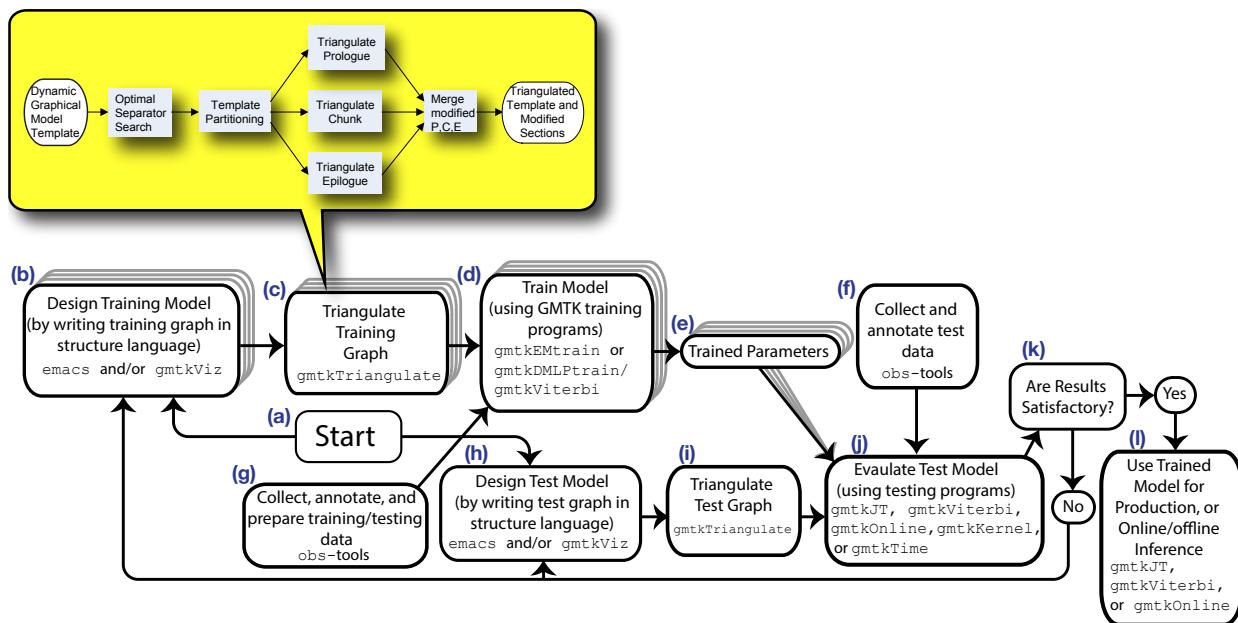


Figure 14.1: GMTK’s design workflow. This diagram shows the typical process that a user will use to conceptualize, design, build, iteratively test and debug, and then use a DGM in GMTK.

To start using GMTK, it is important to get an idea of the GMTK’s workflow, namely how and when does one specify a model, train it based on data, test, debug, and evaluate it, iterate the process, and then ultimately use it.

This workflow is shown in Figure 14.1. We start in box (a). In (b), we begin with an idea for what we might want to do (e.g., a new construct or feature) and design a graph template (which specifies a family of distributions $\mathcal{F}(G, \mathcal{M})$ that one will be working within, as described in Chapter 1). In GMTK, the graph is written in a textual language and done with an editor (such as `emacs`) possibly also with the assistance of GMTK’s graph viewer `gmtkViz` (which allows you to easily view and format they layout of but not modify existing models).

Once this has been done, we must triangulate (Figure 14.1, box (c)) the template using `gmtkTriangulate`. For those familiar with using HMMs, note that unlike with a standard HMM-based system, there is a triangulation step with graphical models that can have a significant effect on the performance of the model, both for exact and approximate inference. The process of general graph triangulation, and various heuristics for doing so as well as some tips for graphical models, is described in §6.8.1. GMTK’s `gmtkTriangulate` tool will help to perform this triangulation step for you, and has implemented a large assortment of triangulation heuristics. Also, it is important to realize that forming a triangulation and junction tree does not imply that exact inference must be used — as described in §13.4, the approximate inference strategy can be guided by an underlying junction tree (and which might not be a minimal triangulation of the graph). Interestingly, the best triangulations for exact inference are (often) not the best triangulations for approximate inference, and finding a triangulation for either exact or approximate inference, given that even in the exact case it is an NP-hard optimization problem, sometimes involves more art, luck, and intuition than anything else. A variety of tricks that have been found to be useful for triangulating specific dynamic graphical models using `gmtkTriangulate` are described in §18.4.

Most machine learning methods require training data to be used to adjust the parameters of a model. It’s important not only to collect this data, but also to accurately annotate (or label, or transcribe) this data and prepare it (e.g., data-type transformations) for use in GMTK. GMTK supplies a number of tools for this purpose collectively called the `obs-tools` (Figure 14.1, box (g)), which includes utilities to print, augment, compute simple statistics over, convert, concatenate, and compress data stored in a number of forms useful for sequential and dynamic graphical models. A number of file formats are supported including simple ASCII and binary files, flat ASCII files (which are minimally annotated ASCII files), pfiles (which are used in speech recognition), HTK files (used by the HTK HMM toolkit), and hdf5 files (commonly used in bioinformatics). These tools, and the various file formats, are described in §18.6.

The next step is to train the model (Figure 14.1, box (d)). There are several ways of training, one of which is to use the celebrated EM (expectation maximization) algorithm [110, 47] via the program `gmtkEMtrain`. An alternative, more recent, method is to combine this form of training with embedded Viterbi training of perhaps deep models iteratively alternating between using `gmtkViterbi` and `gmtkDMLPtrain`. However the training process is done, the result are a set of trained parameters (box (e)) associated with the graph specified in box (b). Once we’ve done this, we have instantiated one particular $p \in \mathcal{F}(G, \mathcal{M})$.

It is often the case that the test graphical model is different than the one used for training, despite the fact that the parameters produced for the training graph are used for the testing graph. Hence, box (h) requires a model design similar to box (b), except here the goal is testing or decoding. One example of this occurs in speech recognition, where a different (and potentially much larger) data set is used for training a language model $p(y_t|y_{t-1})$ than the data set which is used for training the acoustic model $p(x_t|y_t)$ in an HMM $\prod_t p(x_t|y_t)p(y_t|y_{t-1})$ that combines both kinds of factors (see §8.4 for a complete description of HMMs).

In fact, it is not uncommon for multiple separate training graph structures to be designed, each of which is used to produce only a subset of the parameters (say for each factor) that are ultimately used together in a single unified test model of box (h) (In Figure 14.1, (b), (c), (d), and (e), this is depicted by some of the boxes showing as multiple grey boxes underneath a top box). This ability, indeed, is one of the great benefits of generative models (§8.8.16), namely that very different size and kinds of training sets can be used to train parts of a statistical model that, thanks to the local normalization properties, can happily coexist together in a final test model.

Once the test model is developed, it is triangulated in box (i) (similar to box (c)). The trained parameters (box (e)) and the test model (box (h) and (i)) are then evaluated with new test data (collected in box (f)), as shown in box (j). This is also called the “decoding” process, and often refers to finding the most likely hidden variable values, something that is necessary for, say, speech recognition or tagging or any pat-

tern recognition process. There are a number of GMTK programs one might use here, including `gmtkJT` (general probability of evidence, but which also has many facilities for producing posterior distributions), `gmtkViterbi` for doing Viterbi decoding, `gmtkOnline` for doing online or real-time streaming inference (akin to Kalman filtering), `gmtkTime` for timing and benchmarking a given model, and `gmtkKernel` used for producing quantities such as the Fisher kernel.

If the results are satisfactory (and they seldom are), which means that any error rates are sufficiently low, or analysis or indirect results that use the output of the inference in box (j) are acceptable, then we move to box (l) and start using the system. The more typical case is that something is not quite working as well as desired yet (or there might be model specification errors that result in, say, zero-clique errors), and we have to begin again (or modify the existing structures) by visiting again boxes (b) and/or (h). When this happens, we then analyze the output, by looking at resulting parameters, look at GMTK's debugging output (the `-verbosity` options, which allow the user to trace the inference process), examine accuracies or error rates, understand why we might be getting zero-clique errors, attempt to understand why results not yet working as well as desired, and think about what modeling and/or training modifications need to be done.

Chapter 15

Representing Structure in GMTK

The next most crucial property of any graphical model after its semantics has been specified is its structure. The structure of a graphical model specifies the set of nodes and edges that constitute a graph. It is therefore important to have a natural and relatively easy-to-use method with which to specify these structures.

There are a number of possible ways of doing this. For example, given the number of nodes (say N), it is possible to specify an $N \times N$ adjacency matrix where a non-zero in the matrix position i, j says that there is a directed edge from parent i to child j . Alternatively, a textual language can be used to specify structure. A third possible way is to use a visual graphical user interface, where pointers, menus, and on-screen icons are used to specify structure.

GMTK is meant for dynamic graphical models, and hence its abilities are flexible in specifying templates (§8.1) that expand along one dimension. GMTK uses a textual template specification language (sometimes called “GMTKL”) that generalizes the typical intra-frame and inter-frame dependencies used for DBN and DGM specification whose purpose was fully described in §8.11.1. We’ve already seen a few simple examples of the use of this language in §14.1. In this section, we fully describe GMTKL and its abilities.

15.1 GMTKL: The GMTK Structure Language

GMTKL is a structure specification language for specifying generalized DGM templates in GMTK (see §8.11.1 for discussion of generalized DGM templates). GMTKL uses the generalization (see §8.11.1) of the standard two-frame DBN methodology to specify and then unroll a template.

A GMTKL template declares a monotonically increasing list of (time) frames, starting at time zero, and follows with a chunk specifier. Each frame defines an arbitrary collection of random variables or a factor (a bundle of random variables that interact via a function). All random variables within a frame are given a frame number corresponding to the frame in which they are defined. A random variable definition consists of naming the random variable, and specifying a variety of possible attributes such as the variable’s parents, switching parents, type (discrete or continuous), parameters and implementation (e.g. discrete probability tables, deterministic relationship, decision trees, Gaussian mixtures, etc.). A frame can also specify undirected factors (and hence specify undirected models) — minimally, a factor consists of a bundle of random variables that changes the graph structure so that the corresponding set of random variables is a clique. A frame in a template might not (and in fact often does not) have the same nodes and edges as other frames, thereby making GMTKL quite flexible in the graphs that it can represent.

At the end of a template is a chunk specifier, and is given as two integers $N : M$ where $M \geq N$, which divides the template into a prologue (the first N frames, namely frames 0 through $N - 1$), a chunk (consisting of template frames N through M inclusive), and an epilogue (the last $T_t - M - 1$ frames, where T_t is the number of frames of the template). The middle $M - N + 1$ chunk of frames is “unrolled” or repeated a number of times until the dynamic network is long enough to fill up the frames of a specific

```

%%% This is a comment %%%
% Simple GMTKL template for an HMM.

% The first frame, starting at index 0
frame: 0 {

    % variable 'state' in first frame
    variable : state {
        type : discrete hidden cardinality 4000;
        % state at frame 0 has no switching parents.
        switchingparents : nil;
        % state at frame 0 has no normal parents.
        % 'pi' is the name of the conditional probability table (CPT)
        % containing the parameters for this variable. Since
        % the variable has no parents, it is only a 1-D table.
        conditionalparents :
            nil using DenseCPT("pi");
    }

    % variable observation in first frame
    variable : observation {
        % 'observation' is continuous 39-dimensional feature
        % vector. '0:38' refers to elements in the observation file.
        type : continuous observed 0:38;
        % 'observation' at frame 0 has one parent, 'state(0)'.
        % Note that 'state(0)' refers to the state variable
        % in the current frame.
        switchingparents : nil;
        conditionalparents : state(0)
            % collections and mappings will be described later,
            % when we talk about parameters.
            using mixture collection("global")
            mapping("state2obs");
    }
}

% The second frame
frame: 1 {

    % variable 'state' in second frame
    variable : state {
        type : discrete hidden cardinality 4000;
        switchingparents : nil;
        % In this case, state in the second frame has
        % as a parent 'state(-1)' which means the state
        % variable one time frame preceding this one. In other
        % words, when a variable named 'var' is specified
        % as 'var(n)' to designate a parent, the 'n' is
        % the relative frame offset. If n=0, it refers
        % to the variable named 'var' in the current frame,
        % if n=-1, it refers to the previous frame. Other
        % values of n (both positive and negative) are also
        % possible.
        conditionalparents : state(-1)
            using DenseCPT("transitions");
    }

    % variable 'observation' in second frame
    variable : observation {
        type : continuous observed 0:38;
        % 'observation' at frame 1 has one parent, 'state(0)'
        % Note that 'state(0)' refers to the state variable
        % in the current frame, which in this case is frame 1.
        switchingparents : nil;
        conditionalparents : state(0)
            using mixture collection("global")
            mapping("state2obs");
    }
}

% The chunk specifier, saying only that
% frame 1 is the chunk to be repeated.
chunk 1:1;

```

Figure 15.1: GMTKL specification of a simple HMM structure. The feature vector in this case is 39 dimensional, and the total possible number of hidden states is 4000. The chunk specifier is 1 : 1 which means that frame 1 is duplicated or "unrolled" enough times to create an arbitrarily long network structure.

utterance. GMTK’s generalization into a prolog, chunk, and epilogue of the a standard DBN notion of intra- and inter-frame edges can be quite useful in specifying a wide variety of both desirable and computationally tractable structures, as was discussed in §8.11.1 and which will be further seen in this chapter.

Figure 15.1 shows an example of a GMTKL template for a simple but complete HMM. The template consists only of two frames (frame 0 and frame 1) each with a hidden variable `state` and an observed variable `observation`. In both frames, `state` is a hidden discrete variable with cardinality 4000, which means that it can have at most the 4000 values, and the values it may take on consist of the integers 0 through 3999. Note that all discrete variable in GMTK takes on values consisting of the integers ranging from 0 up through one less than its cardinality. The variable `state` has no switching parents (to be described below). In fact, the switching parents attribute need not be given in the structure file when there are no switching parents. At time frame 0, `state` has no parents at all, and is distributed according to a single one-dimensional dense conditional probability table (CPT) named `pi`, that must contain 4000 probability values. GMTK probabilities and other parameters are not represented in a structure file, and instead are kept elsewhere (see Section 16).

At frame 1, `state` now has a parent which is the variable `state` at the preceding frame. This is specified using the relative integer frame offset `-1`. In general, variables are defined in a frame, using the `variable : varname` notation, without using any relative frame offset. Different frames can specify the same variable name, and it is the frame location that disambiguates multiple variables of the same name (but in different frames). Two variables may *not* have the same name within the same frame.

When a name of a parent variable is to be specified, the parent variable name uses a relative frame offset to specify, relative to the frame of the current child, which variable is the parent. The syntax is `varname(n)` where `n` is either zero (the same frame as the current child), negative (one or more frames **before** (earlier than) the current frame), or positive (one or more frames **after** (later than) the current frame). This and the above, for example, imply that a variable can not have a parent variable both with the same name and with an offset of 0. In other words, a (string-name, integer-offset) pair must be unique in a structure file.

In frame 1 of Figure 15.1, the `state` variable uses a dense CPT called `transitions`. Unlike in frame 0, in frame 1 this must be a two-dimensional 4000×4000 CPT, specified in the parameter files. This is because the `state` variable in frame 1 has as a parent in frame 0 named ’`state`’ (i.e., `state(-1)`) which has cardinality 4000. The `state` variable in frame 0, on the other hand, has no parents.

The `observation` variable is also specified in both frames, and in each case `observation` uses the current `state` variable as a parent. `observation` is a continuous and observed variable. Moreover, it is a 39-dimensional vector observation, as specified by the string `0 : 38`. This is a range specification which says that this variable corresponds to features 0 through 38 in the observation feature file (to be described in Section 17.9). For this `observation` variable, the parent `state(0)` affects the child variable by determining which of a number of Gaussian mixtures to use, one possible mixture for each value of `state(0)`. This mapping, from the value of a discrete variable `state(0)` to a particular Gaussian mixture is specified using the decision tree mapping `state2obs` and via the collection of names `global`. These are described in later sections.

In Appendix B, a context free grammar for GMTKL is given as a set of production rules. Those familiar with this notation can quickly determine what syntactic constructs are allowed in GMTKL. Later sections will describe each case with a number of examples.

15.1.1 GMTK’s Random Variables

A random variable (RV) is the basic unit in GMTK. A variable may be discrete or continuous.

A discrete RV is always scalar valued in GMTK. A discrete RV has a cardinality c which specifies the number of valid values the RV may possess, ranging from 0 to $c - 1$ inclusive.

A discrete RV may be hidden or observed. If the variable is observed, that means that its value is always known, and must be specified in some way, either immediately in the structure definition (see below in this section), or the variable must be associated with a slot in the global observation matrix (see Section 17.9.2). An observed random variable is one for which “hard evidence” (see §5.4.1) has been provided. For this kind of random variable, the attribute must specify that the random variable is observed. To specify virtual evidence §5.5, the random variables disposition must be given as hidden, and an additional factor used (§17.10).¹

A hidden discrete RV is most commonly integrated or “marginalized away” (i.e., summed) or is “maxed away”, as in a Viterbi-like computation (see §8.4.11 and §8.4.11.10) as part of probabilistic inference in GMTK. Therefore, the cardinality of a hidden random variable can have a significant affect on the computational complexity and memory requirements of the resulting inference.² There are of course aspects other than just the cardinality which can affect complexity. For example, the average number of a variable’s non-zero probability values (averaged over the different sets of possible parent variable values) can have a enormously significant affect on the wall-clock running time of GMTK. The following is an example of a discrete hidden RV.³

```
variable : state {
    type : discrete hidden cardinality 4000;
    ...
}
```

An observed RV, on the other hand, must always have its value specified somewhere (it is observed after all and, as mentioned above, corresponds to “hard evidence” in §5.4.1), and there are two ways this can be done. In the first way, a range specification is used to specify a single element in the observation file (Section 17.9), for example:

```
variable : state {
    type : discrete observed 26:26 cardinality 4000;
    ...
}
```

In this example, feature number 26 (specified as a length-one range 26:26), which *must* be an integer in the observation file (i.e., not a floating point value), is used to supply the observed value for the variable state at each time frame. Hence, the value which is observed can change at each time frame. Furthermore, all values supplied in the observation file must be in the valid range of the RV (it must be non-negative, and

¹While hard evidence can be given with virtual evidence, §5.5, it is more computationally efficient to in such cases specify the random variable as observed even though mathematically it is equivalent.

²In GMTK, moreover, this effect can be very non-linear, as a function of the random variable’s cardinality, due to the way GMTK stores variables in memory, so there are times that even a decrease in cardinality by one can lead to a significant memory savings while at other times a decrease by more than one has no effect at all.

³In GMTK, there are distinct structure files and parameter files. A structure file for the most part specifies a graph structure and the resulting factorization properties of a given model. The parameter file specifies the specific parameterization of a distribution that abides by the factorization specified in a structure file. Parameter files and structure files are not the same file. There are, however, a few cases in GMTK where values that perhaps can be called “parameters” are given in a structure file (e.g., see §15.1.3), and other cases where certain graphical structures over observed variables are given in a parameter file (e.g., see §17.5). The present section concentrates on structure files.

must not be greater than $c - 1$ if c is the number of possible values the RV might have, something that we call the RV’s *cardinality*), or a run-time error will occur.

Also, the value 26 must be within the right range – it must be valid with respect to the current observation file and specify an integer. If you try to choose an value is outside the range of integer values in the observation file, you will get a run-time error.

The second way of specifying a value for a discrete observed random variable is to do so inline, or what is called an *immediate RV value*, as in the following example:

```
variable : state {
    type : discrete observed value 2000 cardinality 4000;
    ...
}
```

In this example, the variable `state` will always have observed constant value 2000 (specified “immediately”) for any and all time frame in which it occurs. This is useful when using the observed variable as a (virtual) child, and which can greatly enrich the sets of distributions and conditional independence statements that a Bayesian network (i.e., a type of directed graphical model) can represent, for example via the use of virtual evidence (see §5.5 and §17.10).

A continuous random variable in GMTK is a vector of real-values. At this time, GMTK supports only *observed* continuous random variables. Hidden random variables must be simulated by quantizing them and using a discrete random variable.

```
variable : obs {
    type : continuous observed 0:38;
    ...
}
```

The example defines a continuous observed random vector named `obs` of dimension 39. The given range determines which feature elements in the observation file correspond to the RV at each time frame. These observations must be continuous (not discrete). The range `0:38` is a vector of length 39, and the range given must be consecutive (i.e., at this time, GMTK does not support skip integer ranges to specify a continuous observed vector as you have seen used elsewhere, for example if you wanted to get all even numbered features via the range specification `0:2:38` this would result in an error⁴.

For another example, to create three separate continuous observation vectors over regular features (such as MFCCs), their deltas (derivatives), and their double deltas (second derivatives), one could use the following:

```
variable : obs_feats { type : continuous observed 0:12; ... }
variable : obs_delts { type : continuous observed 13:25; ... }
variable : obs_ddelts { type : continuous observed 26:38; ... }
```

⁴But note that the `obs-tools` and GMTK’s command-line parameters have the ability to choose a subset of rows of an observation matrix so that one need not be consecutive relative to the data files themselves. See the section on the `obs-tools`, §18.6

			Child			
			Discrete		Continuous	
			Hidden	Observed	Hidden	Observed
Parent	Discrete	Hidden	X	X		X
		Observed	X	X		X
	Continuous	Hidden				
		Observed				X

Table 15.1: GMTK can implement dependencies between two parent and child random variables depending on the type of the variables. This table shows which forms are currently implemented in GMTK. The squares marked with an X are currently supported.

This of course makes the assumption that the first 13 features in the observation file are the original features (i.e., standard MFCC cepstral coefficients C_0 through C_{12}), the next 13 are the deltas, and so on.

There is no restriction on observed range overlap so one could (if one wanted) produce two continuous (or discrete) observed random variables that taken on some (or all of the same values). For example, the following construct is allowed.

```
variable : obs_feats { type : continuous observed 0:12; ... }
variable : obs_overlap { type : continuous observed 6:15; ... }
```

The variable `obs_feats` and `obs_overlap` will have the same value for the observation range for which there is an intersection. The meaning of this is analogous to the following: suppose one had two 2-dimensional observed variables $X_t = (X_t^1, X_t^2)$ and $Y_t = (Y_t^1, Y_t^2)$. Having their values overlap at one value would be similar to asking for the probability $p(X_t^1 = a, X_t^2 = b, Y_t^1 = b, Y_t^2 = c)$, where b is the overlapped common value.

15.1.2 RV Dependencies: Parents and Children

Dependencies⁵ between parent and child random variables may be specified in several ways, depending on the types of the corresponding parent and child. There are a total of 16 possible combinations of parent and child type, as shown in Table 15.1. GMTK supports a subset of the possible dependencies in these 16 cases. In particular, if both parent and child variables are discrete, then any configuration of observed or hidden is supported. Furthermore, both hidden and observed discrete variables may be parents of observed continuous variables, and observed continuous variables may be parents of other observed continuous variables.

GMTK currently does not support hidden continuous parents or children, or observed continuous parents of discrete children. Note that some of these options can cause probabilistic inference to become intractable.

Discrete variables may be parents of other discrete variables regardless of their continuous/observed disposition. In the following example, the variable `parent` is a binary parent of the variable `child`, using a dense 2×2 CPT named `foo` to obtain the probabilities. In this case, both the parent and the child live in the same frame, since the frame offset (0) is used (see above).

⁵More precisely, we should say “edges” since the word “dependency” would suggest that there indeed is a dependency between a parent and a child. In a Bayesian network, the existence of an edge implies that there might be but there is not necessarily a dependency. A lack of an edge implies that there is guaranteed to be some conditional independence. We use the term “dependency” rather loosely, however, since this has become common. Feel free in the following to substitute the word “edge” where “dependency” is used.

```

variable : parent {
    type : discrete hidden cardinality 2;
    ...
}
variable : child {
    type : discrete hidden cardinality 2;
    ...
    conditionalparents: parent(0) using DenseCPT("foo");
}

```

In the example, either of the `hidden` keywords could change to `observed` in order to make one or more of the variables observed. In that case, of course, the observed value would need to be specified in some way.

Discrete parents of continuous observations may also be specified. In such a case, the value of the discrete parent determines which distribution (e.g., amongst say a set of forms of Gaussian mixtures) should be used. In the following example, the variable `state` is a discrete hidden parent of the continuous observed `child` variable.

```

variable : state {
    type : discrete hidden cardinality 2;
    ...
}
variable : child {
    type : continuous observed 0:1;
    ...
    conditionalparents: parent(0) using
        mixture collection("global")
        mapping("state2gaussianIndexfoo");
}

```

Continuous observed variables may also be parents of other continuous observed parents. The edge structure that should exist between continuous observed variables is not represented in the structure file, however, and instead is represented in the data file as `dlinks` and `dlink` matrices (see Section 17.5). While a design goal of GMTK was to keep everything relating to structure in the structure file, an exception was made in this case for several reasons. First, observation variables are vectors, and therefore the dependency structure in this case is between the *individual elements* of the vector observation variables. Second, the vectors themselves can be fairly long (e.g., typically 39 or more). Third, the dependency structure is often learned in a data-driven fashion. Fourth, such structure often *switches* (see below) which is to say that the structure might change depending on the current values of the discrete parents. For all of these reasons, including observation structure in the structure file would complicate the file beyond what seemed reasonable. Structure over observations is fully described in Section 17.5.

15.1.3 Random Variable Weights

The probability of a set of random variable values are determined by the parameters associated with the random variable and any factors (directed or undirected) in which they are contained.

The probability of a set of random variable values, when they are involved in a directed factor, be they discrete or continuous, hidden or observed, may also be adjusted in a variety of different ways using a GMTK “weight.” In particular, let’s suppose that p is a probability score given by a random variable and its set of parent values (or factor and a set of instantiated random variable values). GMTK has the ability to modify (or weight) this score in the following ways:

$$p' = \text{penalty} * p^{\text{scale}} + \text{shift} \quad (15.1)$$

When this is done, the value p' will be used rather than p whenever it is the case that the value p is needed during inference. The value of penalty or scale can be one and the value of shift can be zero, thereby annulling the weight’s effect, thus one need not have the affect of all three adjustments simultaneously.

In GMTK, each of the values (for penalty, scale, or shift) is specified only optionally (so any of the eight combinations can occur). Typically, a random variable weight has a fixed value (or set of values) that is constant for all time frames. Alternatively, it can be a value that is obtained from an real-valued element of the global observation matrix (see Section 17.9), thereby potentially giving a different (time-dependent) weight for this random variable at each time frame.

A weight can also switch depending on a switching parent (to be described in §15.1.4) which means that the place where the value is obtained is dependent on the current value of some other discrete random variable. This latter capability allows us to express constructs like word insertion penalties (§15.1.5.1), commonly used in speech recognition, and to generalize word insertion-like penalties to any dynamic graphical model or data domain other than speech.

There are a number of reasons for why one might wish to adjust the score of a random variable value (and for some of the reasons, it will not be clear how we can achieve this until we describe switching weights in §15.1.5).

One reason is that we might have multiple “streams” of features, each using its own observation distributions (e.g., each stream might have its own set of Gaussian mixtures or its own set of deep models). With weights, it is possible to weight each stream separately in order to, say, capture a notion of the reliability of each stream (the less reliable stream can be down-weighted).

It may also be possible to induce sparsity during training of a CPT using the weight mechanism and the EM algorithm. The learning of the CPT $p(a|b)$ depends on the posterior $p(a, b|\text{evidence})$ computed via the EM algorithm. One can (in some sense) reduce the entropy of this posterior to some extent using a probability scale on the CPT for $p(a|b)^\alpha$. What this does is change any probability from p to p^α and if done during EM training, $\alpha > 1$ will reduce the entropy of the posterior. That is, given a vector $p = (p_1, p_2, \dots, p_k)$ and another vector $p' = (p_1^\alpha, p_2^\alpha, \dots, p_k^\alpha) / \sum_i p_i^\alpha$ then $H(p') < H(p)$ if $\alpha > 1$ and $H(p') > H(p)$ if $\alpha < 1$ where $H(p) = -\sum_i p_i \log p_i$ is the entropy of the distribution. Hence, one way to encourage sparsity in the CPT is to use $\alpha > 1$ during training and $\alpha = 1$ during testing/decoding. Alternatively, one can use $\alpha > 1$ during training, and then truncate the low-probability values and then re-normalize. This constitutes a form of *posterior regularization*.

In speech recognition, moreover, it is common to adjust the dynamic range of different factors of a probability distribution. For example, it is common in speech recognition to use a Gaussian mixture where $p(x|q) = \sum_i p(x|q, i)p(i|q)$ where $p(x|q, i)$ is a multivariate Gaussian, and $p(i|q)$ is a state specific mixture weight. This might be combined with a state posterior distributing of the form $p(q)$ which is a discrete distribution. The issue is that $p(x|q)$ might have a very high dynamic range, meaning that its values might get very small (say 10^{-50} or 10^{-100} , depending on the dimensionality of x and also how far away x is from any of the means of the Gaussian mixture). On the other hand, $p(q)$ is unlikely to get anywhere close to that small. I.e., if q has, even, 10^6 possible values, a reasonable smallest value of $p(q)$ might be 10^{-8} . In such a scenario, when considering $p(x|q)p(q)$, the $p(x|q)$ factor will determine the most likely q irrespective of $p(q)$ since the largest and smallest value of $p(q)$ for different q values will not change the rank of q when scored by $p(x|q)p(q)$. In order to rectify this situation, one might wish to scale the Gaussian, as in $p(x|q)^\alpha p(q)$

with $0 < \alpha < 1$ or alternatively as in $p(x|q)p(q)^\alpha$ where $\alpha > 1$. GMTK's weight scale mechanism can easily do this.

In the multi-stream example above, if each stream is scored with a Gaussian of differing dimensions, and one wishes to equalize the dynamic range of the different dimensional Gaussians (and thereby equalize the “importance” of each of the streams, even though they are of different dimensions), a scale can be used for this purpose. For example, if x and y correspond to features of different dimensional streams (meaning x and y are different length vectors), and they are scored respectively using $p(x|q)$ and $p(y|q)$, then one could produce a combined score of the form $p(x|q)^{\alpha_x} p(y|q)^{\alpha_y}$ where α_x and α_y are the corresponding stream scales. For example, if $x \in \mathbb{R}^{d_x}$ and $y \in \mathbb{R}^{d_y}$, then one might set $\alpha_x \leftarrow 1/d_x$ and $\alpha_y \leftarrow 1/d_y$.

GMTK currently does not have the ability to “train” (i.e., automatically adjust) the values of the weight attribute. If learning is to be done, they must be considered as hyper-parameters to be adjusted using multiple GMTK evaluations with adjustment made using either a grid-search or derivative-free optimization method [95] such as a Nelder–Mead method, which can be significantly faster than grid search.

Inside a random variable attribute list, a penalty of -100 , scale of -2 and shift of -5 would be specified as:

```
Parameter File
weight:
    penalty -100 scale -2 shift -5;
```

Of course you can just do one or two of them as in:

```
Parameter File
weight:
    scale -2;
```

or

```
Parameter File
weight:
    penalty 3 scale -2;
```

Any random variable, discrete or continuous, can have its own weight specification. The order of penalty scale shift doesn't matter in a weight specification in a structure file, it will have the same effect, meaning that this construct:

```
Parameter File
weight:
    penalty -100 scale -2 shift -5;
```

and this one

```
Parameter File
weight:
    scale -2 penalty -100 shift -5;
```

are identical.

The above shows only the weight attribute itself. What follows will be some examples of a complete random variable specification along with various kinds of weight attribute.

In the simplest form, a random variable might have an exponential *weight* associated with it. The weight is such that, for a random variable named X , the probability evaluation is $p(X = x)^{\text{weight}}$. A complete random variable specification with a such a weight could be defined as follows:

```

variable : state {
    type: discrete hidden cardinality 3;
    weight: scale 5.0;
    conditionalparents: nil using DenseCPT("state0");
}

```

In this case, all probability scores will be raised to the power 5.0.

The next example shows a lone penalty of -100.

```

variable : word {
    type: discrete hidden cardinality VOCAB_SIZE ;
    conditionalparents:
        word(-1) using DenseCPT("wordBigram");
    weight:
        penalty -100 ;
}

```

The following is an example that uses both a penalties and a scale.

```

variable : P {
    type: discrete hidden cardinality NUM_OF_PHONES;
    switchingparents: PT(-1) using mapping("directMappingWithOneParent");
    conditionalparents: P(-1) using NGramCPT("timit_bigram");
    weight: penalty -1.0 scale 2.0;
}

```

Since the shift is not specified, it is identical to if it was set to zero in Equation (15.5).

The following, moreover, shows an example where the adjustments are time-dependent and come from an observation file:

```

variable : P {
    type: discrete hidden cardinality NUM_OF_PHONES;
    switchingparents: PT(-1) using mapping("directMappingWithOneParent");
    conditionalparents: P(-1) using NGramCPT("timit_bigram");
    weight: penalty 3:3 scale 4:4;
}

```

The 'int:int' notation means that the value should be obtained the global observation matrix (§17.9) at that position at the frame number of the child. This ability, therefore, is quite flexible, as any of penalty, scale, and shift can come from the observation file at the frame of the child variable. In the example above, we get the penalty (respectively scale) at each frame from position three (respectively four) in the global observation matrix.

A complete specification of a random variable with a time-dependent weight is as follows:

```

variable : state {
    type: discrete hidden cardinality 3;
}

```

```

    weight: scale observation 2:2;
    conditionalparents: nil using DenseCPT("state0");
}

```

This indicates that the time-varying weight of the state variable should come from the 3'rd continuous observation element of the global observation matrix (Section 17.9)o.

Here's what precisely penalty, scale, and shift do, based on the values specific the structure. Values are given in the log domain, so the adjustments are actually as follows: For the penalty:

$$\log(p) + \text{penalty} \quad (15.2)$$

for the scale:

$$\log(p) * \text{scale} \quad (15.3)$$

and for the shift:

$$\log(p) \boxplus \text{shift} \quad (15.4)$$

where \boxplus is the binary log-addition operator. Thus, the scale directly multiplies the log probability, i.e., $\text{scale} * \log(p)$, which means the scale value can be positive or negative when given in the structure file.

Both discrete and/or continuous variables may have weights. This feature can therefore be used to produce acoustic, pronunciation, and/or language model scaling factors that have been shown to be useful for speech recognition decoding systems.

The use of weights will make it such that the resulting probability distributions are no longer proper (i.e., they do not necessarily sum to unity). Great care should therefore be exercised when using weights during training (if they are used at all) as unexpected results may occur if this is not done. For example, the posterior regularization framework mentioned above (to produce sprase distributions) is one case where the use of weights during training can make sense. It is expected that weights will often be useful during decoding, *after* training has occurred, to give different relative importance to either different streams of observations, or between the discrete (e.g., language, pronunciation) and continuous (e.g., acoustic) random variables.

As mentioned above, there will be more discussion on weights in §15.1.5 when we discuss switching weights, where the weight specification that is applied to a random variable is dependent on the value of some other set of random variables (and that will make possible constructs like word insertion penalties, §15.1.5.1).

There is one last perhaps slightly exotic example that should be mentioned here, and that is regarding explicitly given but zero valued scales. I.e., in the transformation:

$$p' = p^\alpha \quad (15.5)$$

with $\alpha \geq 0$, what happens when either p , or α , or both are zero. In GMTK, when $p = 0$ and $\alpha > 0$ we always get back the value of 0 (i.e., $0^\alpha = 0$). When $p > 0$ and $\alpha = 0$ we get back the value of 1 (i.e., $p^0 = 1$). When both $p = 0$ and $\alpha = 0$, we still get one (i.e., $0^0 = 1$), which is arguably the right result due to limiting arguments, $\lim_{\beta \rightarrow 0} \beta^\beta = 1$.

There is a subtle effect that can occur in the $0^0 = 1$ case, which is that with different triangulations, different numbers of zero probability events might occur. Normally, any zero-scoring event has no effect on the final score in the inference since any hypothesis having a zero probability clique event is removed from the set of active hypotheses. When the zero probability events are not canceled due to $0^0 = 1$, however, then there will be a different number of clique accumulations depending on the current triangulation, and we will get different resulting scores. Hence, using zero-valued weights one must be careful, as otherwise unexpected results might occur.

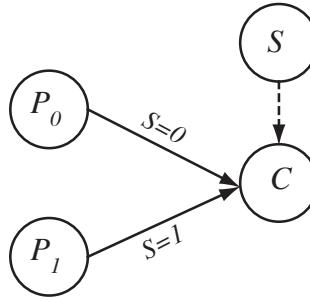


Figure 15.2: When $S = 1$, P_0 is C 's only parent, and when $S = 2$, P_1 is C 's only parent. S is called a switching parent, and P_0 and P_1 conditional parents.

15.1.4 Switching Parents and Dependencies

In GMTK, there are two types of parent designations – a parent may be either a switching parent (a parent that determines the set of other parents a variable might have), a conditional parent (a parent whose parental status is conditional on the value of a switching parent), or both. Moreover, the way in which a conditional parent affects a child variable (e.g., the CPT) might switch depending on the current value of the switching parent.

Networks that use switching parents are also called Bayesian multi-nets [168, 45]. Normally, a variable has only one set of parents. GMTK, however, allows a variable's parents to change (or switch) conditioned on the current values of other (conditional) parents.

We will motivate this feature with an example. Suppose we are given four random variables, the binary variable S and variables C , P_0 , and P_1 , and we are interested in representing the distribution $p(c|p_0, p_1)$, but depending on the value of S , either P_0 or P_1 has no effect on C . In other words, we have that

$$\begin{aligned} p(c|p_0, p_1) &= \sum_s p(c, s|p_0, p_1) \\ &= p(c|p_0, p_1, S = 0)p(S = 0) + p(c|p_0, p_1, S = 1)p(S = 1) \\ &= p(c|p_0, S = 0)p(S = 0) + p(c|p_1, S = 1)p(S = 1) \end{aligned}$$

This representation can significantly reduce the number of parameters required to represent a probability distribution. For example, $p(C|P_1, S = 1)$ needs only a two-dimensional table whereas $p(C|P_0, P_1)$ requires a three-dimensional table. The equation above corresponds to the conditional independence statements $C \perp\!\!\!\perp P_1 | \{P_0, S = 0\}$ and $C \perp\!\!\!\perp P_0 | \{P_1, S = 1\}$. In other words, conditional independence can occur for certain values of random variables, rather than for all values (as is more typical). This scenario is depicted graphically in Figure 15.2.

GMTK supports switching parents by having two types of parent designators. The following example shows how to implement the switching parent functionality described in Figure 15.2.

```

variable : S   { type : discrete hidden cardinality 2; ... }
variable : P0  { type : discrete hidden cardinality 2; ... }
variable : P1  { type : discrete hidden cardinality 2; ... }
variable : C   {
    type : discrete hidden cardinality 2;
    switchingparents: S(0) using mapping("s1toindex");
    conditionalparents:
  }
  
```

```

    P0(0) using DenseCPT("p0CPT")
    | P1(0) using DenseCPT("p1CPT");
}

```

Note that the decision tree `s1toindex` is such that it can query based on values of its “parents”, which in this case is the variable `S(0)`. That is, when DT `s1toindex` mentions value `p0`, it is getting the current value of `S(0)`. *TODO: include an example of what a DT s1toindex might look like.*

In this example, all variables are binary (cardinality 2), and the decision tree⁶ named `s1toindex` is used to map from values of `S` (which in this case is either the value 0 or the value 1) to either the integer 0 (which causes parent P_0 to be used) or to the integer 1 (which causes parent P_1 to be used). When P_0 is used, the two-dimensional CPT `p0CPT` will provide the conditional probabilities of C , and correspondingly for P_1 .

A slightly more complicated example follows.

```

variable : S1 { type : discrete hidden cardinality 100; ... }
variable : S2 { type : discrete hidden cardinality 100; ... }
variable : P1 { type : discrete hidden cardinality 100; ... }
variable : P2 { type : discrete hidden cardinality 100; ... }
variable : P3 { type : discrete hidden cardinality 100; ... }
variable : C {
    type : discrete hidden cardinality 2;
    switchingparents: S1(0), S2(0) using mapping("s1s2toindex");
    conditionalparents:
        P1(0), P2(0) using DenseCPT("p1p2CPT")
        | P1(0), P3(0) using DenseCPT("p1p2CPT")
        | P2(0), P3(0) using DenseCPT("p1p2CPT");
}

```

In this example, variable `C` is the child variable with two switching parents `S1` and `S2` and conditional parents `P1`, `P2`, and `P3`. This corresponds to the equation:

$$\begin{aligned}
P(C|P_1, P_2, P_3) &= P(C|P_1, P_2, \{S_1, S_2\} \in \mathcal{R}_0)P(\{S_1, S_2\} \in \mathcal{R}_0) \\
&\quad + P(C|P_1, P_3, \{S_1, S_2\} \in \mathcal{R}_1)P(\{S_1, S_2\} \in \mathcal{R}_1) \\
&\quad + P(C|P_2, P_3, \{S_1, S_2\} \in \mathcal{R}_2)P(\{S_1, S_2\} \in \mathcal{R}_2)
\end{aligned}$$

where \mathcal{R}_1 , \mathcal{R}_2 , and \mathcal{R}_3 are disjoint regions in the joint state space of S_1, S_2 . The regions are determined by the decision tree `s1s2toindex`. The decision tree functions to map any pair of values of S_1, S_2 that lives within \mathcal{R}_0 to the integer 0, thereby selecting the first set of conditional parents (P_1 and P_2 in this case). Similarly, the decision tree will map any pair of values of S_1, S_2 that live within \mathcal{R}_1 to the integer 1 (and likewise for \mathcal{R}_2 and integer 2). This way, even though the joint state space of the set of switching parent variables might be large (10,000 in this case), only a small list of conditional parents need be specified.

Note also that GMTK’s switching parent functionality can also be used to switch implementations, or CPTs (conditional probability tables, described in Section 17.2). For example, in the following:

⁶Decision trees are described in Section 17.2.5.

```

variable : S { type : discrete hidden cardinality 2; ... }
variable : P { type : discrete hidden cardinality 2; ... }
variable : C {
    type : discrete hidden cardinality 2;
    switchingparents: S(0) using mapping("s_to_index");
    conditionalparents:
        P(0) using DenseCPT("pCPT_1")
        | P(0) using DeterministicCPT("pCPT_2");
}

```

The switching parent S is used not to switch between different parents, but instead to switch between two different CPTs. Hence, a switching parent can also switch between distribution types while keeping the conditional parents the same (i.e., there is no need to change the parents).

Switching parents have an analogous effect when used as parents of continuous random variables. Consider the following example:

```

variable : S { type : discrete hidden cardinality 2; ... }
variable : P0 { type : discrete hidden cardinality 2; ... }
variable : P1 { type : discrete hidden cardinality 2; ... }
variable : C {
    type : continuous observed 0:1;
    switchingparents: S(0) using mapping("s1toindex");
    conditionalparents:
        P0(0) using mixture collection("p0_collection")
            mapping("p0_to_gm_index")
        | P1(0) using mixture collection("p1_collection")
            mapping("p1_to_gm_index");
}

```

In this case, the switching parent S determines which parent will determine the Gaussian mixture at the current time. Each parent uses a different collection and different decision tree mapping function (collections are described in Section 17.1).

In GMTK, switching parent functionality may also exist for dependencies that exist between continuous observations. In fact, explicit switching parents as described above are not used to specify switching structure over observations. Consider the following example:

```

variable : P { type : discrete hidden cardinality 2; ... }
variable : C {
    type : continuous observed 0:1;
    switchingparents: nil;
    conditionalparents:
        P(0) using mixture collection("global")
            mapping("p_to_gm_index");
}

```

Even though parent P is not officially a switching parent, it could in effect be a switching parent, depending on the type of Gaussian mixture that is used. As will be seen in Section 17.4, each Gaussian mixture can have a mixture of different types of mixture components, and each component is a general conditional Gaussian. This means that the component could be a diagonal- or full-covariance Gaussian, a sparse-covariance Gaussian, or a Gaussian with conditional dependencies on parents that come from either the past or the future of the current time frame. Furthermore, each Gaussian component can have a separate set of such dependencies. Therefore, by having P select amongst sets of mixtures having different components, and by having each mixture use components with different dependencies, P becomes a switching parent. More on this topic will be described in Section 17.5.

15.1.5 Switching Weights

In §15.1.3 we described how scores can be modified by a weight. In this section, we extend this flexibility to show how the score modification via a random variable weight can depend on the value of another set of random variables using the same “switching” mechanism described used for switching parents as described in §15.1.4. We also in this section give a number of examples of doing this.

The syntax of adding weights to a random variable set of attributes (which will be further elaborated upon below with examples) is:

```
Parameter File
weight:
% constant scale, the penalty comes from position 0 in the
% observation file, and constant shift.
    scale value 1.0 penalty 0:0 shift value 0.5 ;
% second shift condition, scale from pos 0 in file, penalty from
% pos 1 in the file
    | scale 0:0 penalty 1:1
% third shift condition, scale fromm pos 2 in the file.
    | scale 2:2;
```

Here, we have three switching conditions, the first condition specifies a weight with an immediate scale value of 1.0, an penalty of whatever happens to be the value in the observation file at position 0, and an immediate shift value of 0.5. The second condition specifies only a scale and a pentalty, both of which come from the observation file (at positions 0 and 1 respectively). The third condition specifies only a scale to come at observation position 2. This example shows that one may mix the form where the values of the weights come from, meaning some might be immediate (and specified directly in the structure file) and others might be time-dependent and come from an observation file.

We next answer what controls the switching between the different weights. GMTK uses the same mechanism as is used for switching parents, meaning the switching parents are mapped down to a finite set of conditions (equivalently a partitioned into a finite set of sets of values) and for each condition, a different weight specification may be applied.

The following is an example that uses both switching parents with penalties and shift.

```
variable : P {
    type: discrete hidden cardinality NUM_OF_PHONES;
    switchingparents: PT(-1) using mapping("directMappingWithOneParent");
    conditionalparents: P(-1) using DeterministicCPT("copyParent") |
        P(-1) using NGramCPT("timit_bigram");
```

```

    weight: shift 1.0 | penalty -1.0 scale 2.0;
}

```

In this example, when $\text{PT}(-1) = 0$, a shift of 1.0 is applied, but when $\text{PT}(-1) = 1$, both the penalty and the scale kick-in. Of course you can use all the three, i.e., penalty, shift and scale in each case.

This means that the syntax for switching weights is almost the same as switching parents (except that rather than one of a set of parents specifications being switched between, one of a set of weight specifications are switched between).

Note that the GMTK's lattice CPT specification can be used with switching weights, and this can provide for some interesting constructs.

15.1.5.1 Word Insertion Penalties, and their Generalization

In speech recognition, it is common to allow for a separate mechanism to be used to penalize the score each time an additional word is hypothesized, irrespective of which word is being hypothesized. The reason for this is that there is some desire to keep the recognizer from highly scoring hypothesizes that correspond to many words. For example, suppose the shortest possible length of each word is only three frames long. It is perfectly reasonable to assume that every so often one might find a word that is realized so quickly in an utterance, it is unlikely that all words are realized as quickly as is allowed in a word model. A word insertion penalty is a system that provides additional penalty (i.e., a probability score reduction) based on how many words in an utterance are hypothesized.

Given an HMM for a T -frame speech recognition utterance $x = x_{1:T}$ corresponding to a series of $n(w)$ words $w = w_{1:n(w)}$, we can describe the process of imposing word insertion penalties using the following equation:

$$W^* = \underset{w}{\operatorname{argmax}} (\log p(x|w) + \alpha \log p(w) + n(w) * \beta) \quad (15.6)$$

where α is a language model scale factor, and β is the per-word word insertion penalty, and $n(w)$ is number of words in the string w (so the more words we choose, the more the penalty if we choose β to be negative).

Word insertion penalties can easily be specified via the switching weight method in GMTK. Suppose there is a `word` variable at each frame that contains a word, and that uses as a switching parent a binary variable `wordTransition(-1)` from the previous frame. Whenever `wordTransition(-1) = 1` we hypothesize a word using a bigram language model `wordBigram`, but each such new hypothesis should be penalized by a certain amount, say $\beta = -100$. On the other hand, when there is no word insertion, the score should not be so penalized. This can be done using the following fragment:

```

variable : word {
    type: discrete hidden cardinality VOCAB_SIZE ;
    switchingparents: wordTransition(-1)
        using mapping("directMappingWithOneParent");
    conditionalparents:
        word(-1) using DeterministicCPT("copyCPT")
        | word(-1) using DenseCPT("wordBigram");
    weight:
        nil
}

```

```

    | penalty -100 ;
}
```

This gives a switching weight, and adds a penalty in the 2nd case (when we insert a new word). Penalties are given as log base e (natural log), so that we can implement true word insertion penalties, by directly specifying β as in Equation (15.6).

Note that we can have both a word insertion penalty and a language model scale factor by changing the weight to:

```

weight:
nil
| penalty -100 scale 5.0;
```

which would then exponentiate the language model score by 5.0 only each time it is used to score a word transition, while at the same time penalizing the score by -100 for inserting an additional word.

Since GMTK can specify any DBN, however, systems like word insertion penalized can be used to add extra additive penalty to any hypothesis that is whose number of occurs is to be discouraged from being too large.

15.1.6 GMTKL Templates and Unrolling

As mentioned above, a GMTKL structure file defines a template which then is “unrolled” enough times in order to be of length T frames. GMTK uses the generalized DBN template system that is fully described in §8.11.2. Still, we offer a description of this system here, as well as a few examples, to allow the toolkit user to quickly start using GMTK. Some of the nuance of these templates, however, (and there is some) is described only in §8.11.2.

Consider the following structure file fragment which shows only parent-child relationships for clarity:

```

frame: 0 {
    variable : S { ...; conditionalparents : nil ...; }
    variable : O { ...; conditionalparents : S(0) ...; }
}
frame: 1 {
    variable : S { ...; conditionalparents : S(-1) ...; }
    variable : O { ...; conditionalparents : S(0) ...; }
}
frame: 2 {
    variable : S { ...; conditionalparents : S(-1) ...; }
    variable : O { ...; conditionalparents : S(0) ...; }
    variable : E { ...; conditionalparents : S(0) ...; }
}
chunk 1:1;
```

In this template, the three frames are not identical to each other. In first frame, S has no parents. In the second frame, S has one parent. In the third frame, there is an extra variable E which has S in the current

frame as a parent. The chunk specifies an integer range of 1:1 which designates only frame 1. Hence, it is only frame 1 that is unrolled $T - 3$ times to make a network of length T frames (recall, unrolling zero times is the same as just using the template). This is depicted in Figure 15.3.

GMTK allows unrolling of more general structures than the one described above. A GMTK template consists of a prologue, a chunk to be repeated, and an epilogue. Each of the prologue, chunk, and epilogue can consist of any number of frames, and need not have the same set of random variables. While the prologue and the epilogue can have zero frames, the chunk must have at least one frame.

Let's say that T_p , and respectively T_c and T_e , are the number of frames in the prologue, chunk, and epilogue, meaning that the template consists of a total of $T_p + T_c + T_e$ frames. After GMTK unrolls the chunk to obtain a T frame network, the network will consist of $T = T_p + kT_c + T_e$ for some positive integer k (see Figure 15.4).

If no such k can be found which satisfies this equation, then GMTK can optionally either skip the first few frames (justify to the right), or skip the last to frames (justify to the left), or skip a bit at the beginning and the end (center justify), depending on command line parameters.⁷

Note that when the chunk is unrolled, the parameters for each of the variables in the unrolled network are shared with the corresponding variables in the rolled template. In this way, unrolling allows for an easy way to tie parameters of different random variables together, based on only specifying the parameters in the original template (See Figure 15.3). Other forms of GMTK parameter sharing are described in Section 17.6.⁸

One must be careful not to specify a template that, when unrolled, results in an invalid network. Consider the following template fragment for example.

```
frame: 0 {
    variable : R { type : discrete hidden cardinality 2; ... }
    ...
}
frame: 1 {
    variable : S { type : discrete hidden cardinality 4; ... }
    conditionalparents : R(-1) using ... ;
    ...
}
frame: 2 {
    variable : S { type : discrete hidden cardinality 4; ... }
    conditionalparents : S(-1),R(-2) using ... ;
    ...
}
frame: 3 {
    variable : S { type : discrete hidden cardinality 4; ... }
    conditionalparents : S(-1) using ... ;
    ...
}
chunk 2:2
```

⁷The justification is controlled by the `-justification str` parameter in which `str` may be either `left`, `center`, or `right`. This then places the model either on the left, the center, or to the right relative to the observations. This is further described in §18.3.

⁸Note, that if no parameter sharing is desired across an unrolled network, then an extra frame counter variable can be specified which switches the implementation of all the variables in the current frame.

There are several problems with this template. Consider the graph in Figure 15.5 which shows the template after unrolling one time, and the variables are renamed with primes. From the semantics of unrolling, the variables S'_3 and S'_2 share exactly the same set of parameters since they both come from S_2 in the template. S_2 , however, requires a parent named R_0 which has cardinality 2, and a parent named S_1 which has cardinality 4. The same set of parent names and cardinalities need to exist in the same frames relative to the frames containing S'_2 and S'_3 , or for any variable which is derived from S_2 after unrolling. The problem with the above network that S'_3 is not able to find a parent named R in frame 1.

Even if there was a variable called R in template frame 1, if that R in frame 1 had cardinality 4 there would be a problem since S'_3 would ask for a parent variable named R that has cardinality 2 rather than 4. Therefore, any such R variable in template frame 1 should also have cardinality 2 for this template to be valid.

In general, unrolling is valid only if all parent variables in the unrolled network are *compatible* with those parents that exist in the template. A variable is said to be compatible if has 1) the same name, 2) the same type (i.e., discrete or continuous), and 3) the same cardinality (number of possible values). For example, suppose that A is a random variable in the template having B as a parent that is k frames to the left of A in the template. After unrolling, each variable A' in the unrolled network that is derived from A in the template must have a parent having the same name as B , it must be k frames to the left of each A' , and must have the same type and (if discrete) cardinality as B . If these conditions are not met, the template is invalid and GMTK will report an error.

As can be seen, one must take care to specify templates that correspond to valid unrolled networks. GMTK will detect when an unrolled template is invalid, and will report an error. There is much discussion about valid and invalid templates in §8.11.4 — it is recommended that the reader understand that section as there are indeed some obscure and slightly counterintuitive cases that can arise.

15.1.6.1 Templates and Multi-Rate/Multi-Scale Processing

Since a template chunk may consist of several frames, and each frame may contain any number of different variables, it is possible to specify multi-rate or multi-time-scale networks using GMTK. This may be done by specifying a multi-frame chunk with a different set of variables (either hidden or observed, or both) in each frame. Consider the template fragment given in Figure 15.6 and shown graphically in Figure 15.7.

In this template, the chain R_t exists every frame, and is used to determine the distribution over observation features 0 through 4. The chain Q_t proceeds at 2/3 the rate of R_t , and it is used to determine the distribution over observation features 5 through 9, but these features only have a distribution for 2 out of every three frames. Observations must exist for each frame in the global observation matrix (see Section 17.9), so in this case, features 5 through 9 are ignored every third frame. Also, for every two steps of Q_t (equivalently every three steps of R_t), Q_t possesses a dependency on R_t . Using this feature, therefore, one can easily specify general multi-rate GM networks where variables occur at rates which are fractionally but otherwise arbitrarily related to each other.

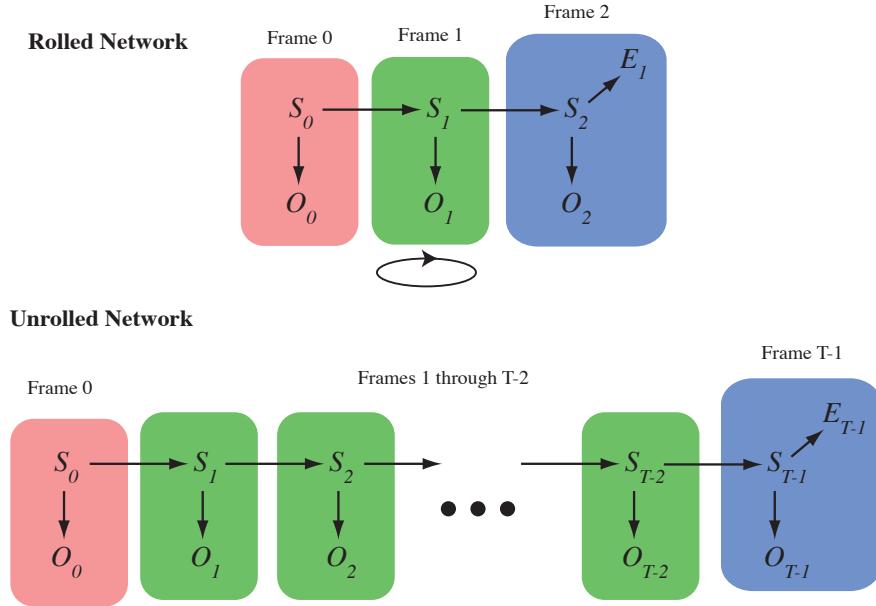


Figure 15.3: An example of a rolled and a correspondingly unrolled network. On the top, the template corresponds to a rolled up network (i.e., the template) with three frames and 7 random variables. Note that both the first and the last frame of the template are different from the middle frame. The chunk consists of the variables in frame one. On the bottom, frame one has been unrolled $T - 3$ times producing a network of length T consisting of frames 0 through $T - 1$ and with $2T + 1$ total distinct random variables. This is reflected in the unrolled network, as the middle $T - 2$ frames are identical in structure, but again it should be clear that they are distinct random variables. That is, in the $T - 2$ middle frames of the unrolled network, all variables share the same sets of parameters specified by their respective counterparts in the template. Note that the parameters of S_1 through S_{T-2} are essentially *shared*, and are the same as that which is specified in the template for variable S_1 . Sharing in this case means that the probabilities of the same value for different random variables are forced to be identical (so that $p(S_1 = s) = p(S_2 = s) = \dots = p(S_{T-2} = s)$). Similarly, the parameters for O_1 through O_{T-2} are the same, and are those specified for template variable O_1 . Parameter sharing amongst variables at different times is utilized both during parameter training (such as with the EM algorithm) and just during probability evaluation. It is also possible with GMTK to share parameters between variables that do not correspond to the same template variable (see Section 17.6). Parameter sharing as a form of regularization and Bayesian priors is also discussed in §8.11.

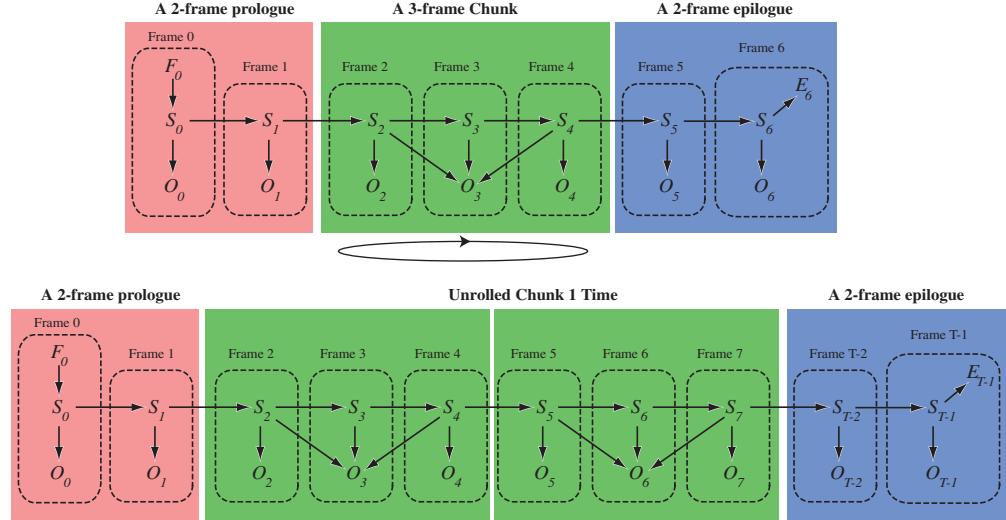


Figure 15.4: A multi-frame template (top) with a two-frame prologue, a 3-frame chunk, and a 2-frame epilogue when unrolled one time produces the network at the bottom.

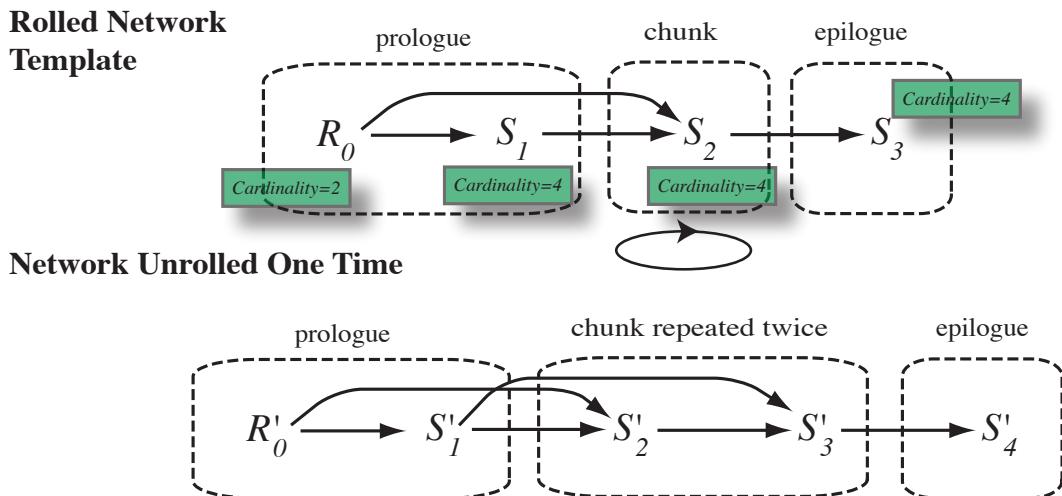


Figure 15.5: An example of an invalid template because of name and cardinality incompatibilities. In particular, in the unrolled network, the variable S'_3 (originally S_2 in the template) has parents S'_1 and S'_2 . S'_1 has the wrong cardinality (4 rather than 2) and the wrong name (S rather than R).

```
frame: 0 { variable : R {...}  variable : Q {...} ...
           variable : O { type: continuous observed 5:9;
                           conditionalparents : Q(0) using ... ; }
           variable : X { type: continuous observed 0:4;
                           conditionalparents : R(0) using ... ; })
frame: 1 { variable : R {...}  variable : Q {...} ...
           variable : X { type: continuous observed 0:4;
                           conditionalparents : R(0) using ... ; })
frame: 2 {
           variable : R { type : discrete hidden ... ;
                           conditionalparents : R(-1) using ... ; }
           variable : Q { type : discrete hidden ... ;
                           conditionalparents : Q(-2),R(0) using ... ; }
           variable : O { type: continuous observed 5:9;
                           conditionalparents : Q(0) using ... ; }
           variable : X { type: continuous observed 0:4;
                           conditionalparents : R(0) using ... ; }
}
frame: 3 {
           variable : R { type : discrete hidden ... ;
                           conditionalparents : R(-1) using ... ; }
           variable : Q { type : discrete hidden ... ;
                           conditionalparents : Q(-1) using ... ; }
           variable : O { type: continuous observed 5:9;
                           conditionalparents : Q(0) using ... ; }
           variable : X { type: continuous observed 0:4;
                           conditionalparents : R(0) using ... ; }
}
frame: 4 {
           variable : R { type : discrete hidden ... ;
                           conditionalparents : R(-1) using ... ; }
           variable : X { type: continuous observed 0:4;
                           conditionalparents : R(0) using ... ; }
}
...
chunk 2:4
```

Figure 15.6: A structure file for a template specifying a multi-rate model. The graph for this structure file is shown in Figure 15.7.

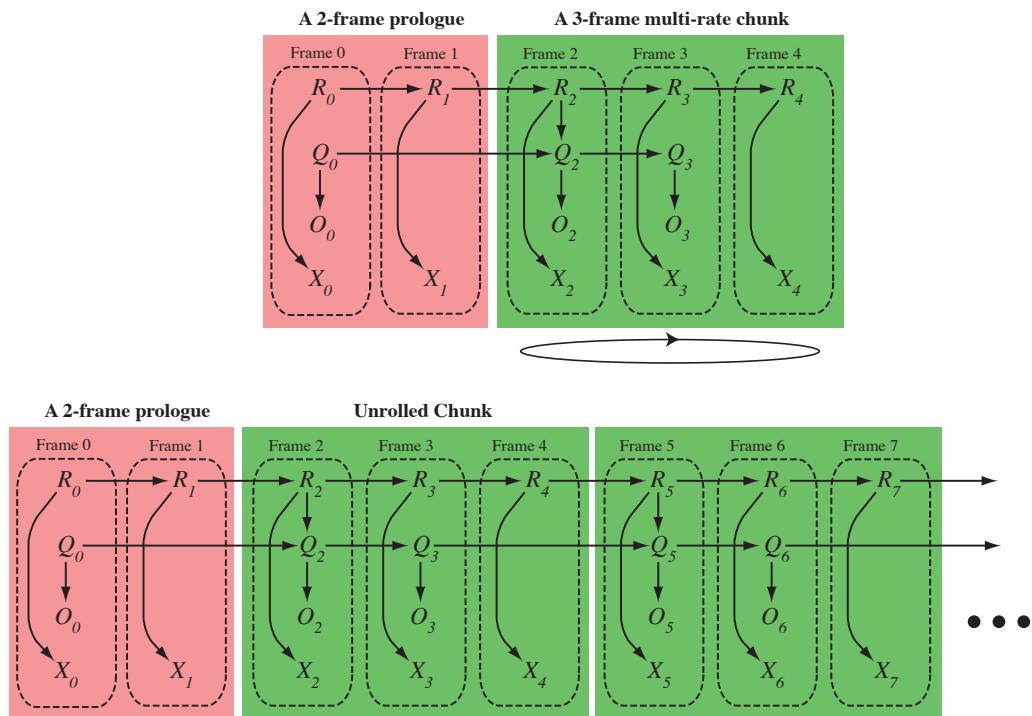


Figure 15.7: An example of a network template that specifies a multi-rate model, both over hidden and over observed variables. Top: the multi-rate template. Bottom: the unrolled multi-rate network. The structure file for this graph is shown in Figure 15.6.

Chapter 16

Representing Parameters in GMTK

The structure, the implementation, and the parameters of a graphical model are separate aspects of the model, a feature that GMTK exploits. For example, the structure consists of a graph’s set of nodes and edges, the implementation consists of what the various dependencies or edges mean in terms of how parent and child variables are related to each other (linear, nonlinear, sparse CPT, deterministic CPT, etc.), and the parameters complete the representation by specifying values used by the implementations of the dependencies.

In GMTK, all model parameters for a graph are represented in a set of files separate from the structure file. Furthermore, the graph structure over continuous vector observations are also represented in a separate file than the hidden structure, as mentioned in Section 15.1.2.

In this section, we describe GMTK’s general parameter format and observation structure. GMTK parameters include probability values for CPTs, decision trees (used to represent deterministic mappings), means and variances of Gaussians, mixture coefficients (i.e., component responsibilities), all in a flexible way that allows for a rich set of parameter tying/sharing.

16.1 Numerical vs. Non-Numerical Parameters

GMTK makes a distinction between those parameters that are numeric in nature and are typically trained by, say, the EM algorithm, and those parameters which are entirely structural or relational and which are not trained using standard training procedures.

Numerical parameters include things such as Gaussian means, variances, component responsibilities, dense and sparse CPTs, and so on. Non-numerical parameters include things like decision trees, and, as will be seen, dependency link (or *dlink*) structures.

16.2 The GMTK basic parameter “object”

All parameters in GMTK are contained in basic GMTK parameter *objects*. Objects have names (any text string) which are used to specify and identify the object for various purposes. For example, some objects (e.g., Gaussian components) might require other objects (e.g., mean vectors) for their definition. These textual names are used by objects to refer to other needed objects. The names are also used by error messages to identify which object is having trouble. Within a given type of object (say covariances), names must be unique. This means that two covariance matrices may not have the same name. Different types of objects, however, may have the same name (so you may have both a mean and a covariance named `foo`).

Objects are defined in GMTK parameter files. These files use a syntax that reduces the chance of errors that might occur when reading in a parameter file. For this reason, you will notice a certain amount of

redundancy in the parameter files. The files are also designed so that a parser can quickly and efficiently read and interpret the files. The redundancy helps to both reduce errors and to speed reading.

There are a number of different types of objects (defined below). Objects are normally specified as a list of N objects of the same type. A list consists of 1) the number of objects that are about to follow, and 2) a list of that many objects, each preceded by the index number (zero-offset) of that object. The definition of each object includes a specification of its name, and any other required parameters. The definitions of each object are given in the following sections. An example of a generic list of objects follows:

<pre>% this is a comment 3 % three objects are to follow 0 % object 1 (zero offset) will follow <definition of object 1> 1 % object 2 will follow <definition of object 2> 2 % object 3 will follow <definition of object 3></pre>	Parameter File
--	----------------

The list consists of three objects. The contents of the definition of the objects depends on the type of object being defined (e.g., a decision tree, conditional probability table, and so on). As mentioned above, redundancy exists in the list (a specification of the number of objects that will follow in addition to the object number itself, which could be inferred while reading). If any inconsistencies are found when the parameter files are read in, GMTK will report an error. The basic data objects supported in GMTK are given in Table 16.1.

16.3 ASCII/Binary files, Preprocessing, and Include Files

Many parameter files can be either in plain ASCII or in binary. Clearly, binary files are preferred (and are often necessary) when speed and size become an issue. ASCII files can be useful when one is wishing to quickly view the parameter files, or when building scripts that generate initial sets of parameter files.

Note: ASCII files are meant for when you use GMTK on very simple models. ASCII files are appropriate when you are doing tutorials or testing out ideas on simple models. Once you start performing large-scale experiments, you should move to binary files. In some cases in the past, in fact, there have been instances where ASCII files have gotten so large that the system string libraries start failing (causing segmentation faults). Therefore, to avoid such things, it is best to move to binary files as soon as possible. Note that there is a gmtk tool that makes it easy to convert to/from ASCII and binary files: see the GMTK command `gmtkParmConvert` in Section ??.

All ASCII files (both structure *and* parameter files) are processed using the C preprocessor, `cpp`. This means that all features of the C pre-processor that are available in C are also available in GMTK parameter files. This includes things like defining variables, creating macros, conditional selection of text, including files, and so on. For example, you can include files as

```
#include "foo"
```

or define a macro as in

```
#define PATH /foo/bar/baz
```

Object	Master File Keyword	Description
Dense CPT	DENSE_CPT	A Dense CPT (conditional probability table)
Sparse CPT	SPARSE_CPT	A Sparse CPT, i.e., a CPT where many of the CPT values are zero, and are therefore not represented in the table.
Deterministic CPT	DETERMINISTIC_CPT	A deterministic CPT, i.e., a CPT where for each value of the parents, only one value of the child variable has non-zero probability.
Decision Tree	DT	A decision tree, used for a number of things including sparse CPTs, deterministic CPTs, collection of discrete parent variable values to selection of a Gaussian mixture, and set of switching parent values to a choice of conditional parents.
Dense Probability Mass Function	DPMF	A DPMF is a dense one-dimensional mass function which is used for mixture weights (responsibilities) in Gaussian mixtures. A DPMF is also used as the probabilities for Sparse CPTs
Sparse Probability Mass Function	SPMF	A SPMF is a sparse one-dimensional mass function, used for a Sparse CPT. A SPMF is just like a DPMF except zero probability elements are not represented.
Gaussian Mean	MEAN	A mean vector of a multi-variate Gaussian.
Gaussian Covariance	COVAR	A positive vector for the diagonal covariances of a multi-variate Gaussian.
dlink (dependency link) matrix	DLINK_MAT	A matrix of values giving the regression coefficients of a Gaussian. This corresponds to off-diagonal elements of a covariance matrix when non-diagonal covariance matrices are used.
dlink structure	DLINK	A table giving just the (potentially sparse) structure of dlink matrix values.
Gaussian component	GC	An object which groups together a mean, covariance, and (possibly) a dlink matrix to form a Gaussian component.
Mixture of Gaussians	MG	An object which groups together multiple Gaussian components and a DPMF to form a Gaussian mixture object. Note that these objects have the ability during training to split/vanish components.
Name collection	NAME_COLLECTION	An object which consists of a set of object names used for indirect reference in sparse CPTs, and in the mapping going from random variable values via decision trees to selections of Gaussian mixtures.

Table 16.1: Summary of GMTK basic objects.

Documentation on `cpp` is certainly outside the scope of this article, but we will in the following provide a few tips/hints on some of the available features when using `cpp` with GMTK.

First, note that for GMTK to run, `cpp` must be in a user's path. Normally, this is fine since the C compiler is typically in the user's path, and `cpp` is in the same location as the C compiler. It seems, however, that on some machines and for some users, `cpp` is not in the standard location or the standard or default path.

Note also that there is a number of ways to interface with `cpp` on the command line of GMTK programs. For example, you can define macros or variables on a GMTK program command line which can often be more convenient in scripts than having to write out multiple files, one per each training or testing chunk. See Section 18 on GMTK program command line arguments for a further description of this feature.

Note that user's environment variables are not used in GMTK parameter files. But supposing you would like to specify the path of an included DT with a script, you can use `cpp` variables, possibly by defining the value of the variable on the command line of one of the GMTK programs.

There are certain issues to beware of, however, when using `cpp` when *concatenating strings*. For example, supposing you would like to do the following:

Parameter File	
<pre>#define PATH /foo/bar/baz/ #define FILE1 spoons1 #define FILE2 spoons2</pre>	

and then wish to concatenate the `PATH` variable with both the `FILE1` and `FILE2` variables to create two file names with a complete path specification. Depending on the version of `cpp` you use, you might be able to get away with using the `cpp` concatenation operator `##`, and do something like:¹

<pre>... DT_IN_FILE PATH ## FILE1 ascii ... DT_IN_FILE PATH ## FILE2 ascii</pre>
--

For some versions of `cpp`, this won't work. Either there will be a space inserted between the path and the filename, or something else might be wrong. In that case, you can define a concatenation macro as follows:

<pre>#define CAT_B(a,b) a ## b #define CONCAT(a,b) CAT_B(a,b)</pre>

For esoteric reasons that we will not describe here, you need to define the concatenation macro in this way. Once this is done, you can then use:

<pre>... DT_IN_FILE CONCAT(PATH,FILE1) ascii ... DT_IN_FILE CONCAT(PATH,FILE2) ascii</pre>
--

which will expand out to

¹The example, by the way, states that there are two ASCII files named `spoons1` and `spoons2` both in directory `/foo/bar/baz` and both of which contain decision trees.

```

...
DT_IN_FILE /foo/bar/baz/spoons1 ascii
...
DT_IN_FILE /foo/bar/baz/spoons2 ascii

```

as you would expect.

There are a number of ways of using `cpp` to make your scripts easier to write. One thing to keep in mind, however, is that when ASCII files get large, sometimes it can take `cpp` quite a long time (30 minutes or longer!!) to parse and read in these files. In such cases, it is best to convert the files to binary. This will be faster not only because there is no ASCII to binary conversion required when reading in the file but also because `cpp` is not used to parse the file in this case. Converting from ASCII floating point numbers to an internal binary IEEE floating point representation itself can itself take quite a long time. There is an easy `ascii/binary` conversion program available for this purpose (see Section 18).

16.4 Input/Output Parameter Files

Apart from the graphical structure specification file, there are several ways to get parameter input into and output out of GMTK. All of the GMTK programs (the training, decoding, etc.) support this file procedure, so we will describe it in general terms. The general format described below applies to all the GMTK objects described in Section 16.2.

Each GMTK program supports an `inputMasterFile`, an `outputMasterfile`, an `inputTrainableParameters` file and an `outputTrainableParameters` file. These are specified in GMTK by command line options going by the same name. For each of the objects given in Section 16.2, GMTK keeps an internal array. These data files give the user several ways to manipulate these arrays of objects. The fundamental way this occurs is either to read in and append new objects to these arrays, or write the entire arrays out to disk (in either ASCII or binary format).

The master files have the capability of reading and/or writing all of the possible types of input parameters and/or objects (other than the graph structure). The master files are always in ASCII, and are always pre-processed by `cpp`. The master files do specify other files which might or might not be ASCII.

16.4.1 Input Master File

The input master file can be used to specify all the input parameters to a GMTK program. An input master file consists of a list of input specifiers, each consisting of 1) an *object keyword*, 2) an *object locator* specification, and then 3) a *list* of objects in the appropriate location.

An object keyword specifies one of the object types listed in Section 16.2. These keywords are listed in Table 16.1, but where each keyword is appended with the string `_IN_FILE`.

An object locator specification is either 1) the word `inline` or 2) a file name followed by either the string `ASCII` or `BINARY`. If `inline` is given, then the list of objects is given immediately following the `inline` keyword, in ASCII, within the master file itself. If, instead, a file name is provided, then the list of objects is taken from the file with that name which is presumed to be in either ASCII format or binary format depending on what is specified.

For example, here is a portion of a master file that consists of an inline list of dense conditional probability tables (the objects themselves are described in Section 17.2).

```

1 DENSE_CPT_IN_FILE inline
2 % there are two Dense CPTs

```

```

3 0 % first one
4 wordUnigram % name of first one
5 0 % num parents
6 5 % num values
7 0.2 0.2 0.2 0.2 0.2
8
9 1 % 2nd CPT
10 wordBigram % name
11 1 % one parent
12 3 3 % parent cardinality = 3, self card = 3
13 0.3 0.3 0.4
14 0.4 0.3 0.4
15 0.4 0.3 0.3

```

This portion of the master file declares two Dense CPTs, with names `wordUnigram` on line 3 and `wordBigram` on line 9 respectively.

The next example obtains the list of objects from an ASCII file:

```
DT_IN_FILE decision_tree.dts ascii
```

This section of the master file obtains a collection of decision trees in ASCII form from the file `decision_tree.dts`. If the file was in binary, the `ascii` keyword would instead be `binary`.

Note that if multiple keywords occur in a master file, the internal GMTK arrays are appended with the new objects encountered. For example, given the following:

```
DT_IN_FILE decision_tree1.dts ascii
DT_IN_FILE decision_tree2.dts ascii
```

First, the list of decision trees in the first file are read, followed by the second list.

The same data file may be specified multiple times within a master file, and such data files may contain a heterogeneous mix of object types. Suppose, for example, that you have a file named `file.data` containing a list of Dense CPTs followed by a list of decision trees. This can be read in as follows:

```
DENSE_CPT_IN_FILE file.data ascii
DT_IN_FILE file.data ascii
```

After encountering the dense CPTs in the file, GMTK will continue reading where it left off to read in the decision trees. While a file may contain a heterogeneous mix of different object types, any given file is either entirely in ASCII or entirely in binary (i.e., you can not mix ASCII and binary format data within a single file). Note also that multiple files can be open at the same time, so you can, for example, first read Gaussian means from one file, then read Gaussian covariance matrices from another, and then go back to the first file and continue reading decision trees.

16.4.2 Output Master File

The output master file is the analog of the input master file, but where parameters are specified to be written. Parameters are written, for example, whenever a change has occurred such as after an iteration of EM training, or during a conversion from ascii to binary format.

An output master file consists of a list of output specifiers, each consisting of 1) an *object keyword*, 2) a file name specifying where to write, and 3) an ascii/binary keyword.

An object keyword specifies one of the object types listed in Section 16.2. These keywords are listed in Table 16.1, but where each keyword is appended with the string _OUT_FILE. The file name specified is any file to which you have write permissions. Furthermore, you may write a file in either ASCII or binary format. Note that all of the GMTK internal objects of a given type are written — there is no way at this point to specify that certain Gaussian means should be written to one file, and certain other Gaussian means should be written to another.

Like with input master files, output file names may occur multiple times, where the new objects are appended at the end of the file each time the file name is encountered (except for the first time the file name is given when the file is truncated to zero length, see below). For example,

```
DENSE_CPT_OUT_FILE foo.out ascii
DPMF_OUT_FILE foo.out ascii
MEAN_OUT_FILE bar.out ascii
```

This output master file writes all Dense CPT objects to the file `foo.out`, and then appends all DPMF objects to the same file. It then writes all the Gaussian means to the file `bar.out`.

IMPORTANT: The first time an output file name is encountered in an output master file, the file with that name is truncated if it exists. In the example above, both `foo.out` and `bar.out` are truncated prior to writing. Therefore, it is important to ensure that these files, if they exist, do not contain needed data.

There exists a special character sequence in file names for the GMTK EM training program. If any of the file names have the string “@D” in them, then that string is replaced with the current EM iteration number. For example, consider the following:

```
DENSE_CPT_OUT_FILE foo@D.out ascii
DPMF_OUT_FILE foo@D.out ascii
MEAN_OUT_FILE bar@D.out ascii
```

If the EM iteration is 3, then the files that will be written are called `foo3.out`, and `bar3.out`. This makes it simple to keep track of the parameters of multiple iterations of EM.

16.4.3 Special Input/Output Trainable File

GMTK programs allow a simple way of reading and writing the trainable parameters in a fixed format using a single step. Essentially, trainable files provide a short-cut to read/write trainable files w/o needing to specify a special master file for this purpose. The trainable parameters are any of the parameters that could potentially be modified during training. This, therefore, does not include decision trees which are fixed once they are read in.

Specifying the `inputTrainableParameters` option on the command line of one of the GMTK programs (see Section 18), and, say, with file name `input.in` to be read in ASCII format, is equivalent to a master input file of the following format:

```
DPMF_IN_FILE input.in ascii
SPMF_IN_FILE input.in ascii
MEAN_IN_FILE input.in ascii
COVAR_IN_FILE input.in ascii
DLINK_MAT_IN_FILE input.in ascii
WEIGHT_MAT_IN_FILE input.in ascii
```

```
DENSE_CPT_OUT_FILE input.in ascii
MC_IN_FILE input.in ascii
MX_IN_FILE input.in ascii
GSMX_IN_FILE input.in ascii
LSMX_IN_FILE input.in ascii
MSMX_IN_FILE input.in ascii
```

Note that the last three entries (GSMG, LSMG, MSMG) are reserved, and will be used in future versions of GMTK. For now, those three groups are written with zero objects in each case. Note also that if the binary option is specified on the command line, then the behavior would be the same except all ASCII keywords would change to binary. Input master and input trainable files may be used together, where the input trainable parameters are appended to the internal GMTK arrays for the corresponding object type. In fact, master and trainable files must be used together if decision trees are to be used, as these objects are non-trainable.

Specifying the `outputTrainableParameters` option on the command line is analogous to the input case. In this case, however, all internal objects of the corresponding type are written to disk. This is equivalent to an output master file with the following format:

```
DPMF_OUT_FILE output.out ascii
SPMF_OUT_FILE output.out ascii
MEAN_OUT_FILE output.out ascii
COVAR_OUT_FILE output.out ascii
DLINK_MAT_OUT_FILE output.out ascii
WEIGHT_MAT_OUT_FILE output.out ascii
DENSE_CPT_OUT_FILE output.out ascii
MC_OUT_FILE output.out ascii
MG_OUT_FILE output.out ascii
GSMG_OUT_FILE output.out ascii
LSMG_OUT_FILE output.out ascii
MSMG_OUT_FILE output.out ascii
```

Again, changing the ASCII keyword to binary will change the format written.

Chapter 17

GMTK Representation Objects

All data structures and parameters in GMTK are specified using GMTK objects. This includes everything from objects that represent a collection of names of other objects to objects that contain a collection of real numbers which are the mean of a Gaussian. In this chapter, we will describe in detail the various GMTK objects, how they are used, their syntax, what they do, and how they are used together and depend on each other. In this discussion, we will also overview the syntax of much of GMTK’s parameter file format.

17.1 Collection Objects

Our first basic GMTK object is the named collection object. Simply put, a named collection object is a mapping from integers to strings. Specifically, such an object consists of an ordered list of names of other objects, and is used as a mapping from an integer corresponding to the position in the collection, to the object that has the name given in that position. Collection objects make it much easier to specify indexing by name, rather than having to specify indexing by an integer index.

Name collection objects are used for several purposes.

1. For continuous random variables, they are used as a mapping from the decision tree (DT) integers at the leafs of the DT to a particular Gaussian mixture object. As mentioned in Section 17.2.5, decision trees are used (among other things) to map from a set of discrete parent random variables to a leaf-node integer value. This integer value specifies the index of the Gaussian mixture used for the continuous random variable, under the current set of parent values.

A name collection object is further used to map from that leaf-node integer value to the specific textual name of the Gaussian mixture to use. This makes it much easier to specify decision trees since the absolute index number of the decision tree leaf-node is not needed to determine the specific mixture. Note, therefore, that two decision trees can be shared (see Sections 17.6 and 17.2.3) by two continuous random variable objects that use two different name collection objects. It is the name collection objects that can specify a different set of mixtures, even for the same decision tree.

Note, that there is a special internal name collection object named “global”. When this name is mentioned in a structure file, the decision tree leaf node index maps directly into the global internal array of Gaussian mixtures, rather than indirectly via a collection. This ability makes it easier to rapidly set up a simple system.

2. For sparse conditional probability tables, name collection objects are used to map from the decision tree leaf values to the name of the sparse probability mass function (SPMF) that is to be used for the corresponding leaf value.

In both cases, the use of a name collection makes it much easier to specify other GMTK objects, as otherwise the decision trees would need to keep track of the absolute position in GMTK's object tables of the desired object. If there was a need for different groups of objects (say the first 100 Gaussian mixture was 13 dimensional, and the next 100 was 26 dimensional), not having the name collection ability would mean that the decision tree for the second group would always need to use an offset of 100. The use of name collection means that second group may be specified indirectly.

Note further that the use of a name collection does not require an additional indirection internal to the code. Each of the indirect references specified by a name collection are essentially “compiled out” when GMTK loads the files, rather than calculated each time the indirection is used.

Name collection objects are specified quite simply. The following is an example of how this is done.

```
NAME_COLLECTION_IN_FILE inline
1 % one name collection to follow.
0 % index 0, first collection.

myCol % The name of the collection, 'myCol'
5   % The collection length, only 5 entries.
      % Finally, the list of five names.
      % Note that new line characters are ignored.
object1 object2
object3 object4
object5
```

Note that the white space between the set of object names is ignored.

Here is a more complete example using named collections. The example also uses decision trees (described later in Section 17.2.5).

(CHECK THIS NEXT EXAMPLE FOR BUG WITH 3 rather than 2 object!!)

```
NAME_COLLECTION_IN_FILE inline 2
0   % first
coll1 % The name of the collection, 'coll1'
2   % The collection length, 2 entries will follow.

gm1
gm2
1   % second
col2 % The name of the collection, 'col2'
3   % The collection length, 3 entries will follow.

gm4
gm3
gm5

DT_IN_FILE inline 1
0       % first
map     % name of decision tree, 'map1'
1       % only one parent.
-1 (p0) % just copy value of parent
```

```

variable : obs1 {
    type: continuous observed 0:12 ;
    conditionalparents: foo(0) using mixture
        collection("col1")
        mapping("map");
}
variable : obs2 {
    type: continuous observed 0:12 ;
    conditionalparents: foo(0) using mixture
        collection("col2")
        mapping("map");
}

```

In the example above, two observation variables are defined `obs1` with parent `foo` and `obs2` also with parent `foo`. The decision tree `map` as defined is an identity operator, i.e., it maps whatever is the current parent value to the leave node. The process is as follows. First the value of the parent `foo` is obtained. This value is mapped through a decision tree to an integer index value. This value is used to look up the name in the corresponding position of the name collection object. The name of the Gaussian mixture in that position is then used for the observed variable.

For example, the setup above states that when `foo` is 0, `obs1` uses the Gaussian mixture named `gm1` but `obs2` uses the Gaussian mixture named `gm4`. When `foo` is 1, `obs1` uses the Gaussian mixture named `gm2` and `obs2` uses the Gaussian mixture named `gm3`. When (or if) `foo` is 2, `obs1` would lead to a run-time error, while `obs2` uses the Gaussian mixture named `gm5`.

Name collection objects will be described further in Section 17.2.3 when sparse CPTs are introduced.

17.2 Conditional Probability Tables CPTs

There are a number of ways of representing probabilities in GMTK, depending on if you are using discrete or continuous variables.

The conditional probability table (CPT) is the basic object representing discrete probability values in GMTK. GMTK actually has three different types of CPT objects, dense CPTs (`DenseCPTs`), sparse CPTs (`SparseCPTs`), and deterministic CPTs (`DeterministicCPTs`). The different type of CPTs to use will depend on what function the random variables are mean to perform, and in certain cases (such as language modeling) on the available amount of memory and computation.

17.2.1 Dense CPTs

Lets start with an example of a `DenseCPT`. Suppose, for example, you were interested in representing the quantity $P(X)$ where X is a 10-valued discrete random variable. A structure file might look something like

```

variable : X {
    type : discrete hidden cardinality 10;
    switchingparents : nil;
    conditionalparents : nil using DenseCPT("X_prob_table");
}

```

This declaration of X specifies that there must be a dense CPT that is named `X_prob_table`. This is declared in the object file as follows:

```

1      % The number of DenseCPTs
0      % The index number of the first DenseCPT (zero start)
X_prob_table % the name of the DenseCPT, which is 'X_prob_table'
0      % number of parents, zero in this case.
10     % number of values, or cardinality of the RV using this CPT
% finally, 10 probability values which in this case indicate
% a uniform distribution.
0.1 0.1 0.1 0.1 0.1
0.1 0.1 0.1 0.1 0.1

```

This is an example of a one-dimensional CPT. As X has no parents, the CPT need only specify 10 probability values. The first probability corresponds to $P(X = 0)$, the second to $P(X = 1)$, and so on until $P(X = 9)$ (recall from Section 15.1.1 that all discrete random variables have a possible value starting from 0 to the cardinality minus one). The number of entries in the the CPT is specified in two ways, first by explicitly stating how many values there will be which is then followed by that many values (this is another example of reducing the chance of errors by slightly increasing redundancy in the files). Any random variable that uses this CPT therefore must have two qualities. First, it must not have any parents (as it is a 1-D CPT). Second, the variable must have a cardinality of exactly 10. If either of these conditions do not hold, then a run-time error will occur.

Lets do a 2-D example now. Suppose we declare a variable in the structure file as follows:

```

variable : M { type : discrete hidden cardinality 4; ... }
variable : C {
    type : discrete hidden cardinality 3;
    switchingparents : nil;
    conditionalparents : M(0) using DenseCPT("C_given_M");
}

```

The variable C then assumes that a DenseCPT named `C_given_M` will exist in the parameter files. In this case, C has cardinality 3 but M has cardinality 4 meaning that `C_given_M` must have 12 entries. This is declared as follows:

```

1 % The number of DenseCPTs
0 % The index number of the first DenseCPT (zero start)
C_given_M % name of this CPT
1 % one parent
4 3 % parent cardinality (4 in this case) and self cardinality (3)
% the 4x3 table. For each value of the parent M, we have
% three probabilities for each of the possible child values. This
% means that  $P(C=c|M=m) = A_{mc}$  where  $m$  is the row, and  $c$  is the column,
% and that this is the order the probabilities are given in the table.
0.2 0.4 0.4 % P(C=0|M=0), P(C=1|M=0), P(C=2|M=0)
0.4 0.4 0.2 % P(C=0|M=1), P(C=1|M=1), P(C=2|M=1)
0.3 0.4 0.3 % P(C=0|M=2), P(C=1|M=2), P(C=2|M=2)
0.1 0.1 0.8 % P(C=0|M=3), P(C=1|M=3), P(C=2|M=3)

```

Finally, a 3-D example. We declare a variable in the structure file as follows:

```
variable : F { type : discrete hidden cardinality 4; ... }
variable : M { type : discrete hidden cardinality 2; ... }
variable : C {
    type : discrete hidden cardinality 3;
    switchingparents : nil;
    conditionalparents : F(0),M(0) using DenseCPT("C_given_F_M");
}
```

The variable C assumes that a DenseCPT named `C_given_F_M` exists in the parameter files. In this case, C has cardinality 3, F has cardinality 4, and M has cardinality 2 meaning that the table must have 24 entries.

The order that the parents are declared in the 'conditionalparents' line above determines the order of the parents used to index probability values in the table `C_given_M_F`. The first parent declared (F in this case) is the outer-most index, the second parent (M) is the next index, and so on. The table is declared follows:

```
1 % The number of DenseCPTs
0 % The index number of the first DenseCPT (zero start)
C_given_F_M % name of this CPT
2 % two parents
4 2 3 % parent cardinalities (here, 4 and 2) and self cardinality (3).
% The 4x2x3 table -- for each value of the pair F and M, we have
% three probabilities for each of the possible child values. This
% means that  $P(C=c|F=f, M=m) = A_{fmc}$  where  $f$  is the outer (major) row
% index,  $m$  is the inner (minor) row index, and  $c$  is the column index,
% and that this is the order the probabilities are given in the table.
0.2 0.4 0.4 %  $P(C=0|F=0, M=0)$ ,  $P(C=1|F=0, M=0)$ ,  $P(C=2|F=0, M=0)$ 
0.4 0.4 0.2 %  $P(C=0|F=0, M=1)$ ,  $P(C=1|F=0, M=1)$ ,  $P(C=2|F=0, M=1)$ 
0.3 0.4 0.3 %  $P(C=0|F=1, M=0)$ ,  $P(C=1|F=1, M=0)$ ,  $P(C=2|F=1, M=0)$ 
0.1 0.1 0.8 %  $P(C=0|F=1, M=1)$ ,  $P(C=1|F=1, M=1)$ ,  $P(C=2|F=1, M=1)$ 
0.1 0.5 0.4 %  $P(C=0|F=2, M=0)$ ,  $P(C=1|F=2, M=0)$ ,  $P(C=2|F=2, M=0)$ 
0.9 0.0 0.1 %  $P(C=0|F=2, M=1)$ ,  $P(C=1|F=2, M=1)$ ,  $P(C=2|F=2, M=1)$ 
0.3 0.3 0.4 %  $P(C=0|F=3, M=0)$ ,  $P(C=1|F=3, M=0)$ ,  $P(C=2|F=3, M=0)$ 
0.7 0.2 0.1 %  $P(C=0|F=3, M=1)$ ,  $P(C=1|F=3, M=1)$ ,  $P(C=2|F=3, M=1)$ 
```

The CPT must declare the cardinalities of the variables which are going to use that CPT. These cardinality values serve a dual purpose of also defining the dimensionality of the table, and ensuring that a table is compatible with a set of parent and child random variables values which wish to use that table.

17.2.2 Special Internal Unity Score CPT

Like in the Gaussian case (see Section 17.4.3), GMTK supports a special pre-defined and internal CPT that in effect always produces a unity score no matter what the argument of the distribution might be. This CPT is named `internal:UnityScore` and can be used in place of a Dense CPT name in the structure file. There are restrictions when this can be used, however, and these include:

- The random variable must (of course) be discrete.

- It must be observed.
- It must use a Dense CPT implementation that has no (i.e., `nil`) conditional parents (for a given switching parent value).
- The CPT, however, may be used regardless of the cardinality of the random variable in question.

This special CPT can be useful for a number of models, most importantly for discrete conditional-only observations (similar to the continuous conditional-only features that are available with GMTK's multi-stream BMM-like Gaussians). For example, suppose the discrete random variable O is always observed for a given utterance. One therefor can compute conditional probabilities without seeing any affect of the probabilities of the conditional-only observations. In particular,

$$P(A|O = o)P(O = o) = P(A|O = o)$$

where A is an arbitrary random variable. The equation is true because essentially the CPT produces an effect where $P(O = o) = 1$ for all values o . Note that this does not produce normalization irregularities because the variable O that uses the unity score CPT is not allowed to have any parents anywhere in the network, and it also must be observed.

17.2.3 Sparse CPTs

In this section, you will learn about SparseCPT objects, a way of representing a CPT in a sparse way.

As defined above, a CPT (conditional probability table) is a table representation of a discrete conditional probability distribution. Suppose you are interested in representing the discrete distribution

$$p(x_1|x_2, x_3, \dots, x_N)$$

where each of the random variables X_1, \dots, X_N are discrete. For this discussion, lets assume that all the random variables can have the same number of values (cardinality), the integers in the range between 0 and $K - 1$ (so there are K values in total for each variable). Therefore, the table above requires a total of K^N entries. For each set of values of the parents X_2, \dots, X_N (and there are K^{N-1} possible values), there are an additional K values needed for X_1 leading to K^N . If you were to represent this table using a DenseCPT, this could require a pretty large table, too big to store in memory (depending on the values of K and N). Furthermore, if you knew that for many parent values, and values of x_1 , the probabilities are zero, it would be wasteful to store all the values (not to mention the fact that when doing inference, GMTK would need to iterate over all such values to check which ones are zero).

SparseCPTs solve this problem. A SparseCPT is a representation of $p(x_1|x_2, x_3, \dots, x_N)$ where zeros are not represented. In other words, a SparseCPT is a way of representing a sparse singly-stochastic arbitrary-dimensional matrix (a matrix where one of the dimensions has “rows” which are probability mass functions). These individual mass functions have many entries which are zero. Sparsity is achieved in a SparseCPT using the following two methods.

1. A decision tree is used to map from the set of parent values X_2, X_3, \dots, X_n to integers. You can think of x_2, x_3, \dots, x_N as an $(N - 1)$ -dimensional discrete vector space. A decision tree carves this space up into a set of arbitrarily shaped regions of dimension $(N - 1)$ or less. Each region is then mapped to some integer corresponding to the leaf nodes of a decision tree. Note that the regions need not be connected – this can be done by having multiple leaf nodes in the decision tree (Section 17.2.5) specify the same integer.

2. The integer results from 1) are used to index into a table of sparse 1-dimensional probability mass functions (SPMFs). An SPMF (see also Section 17.3.2) represents only the values of X_1 that have non-zero probability (for a given set of values of the parents x_2, x_3, \dots, x_N). The zero probability values of x_1 are not represented, and therefore don't take up any resources.

Of course there is a little bit more to do than this implies, since we need to define the mappings, and how to create SPMFs. Also, SparseCPTs allow any of the SPMFs to be shared, and moreover, they allow different SPMFs to share the same underlying set of probabilities even if they correspond to different values of the random variable (this can also occur during EM training, and therefore could help to mitigate the effects of data sparsity issues).

Lets start off with SPMFs. An SPMF is a representation of a 1-dimensional probability mass function, where presumably many of the entries in this mass function are zero. Rather than using an array of probabilities with lots of zeros, an SPMF consists of two arrays. The first array contains the set of possible (i.e., non-zero probability) values of the random variable (the possible non-zero probability values of X_1 above) and the second array contains the probabilities themselves.

In the master parameter file, an SPMF can be specified as follows:

```
% SPARSE PMFs
SPMF_IN_FILE inline % SPMF keyword "SPMF_IN_FILE", and "inline"
% to indicate that they are contained right
% here rather than in another file.

2 % the number of SPMFs that will follow (only two in this case)

0           % SPMF index number 0 in this SPMF set.

spmf_name0 % SPMF_NAME: the name of this SPMF

42          % CARD: The cardinality of this SPMF (42 in this case). This
% means that the random variable that this
% SPMF is used for (e.g., X1 above) has 42 possible
% values (including the values which might
% have zero probability). Note that in this case
% the values are integers in the range of 0 to 41.

16          % LENGTH: The length of this SPMF. This number gives
% the number of non-zero random variable values
% that this SPMF specifies. In this case,
% it is saying that there are only 16 of the
% 42 values which do not have zero probability.
% Note that in some cases it might not be worth
% using a SPMF if LENGTH >= CARD/2 since
% a SPMF requires two tables.

          % TABLE: The actual set of values that this random variable
0   1   2   3 % might take on. This means that
5   6   7   8 % p(X1=4) = p(X1=9) = p(X1=14) = p(X1=19) = 0
10  11  12  13 % and that any other value not specified
15  16  17  18 % in the table (namely 19-41) also has
```

```

% zero probability. Note that a value
% can not be less than zero, nor can one
% be greater (in this case) than 41.
% Note that the fact that the values here are arranged
% in a 4x4 matrix has no significance. The values
% can be arranged in all one row, all one column, or
% anything in between.

dense_pmf
% DPMF_NAME: the name of the DPMF (dense 1-D PMF)
% which holds the 16 probability values
% which are used for this SPMF and are associated
% with each of the values in TABLE respectively. This name
% can be any of the DPMF objects which
% have been defined earlier. DPMFs also
% have a cardinality, and in this
% case the DPMF cardinality must be equal to 16 (not 42)
% or you'll get an error message.

1
% SPMF index number 1. The next SPMF
% was actually used for a language
% modeling experiment. This is (a little) more
% like the SPMFs that you are likely to encounter
% when you start using SPMFs for real problems.
% Of course, you probably would want to generate
% it by script.
20000 604 % This r.v. has a card of 20000, and the length is 604.
% What follows are the 604 (well almost) non-zero prob. values.
3 4 5 6 7 18 21 28 29 41 71 194 195 202 203 210 211 269 270 348 362
368 417 418 419 424 436 470 471 472 480 481 492 493 495 496 528 546
...
13108 13109 13110 13116 13117 13118 13119 13159 13161 13166 13189
13220 13248 13250 13251 13252
_COMMA % finally the name of the corresponding DPMF, named '_COMMA'

```

Several SPMF objects can share the same DPMF object by specifying the same DPMF name. This is how parameter sharing of DPMFs by SPMFs are specified. During EM training, the contributions of both SPMF objects will increase the counts of the DPMF.

Also note, that the DPMF objects are exactly the same objects use for the probability coefficients of Gaussian mixture distributions. Therefore, a Gaussian mixture object can share its mixture DPMF with an SPMF. If you find a use for this (for some reason) make sure that you are aware that any Gaussian component vanishing and splitting will change the cardinality of the DPMF at which point it will no longer match the SPMF, and you'll get a run-time error.

Ok, so that describes SPMFs. We said earlier that decision tree leaf node integers map to the index entries in a large table of SPMFs. Entire SPMFs are shared in this way by having regions in x_2, \dots, x_N space refer to the same integer (and thereby an SPMF index).

Internal to GMTK, there is a global table of SPMF objects. In a master file, each time GMTK encounters a line of the form:

```
SPMF_IN_FILE inline
```

...

or of the forms

```
SPMF_IN_FILE filename.spmf ascii
SPMF_IN_FILE filename.spmf binary
```

it will append any SPMFs that follow (either inline, or the ones contained in filename.spmf) to this global array of SPMF objects (this is the same way all GMTK objects work actually).

In order to have the decision tree integer indices of a SparseCPT specify particular SPMF objects, it must somehow specify the objects in this table. Rather than having these indices index directly into the global table (which would be difficult to manage when there are multiple sets of SPMF objects around), a SparseCPT uses a "name collection" object.

A name-collection object is just an array of names. If a collection gives a list of SPMF names, then a SparseCPT can use that collection. The SparseCPT does so by having its decision tree integer indices index directly into the collection. It therefore doesn't need to worry about the global index locations of the SPMF objects (there is, however, a special internal collection that is called "global" which you can use to index into the global table). Each SparseCPT can have its own collection, which makes managing the indices relatively easy.

So, that is basically it. Lets summarize in a simple ASCII-graphic tree, the objects that are required to specify a SparseCPT.

This will ultimately be a figure

```

SparseCPT
|
+-- the SparseCPT's parameters
|   1) number of parents (e.g., N-1 in example above)
|   2) cardinalities of each parent (e.g., for vars X2, ..., XN above)
|   3) cardinality of self (e.g., for var X1)
|
+---- Decision tree
|       A standard definition of a DT
|
+---- Collection object
|       (A collection object that gives a list of SPMF names.
|       Via these SPMF names, it refers to the following
|       objects:).
|
+---- SPMF1 % the first SPMF referred to by the collection
|       |
|       +-- SPMF1's parameters
|           1) the cardinality of self (e.g., for var X1)
|           2) the length L (number of non-zero probabilities)
|
|       +-- DPMF
|           |
|           +--- The DPMF's parameters
|               1) the length L (specified again)
```

```

|           |     +--- a list of L probability values
|           |
|           +-+ list of L integer values giving the
|                   integers values of the random variable that
|                   do not have zero probability.
|
+----+ SPMF2
|           |
|           + Same as above, but for SPMF2 ...
|
+----+ SPMF2
|           +
...
|
+----+ SPMFM (the last SPMF specified by the collection)
|           +
...

```

Lastly, here is a complete example containing portions of all the definitions needed to set up a SparseCPT. These file portions will live in your master file, except for the final DPMF objects which are trainable and which could live either in a master file or a trainable file (a file containing objects which might change as a result of EM training).

First, the SparseCPT object itself:

```

SPARSE_CPT_IN_FILE infile

1 % one Sparse CPT

0 % index number

mySparseCPT % name of 1st SparseCPT

2 % there are two parents, so this is a
% SparseCPT for P(X1/X2,X3)

2 3 % cardinalities of parents. So the variable
% X2 is binary valued (value 0 or 1), and
% and the variable X3 is ternary valued
% (values 0, 1, or 2).

5 % Cardinality of self, so X1 is a
% quinary random variable (one with
% five values)

myDT % the name of the decision tree that
% this SparseCPT will use.

myCol % the name of the collection used by
% this SparseCPT

```

Next, we'll give the definition of the decision tree. See section 17.2.5 for more details on decision tree syntax.

```
DT_IN_FILE inline

1 % one decision tree
0 % index number

myDT    % The name of the decision tree

% There are 6 possible values of the X2,X3.
% This decision tree will map [redacted]
% (X2=0, X3=(0,1)), (X2=1, X3=(2)) => the first SPMF
% (X2=0, X3=2), (X2=1, X3=(0,1)) => the second SPMF

2      % two parents, must be same as mySparseCPT.
% The following line with '0 2 0 default' has four values meaning:
% (parentSpecifier=0, specifying X2), (numSplits=2), (split 1) (default)
0 2 0 default
  1 2 2 default
    -1 1 % second SPMF
    -1 0 % first SPMF
  1 2 2 default
    -1 0 % first SPMF
    -1 1 % second SPMF
```

Here is the definition of the collection. It is quite simple as it only has two entries, one for each of the SPMF objects.

```
NAME_COLLECTION_IN_FILE inline

1 % one collection
0 % index 0

myCol % Collection name.
2      % length, only two entries.
      % The length-two set of spmf names.

mySpmf0
mySpmf1
```

Note that the names "mySpmf0" and "mySpmf1" refer to the global set of SPMFs. In other words, if another collection of SPMFs used the same names, then they would refer to the same SPMF objects (this is one way, in fact, to achieve parameter sharing across different SparseCPTs).

Next come the definitions of the two spmfs.

```
SPMF_IN_FILE inline
```

```

2 % two SPMFs

0 % First spmf.
mySpmf0 % SPMF name.
5 % Cardinality of X1.
3 % Number of values with non-zero probability.
0 1 2 % So in this case, X1 can only take values 0 1 or 2
        % with non-zero probability.
myDpmf0 % Name of corresponding length 3 DPMF.

1 % Second spmf.
mySpmf1 % SPMF name.
5 % Cardinality of X1.
2 % Number of values with non-zero probability.
0 4 % So in this case, X1 can only take values 0 or 4
        % with non-zero probability.
myDpmf1 % Name of corresponding length 2 DPMF.

```

And finally, the definition of the two DPMFs needed above.

```

DPMF_IN_FILE inline

2 % two DPMFs

0 % First DPMF.
myDpmf0 % Name.
3 % DPMF length.
0.25 0.5 0.25 % The three probability values.

1 % Second DPMF.
myDpmf1 % Name.
2 % DPMF length.
0.25 0.75 % The two probability values.

```

17.2.4 GMTK's Deterministic Integer Maps: Decision Trees and C++ Integer Maps

A deterministic map is a function $f : \mathbb{Z}_+^K \times \mathbb{Z}$ from K non-negative integers to a non-negative integer. We use the term “deterministic” to mean “non-random”, in that for such an f , $f(i_1, i_2, \dots, i_K)$ takes on only one value with probability one. Such mappings are used for a variety of different purposes in GMTK. These include:

1. Deterministic CPTs, which are used when for each set of values of a collection of K parent random variables, there is one and only one possible value of the child variable having non-zero probability.
2. Sparse CPTs (§17.2.3) to specify arbitrary regions in the space of possible values of a collection of K parent variables, and map from those regions to SPMFs. Essentially the decision tree maps to a particular row entry in the CPT.

3. To map from a collection of K discrete hidden parent random variables to a value for a child observation variable. The value for the child is then used to determine a Gaussian mixture object for that particular combination of parent values indirectly via a named collection §17.1.
4. To map from the values of a collection of K switching parent variables to one of a set of conditional parent variables. Thus, even when there are many switching parents, there can be only a small number of distinct switching consequences.

In cases where a deterministic map is used to produce values of child random variables, the ap must make sure to never produce a value that is out of range of the child RV, and this must hold for all parent random variable values. For example, given a deterministic map used to implement a deterministic CPT $p(a|b, c)$, where $p(a|b, c) = \mathbf{1}\{a = f(b, c)\}$ where $f()$ is a map, it must be the case that that $0 \leq f(b, c) < \text{cardinality}(a)$ for all b, c .¹

These uses are elaborated upon further in Section 17.2.5.4.

There are two ways that deterministic map's can be specified and used in GMTK, the first is with decision trees where leaf nodes may have arbitrary integer formulas, as described in Section 17.2.5. The second way is using internally compiled C++ functions, as described in Section 17.2.6. The first way is useful to quickly specify and prototype deterministic mapping when developing a model. When using an already debugged model, it can be significantly faster to use internally compiled C++ mappings since decision trees are only interpreted and can be slow.

17.2.5 Decision Trees with Leaf Integer Formula

Like all parameter objects, the decision tree syntax was designed for easy and fast parsing, while including redundancy so that the chance reading in an erroneous decision tree is reduced.

In general, a decision tree is used in GMTK as a deterministic map, to map from a values of a collection of discrete random variables down to a single integer value. That integer value is then used as appropriate for whatever function the decision tree is needed. Decision trees are regular GMTK objects, defined in the same way as all GMTK objects, and as described in Section 16.2.

In general, a decision tree is a tree where each node of the tree contains or uses some query, and the answer to the query at each node determines which single branch of the tree to descend. Note that there are two parts to any query: 1) the object (or set of objects) that are being queried, and 2) the actual question about that object. This is important since the GMTK decision tree syntax separates the two. At the decision tree leaf nodes, the final “answer” of the decision tree (i.e., the set of questions) are found. In a GMTK decision tree, all leave nodes after evaluation have as their answer a single integer.

A general recursive definition of a GMTK DT (decision tree) is given in the following:

```

DT := 
% A decision tree is either a leaf node ...
-1 integer_formula % leaf nodes start with '-1'
% ... or is a branching node, a recursively defined decision tree
| parent_id N split_range_1 split_range_2 ... split_range_{N-1} "default"
% parent_id says which parent to query. N = number of splits

```

¹If this wasn't the case, some strange behaviors might occur. For example, the combination of triangulation and approximate inference is such that certain combinations of random variable values might exist for one triangulation but not for another triangulation. For some triangulations, it might be the case that indeed $p(f(b, c) > \text{cardinality}(a)) = 0$, but during a partial computation $f(b, c) > \text{cardinality}(a)$, and this would cause a runtime error. In other triangulations this might never happen. In other words, one might never see an error for some triangulations (because it is pruned away before it happens) but for others there might be an error. Hence, we avoid this oddity by imposing the said restrictions.

```

DT_1           % DT_1 used if split_range_1 is true.
DT_2           % DT_2 used if split_range_1 is true.
...
DT_{N-1}
DT_{default}   % DT_{default} used if no split ranges are true.

```

This means that a decision tree is either a leaf node (“-1” followed by an integer formula, which is often just a constant) or alternatively is a branching node.

In the case of a branching node, `parent_id` is an integer in the range $0, 1, \dots, K - 1$ giving the index of the parent (or feature using the terminology of decision trees) to query, where K is the total number of possible parents. This means that if the decision tree is used with a random variable having K parents, then `parent_id` specifies which parent should be queried. The `parent_id` specifies what the query at a given DT node is about, but does not specify the actual set of questions which are used to determine which branch to descend. The questions are determined by the range specifications, as described below.

N is the number of splits in that node in the tree. This means that a GMTK decision tree can have any branching factor at each node. The string `split_range_i` is an integer range specification, and it specifies an integer range of values to use for that branch in the tree. A given integer range is really a syntax for specifying a set of integers, and if the parent (as given by `parent_id`) has a value that lies within a given range’s set, then the answer to that particular question is yes.

An integer range is either a list of comma separated integers, or a matlab-like range specification (see Section 18.1). The integers specified, however, must be non-negative. The “`default`” string is the catch-all case, which is used if all of the split ranges fail.

The `DT_i` tags are recursively defined sub-decision trees having exactly the same syntax: they can either be another decision tree or can be a leaf node. There must be as many `DT_i` sub-decision trees as there are splits (N in the above). The last tag `DT_default` is used for the catch-all case.

In the case of a leaf node, the special value “-1” specifies this is the case, and `integer_formula` gives the integer formula corresponding to that leaf node. An integer formula may just be a simple solitary and isolated integer (giving the value of the leaf node), and this must be an immediate constant. Alternatively, a leaf formula may be a integer formula (enclosed in curly braces) and involve most standard binary operators as well as a number of built-in integer functions (e.g., `median`) — in this later case, the formula may also contain any of the special variables p_i (parent i ’s value), c_i (parent i ’s cardinality), or m_i (parent i ’s value multiplied by its cardinality). Integer formula are fully defined in Section 17.2.5.2.

Finally, a decision tree must be given a **name** (any textual string, like all GMTK objects). Following the name must be an integer that specifies the maximum **number of parents** (also called number of “features” in decision tree lingo) that the decision tree might use. In GMTK, ‘features’ and ‘parents’ are used synonymously when used in the context of decision trees, meaning that queries made of a DT feature is really made on parent random variable values. For a GMTK DT, it is only parent random variable values which can be the inputs to DTs. Note that the number of parents of a decision tree should therefore match the number of parents that a random variable has when it tries to use that decision tree. Typically, decision trees have a number of possible features (i.e., multiple parents) that they can query. In GMTK, the different set of instantiated random variable parent values exactly constitute the set of features that the decision tree can query.

Note also that the number of parents specified with a decision tree (K above) must match the context in which it is used. If it is attempted to use a DT with a different number of parents than the number of parents in a CPT (or set of random variable parents), then a GMTK run-time error will occur.

Lets give a few examples, starting with the easiest of decision trees, one that returns a constant.

myDT 1 % name of the DT ‘myDT’, and its number of parents (one)

```
-1 0
```

The decision tree first specifies its name myDT, and then the number of parents (one, in this case). Then, there is only one leaf node (specified by the “-1” tag), and the value of the leaf node is the immediate constant value 0 regardless of any of the parent values.

For our next example, we copy the value of the first parent to the child, ignoring the value of the second parent.

```
myDT2 2 % name of DT 'myDT2', and number of parents (two)
-1 { p0 }
```

Here, there are two parents. The leaf node is the value p0 surrounded in curly braces, saying that the value of the decision tree is whatever is the current value of the 1st parent (i.e., parent zero). This first parent corresponds to the left most parent in the corresponding structure file. Curly braces are used to distinguish the case that an integer formula being used as a leaf node. If the leaf node is just a constant immediate integer, however, curly braces are not needed.

The next example implements the function that adds one to the first parent if its value is less than 10 but otherwise just copies that parent value.

```
myDT3 1 % name 'myDT3', and 1 parent.
0 2 0:9 default % query parent 0, two splits '0:9' and 'default'
-1 p0+1          % if first split criterion true, go here
-1 p0            % else, if second split criterion true, go here
```

This decision tree says that if the first parent (parent 0) is in the range of values between 0 and 9 inclusive, then the result of the decision tree is { p0+1 }, and is otherwise just { p0 }.

As another slightly richer example, suppose there are three random variables A , B , and C all discrete, and we wish to specify the following mapping from jointly A, B, C to a single resulting integer.

If $A = 0$, and $C = 0$ then return 0.

If $A = 0$, and $C \neq 0$ then return 1.

If $A = 1$, then return 2.

If $A \neq 0$, $A \neq 1$, and $B = 1$, then return 3.

If $A \neq 0$, $A \neq 1$, and $B \neq 1$, then return 4.

This can be done using a decision tree as follows:

```
myDT4 3 % name 'myDT4', 3 parents, parents 0,1,2 are A,B,C respectively.
0 3 0 1 default % parent 0 (A), 3 splits: '0', '1', and 'default'
% if A = 0
  2 2 0 default % parent 2 (C), 2 splits, '0' and 'default'
    -1 0          % leaf node returning 0
    -1 1          % leaf node returning 1
% if A = 1
  -1 2           % leaf node returning 2
% if A is not 0 or 1
  1 2 1 default % parent 1 (B), 2 splits, '1' and 'default'
    -1 3          % leaf node returning 3
    -1 4          % leaf node returning 4
```

The next example queries the 2nd parent first, and if its value is 0 returns the value of the 3rd parent. Otherwise (i.e., if the 2nd parent does not have value 0), then the decision tree queries the 1st parent. If that first parent has value 1 then the tree returns the value 0, otherwise (if the 1st parent does not have value 1) then the tree returns the value of the 3rd parent plus one.

```
myDT5 3 % name 'myDT5', 3 parents
1 2 0 default
    -1 p2
    0 2 1 default
        -1 0
        -1 p2+1
```

Note that the following decision tree is not valid:

```
3 % number of parents
0 2 p1 default % query parent 0, 2 splits,
    -1 p2
    -1 0
```

The reason this will not work is that split ranges must be constant. Instead, one can get such a construct just as easy using:

```
3 % number of parents
-1 (p0 == p1) ? p2 : 0
```

using the integer formulas which are defined in Section 17.2.5.2.

There are a few things worth noting when specifying decision trees that use a long list of split ranges. Basically, whenever you have a construct like:

```
0 16 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 default
    -1 50
    -1 39
    etc.
```

Then this ascending sequential order will be fast since internally, GMTK will use direct indexing. In fact, this form of sequence of splits is encouraged, and there is a short cut for this that one can use and that has the identical meaning:

```
0 16 0 ... 14 default
    -1 50
    -1 39
    etc.
```

In other words, the ellipsis “...” is a syntactic shortcut that allows one to avoid specifying the intervening integers between 0 and 14. In fact, if there is any long sequential strings of numbers, this will both be loaded in faster and will use direct indexing. The starting value can be anything, so one could do:

```
0 16 5 ... 19 default
-1 50
-1 39
etc.
```

There is a benefit to using single integers and ascending order. In other words, this construct:

```
0 4 0 1 2 default
-1 50
-1 39
etc.
```

will be the same speed as this one:

```
0 4 0 ... 2 default
-1 50
-1 39
etc.
```

which will be faster than this one:

```
0 4 1 0 2 default
-1 39
-1 50
etc.
```

which will be still faster than this one:

```
0 4 1:1 0:0 2:0 default
-1 39
-1 50
etc.
```

which uses range objects. I.e., even though the four preceding decision tree constructs are identical, the first two use direct mappings (and are fastest), the second is implemented via a hash table, and the third case uses the more general range objects (and is slowest). Depending on how large the decision tree is, and on the context in which this decision tree lives (i.e., the graphical model, the state space, etc.), these differences can be significant.

17.2.5.1 Lists of decision trees: decision tree files

A decision tree file consists of a list of decision trees. The first number in a DT file indicates how many DTs there are, and each DT is preceded by its relative order in the file (starting at number zero for the first DT, one for the second, and so on). This is the same as for all GMTK objects as described in Section 16.2. Here is an example of a DT file:

```
% this is a DT file, typically has extension .dt or .dts
3 % there are three decision trees in this file

0 % the first DT
the_first_DT % this is the name of the DT
1           % one parent
0 0:9 default
-1  p0+1
-1  p0

1 % the second DT
foo_DT % this is the name of the second DT
1           % also one parent
-1 { p0 }

2 % the third DT
bar_DT % this is the name of the third DT
3 % 3 parents in this case
0 3 0 1 default
    2 2 0 default
        -1 0
        -1 1
    -1 2
    1 2 1 default
        -1 3
        -1 4
```

17.2.5.2 Leaf Node Integer Formula

As mentioned above, decision tree leaf nodes may consist of an integer formula, and they can be quite rich, for example consider the following:

```
2
0 1 default
    -1 { p1<12 ? % start of leaf node formula
            min(p0*3, c0-1) : % first expression
            mod(p1,3)==2 ? % 2nd expression is itself conditional
                3 :
                4
    }
```

The leaf node formula uses two nested C-language-like conditional expression operators, and is identical to the following pseudocode:

```
if (p1 < 12)
then
    return min(p0*3, c0-1);
else
```

```

if (mod(p1, 3)==2)
then
    return 3;
else
    return 4;
end
end

```

&	bitwise “and”
	bitwise “or”
~	bitwise “not” (unary)
. /	divide and always round up, same as ceil_divide(a,b)
/	divide and always round down, same as /. and floor_divide(a,b)
/ .	divide and always round down, same as / and floor_divide(a,b)
~ /	divide and round to the nearest integer, same as round_divide(a,b)
==	equals
!=	not equals
^	exponent
>	greater than
>=	greater than or equals
<	less than
<=	less than or equals
&&	logical “and”
	logical “or”
-	minus
!	logical not (unary)
+	plus
<<	shift left
>>	shift right
*	times
a?b:c	conditional expression, (a ? b : c) evaluates to b if a is true (greater than zero), and c if a is not true (equal to zero)

Table 17.1: Available operators in decision tree leaf formula.

There are a large set of constructs that can be part of an integer formula. These include a variety of operators (see Table 17.1), functions (see Table 17.2), and variables (see Table 17.3).

17.2.5.3 Per-Segment Decision Trees

Sometimes, when building a mapping from one set of variables (say A,B,C) to another variable (say D), there is one and only one DT for all times that is used to implement that mapping. So that DT is specified once, and we’re done.

Other times, the DT mapping will need to change once for each observation segment. For example,

<code>abs(a)</code>	absolute value
<code>ceil_divide(a,b)</code>	a divided by b and always round up, same as <code>. /</code>
<code>floor_divide(a,b)</code>	a divided by b and always round down, same as <code>/.</code> and <code>/</code>
<code>round_divide(a,b)</code>	a divided by b and round to the nearest integer, same as <code>/</code>
<code>max(a,b,...)</code>	maximum of a set of elements, can take an arbitrary number of arguments.
<code>median(a,b,c,...)</code>	median of a set of elements
<code>min(a,b,c,...)</code>	minimum of a set of elements
<code>mod(a,b)</code>	a modulo b
<code>allDiff(a,b,...)</code>	True if all values are different from each other
<code>rotate(value,num,pos,length)</code>	bitwise rotate value by num bits, only the length bits starting from pos are rotatated, and the rest left untouched
<code>xor(a,b)</code>	bitwise exclusive “or”

Table 17.2: Available functions in decision tree leaf formula.

during training of an ASR system, we know the training set transcription (i.e., what was said, we have both a speech waveform and the string of words indicating precisely the utterance that was spoken). DTs are used to implement the mapping between variables that encodes the transcription for each utterance (segment). For example, the DT might specify the set of phones that corresponds to the utterance.

Here is a toy example. Suppose we have two possible utterances “CAT” and “DOG”, where we use the transcriptions /C/ (phone 0, i.e., phone zero), /A/ (phone 1), /T/ (phone 2) for “CAT” and /D/ (phone 3), /O/ (phone 4) and /G/ (phone 5) for “DOG”. In the structure file, suppose that there is one random variable Q_t that indicates the current phone for each time t . Q_t must evolve over time in a way that traces out the appropriate transcription of the current utterance.

For “CAT”, a decision tree might look something like:

```
cat_DT 1 % name, and number of parents (=1)
0 2 0 default
    -1 1
    -1 2
```

which implements the mapping:

C → A	A → T
-------	-------

But for DOG, we might have a DT that implements:

```
dog_DT 1 % name, and number of parents (again =1)
0 2 3 default
    -1 4
    -1 5
```

which implements the mapping:

D → O	O → G
-------	-------

parent_X	value of parent number X
pX	value of parent number X
parent_plus_one_X	value of parent X plus one
ppoX	value of parent X plus one
parent_minus_one_X	value of parent X minus one
pmoX	value of parent X minus one
cardinality_parent_X	cardinality of parent number X
cpx	cardinality of parent number X
max_value_parent_X	maximum value (cardinality-1) of parent X
mpX	maximum value (cardinality-1) of parent X
cardinality_child	cardinality of child
cc	cardinality of child
max_value_child	maximum value (cardinality-1) of child
mc	maximum value (cardinality-1) of child

Table 17.3: Available variables (actually symbolic constants) in decision tree leaf formula. Note that there are often two versions, a long version (such as `cardinality_child`) and a short variant (e.g., `cc`). Also, “parent” refers to the parent number when the decision tree has features that correspond to parent, but more generally, can be actual features in a decision tree.

There is a special syntax for decision trees that need to change for each utterance, and is given in the following master file (note the keyword `DT_IN_FILE` which introduces an inline section containing one decision tree).

```
DT_IN_FILE inline 1
% a DT to map from a word counter to a particular word.
% This takes its implementations from a file named 'transcription.dts',
% each utterance will have a different DT implementation.
Q_DT      % name of DT 'Q_DT'
counterToWordMap.dts  % file to take DT implementation.
```

In this case, the file `counterToWordMap.dts` must contain as many decision trees as there are utterances in the utterance file. The i^{th} decision tree is used for the i^{th} utterance. If there were two utterances, then this decision tree file might look like:

```
2 % number of decision trees
0 % DT 1
cat_DT 1 % name, number of parents

0 2 0 default
-1 1
-1 2
1 % DT 2
dog_DT 1 % name, number of parents
0 2 3 default
-1 4
-1 5
```

Note that in a per-utterance decision tree file, one may use the same name for all of the decision trees. For better error message reporting, however, it is advisable to use different names for each such DT.

Note that accessing capability a set of segments is normally random, so that we can jump between any segment and any other segment. Per-segment decision tree files, however, are an ASCII sequence of decision trees. In order to have random access over a list of decision trees, an **index file** (which contains the starting position in a decision tree file of each decision tree) must be built, and for that purpose there is the program `gmtkDTindex`.

This program typically generates a file with the same name as the decision tree file but with the extension `.index`. In case this file does not exist, run `gmtkDTindex -help` which displays the options. Normally, one gives it a list of decision tree files that are in need of an index.

17.2.5.4 Uses of Decision Trees

Now that we are comfortable with the syntax of decision, we elaborate upon the uses of decision trees that were sketched earlier in this section.

1. DTs are used to map from a collection of hidden variables to a particular Gaussian mixture. Suppose we are given the following structure:

```
variable : obs {
    type: continuous observed OBSERVATION_RANGE ;
    switchingparents: nil;
    conditionalparents: wholeWordState(0) using mixture
        collection("global")
        mapping("directMappingWithOneParent");
}
```

In this case, `obs` is an observation variable which has a hidden parent `wholeWordState`. It uses a DT named `directMappingWithOneParent` to map from values of `wholeWordState` to an index into the global (program wide) collection of Gaussians for the entire ASR system.

2. DTs are used to map to a specific row entry in a sparse CPT. Suppose we are interested in implementing $P(A|B, C, D)$. Sometimes CPTs can be sparse, which means that for each value of the tuple B, C, D there might be only a small subset of possible values for A which have non-zero probability. E.g., suppose that when $B = 1$, the event $A = 2$ is impossible.

A sparse CPT uses a DT to map from regions in the tuple (B, C, D) to sparse 1-dimensional arrays that give the possible values of A that have non-zero probability.

A DT is used to map from the collection of parent random variables to an integer which then maps indirectly to a Sparse CPT (i.e., a multi-dimensional Sparse CPT).

Sparse CPTs are described in Section 17.2.3.

3. DTs are also used to map to a specific collection of conditional parents from a set of switching parents. Suppose we have the following structure segment:

```
variable : word {
    type: discrete hidden cardinality 100 ;
    switchingparents: wordTransition(-1)
```

```

        using mapping("directMappingWithOneParent");
        % if there was a word transition, use the bigram, otherwise,
        % this is a copy of previous self
    conditionalparents:
        word(-1) using DeterministicCPT("copyMTCPT")
        | word(-1) using DenseCPT("wordBigram");
}

```

Here, `wordTransition(-1)` is a switching parent. We use the DT named `directMappingWithOneParent` to map from values of `wordTransition(-1)` to either 0 or 1 (since there are only two sets of conditional parents) indicating if either `word(-1)` should be used as a parent (with `DeterministicCPT("copyMTCPT")` as the CPT), or if `word(-1)` should be used as a parent with `DenseCPT("wordBigram")`; as the CPT. Note that switching parents need not just switch the parents, but they can also switch the CPT that is used for the parents. In the example above, the parent `word(-1)` is the same, but the CPT used for that parent switches. Another example might switch the actual set of parents.

- Finally, DTs are used to implement deterministic dependencies between variables (when A is a deterministic function of B). Here a DT is used in conjunction with a `DeterministicCPT`. Given the following structure fragment:

```

variable : wholeWordState {
    % number of words X number of states per word
    type: discrete hidden cardinality NUM_WHOLE_WORD_STATES;
    switchingparents: nil;
    conditionalparents: word(0), wordPosition(0) using
        DeterministicCPT("wordWordPos2WholeWordState");
}

```

Here, we have a variable `wholeWordState` which is a deterministic function of parents `word(0)`, `wordPosition(0)` using a `DeterministicCPT` implementation `wordWordPos2WholeWordState` which in turn uses the DT that lives in the file `wordWordPos2WholeWordState.dts` and is named `wordWordPos2WholeWordState`. Deterministic CPTs are described in Section 17.2.8.

17.2.6 Compiled C++ functions as deterministic maps

In cases where decision trees are too slow (which can happen when the logic that is needed for the deterministic mapping is complicated), it is better to use a compiled C++ function to implement the mapping. There are several ways to do so.

For the first way, the user implements a
Still to write this section.
dynamically linked case.

17.2.7 Built in deterministic mappings

A number of deterministic mappings have been so widely used in practice, they have been built into GMTK and can be used just by referring to them by name. Those names of the ones that are built-in always start

internal:copyParent	always copies its single parent
internal:alwaysZero	always returns zero
internal:alwaysOne	always returns one
internal:increment	return one more than the value of its parent (i.e., return $p_0 + 1$)
internal:decrement	return one less than the value of its parent, saturated at zero (i.e., return $p_0 > 0 ? (p_0 - 1) : 0$)
internal:conditionalIncrement	if p_1 is true, return one more than p_0 and otherwise return p_0
internal:conditionalDecrement	if p_1 is true, return less (saturated at zero) than p_0 and otherwise return p_0
internal:conditionalLimitedIncrement	return one more than p_0 if p_1 and $p_0 < p_2$ and otherwise return p_0
internal:conditionalLimitedDecrement	same but for decrement
internal:allParentsEqual	returns true if all parents are equal to each other
internal:allParentsUnEqual	returns true if all parents are not equal to each other

Table 17.4: Built in deterministic integer map functions available for use in GMTK.

with the “internal:” prefix. They are described in Table 17.4.

17.2.8 Deterministic CPTs

After having mastered sparse CPTs, deterministic CPTs are quite easy. As specified in Section (NOT-YET-WRITTEN), it is quite beneficial (both for computational and memory reasons) for relations between certain random variables in GMTK to be specified deterministically, rather than producing large tables containing an overwhelming majority of zeros.

A deterministic CPT object uses a decision tree to specify the deterministic relationships between sets of parent random variables and a child “random” variable. A deterministic CPT is just another GMTK object which specifies how many parents the CPT has, the cardinalities of the parents and the self, and then gives a reference to an existing DT which implements the dependencies.

Here is an example of a deterministic CPT contained in a master file:

```
DETERMINISTIC_CPT_IN_FILE inline 1
0          % first DETERMINISTIC_CPT
myDETCPT % name of the CPT
1          % num parents of the corresponding random variable
2 3        % cardinalities. cardinality of parent = 2, of self = 3
myDT      % The name of an existing DT to implement the deterministic
          % mapping
```

In this example, the CPT specifies its order in a list (the list has only one CPT in this case), its name ‘myDETCPT’, the number of parents corresponding to this CPT (for example, this could implement a CPT for say $P(B|A)$), the cardinalities of the parents and self (meaning, for example, that A is a binary random variable and B is a ternary random variable), and finally giving the name ‘myDT’ referring to a decision tree that implements the dependency.

The next example is a deterministic CPT which more than one parent. Suppose we wish to create a dependency of the form $P(D|A, B, C)$. This can be declared using the following structure fragment:

```

variable : A { type: discrete hidden cardinality 2 ; ... }
variable : B { type: discrete hidden cardinality 3 ; ... }
variable : C { type: discrete hidden cardinality 4 ; ... }
variable : D {
    type: discrete hidden cardinality 5 ;
    conditionalparents: A(0),B(0),C(0) using
        DeterministicCPT("D_given_ABC"); }
```

The deterministic CPT `D_given_ABC` could be declared as follows:

```

D_given_ABC      % name of the CPT
3                % three parents
2 3 4 5         % cardinalities of A,B,C, and D (self) respectively.
D_given_ABC_DT  % The name of the DT to use for the mapping
```

Note that the cardinalities of the CPT must match the cardinalities given the structure file for the variable that uses the CPT. This CPT could then use the following decision tree.

```

D_given_ABC_DT  % DT name
3                % number of parents
0 3 0 1 default
    2 2 0 default
        -1 0
        -1 1
    -1 2
        1 2 1 default
            -1 3
            -1 4
```

The decision tree in this case does not specify the cardinalities of the random variable. A DT can therefore be used for multiple deterministic relationships amongst random variables having different cardinalities. It is the job of the deterministic CPT to state its cardinalities (just like sparse and dense CPTs), which must match that of the corresponding random variables.

You must ensure that it is not possible for the resulting leaf nodes to produce a value that is not in the range 0 through $c - 1$ where c is the cardinality of the child variable – if this occurs, a run-time error will occur. This could potentially be frustrating as such an error is not possible to detect until possibly long after the program has started. Therefore, particular care should be placed into ensuring that the DTs used can not have out of bounds leaf values.

17.3 Simple 1-D distributions: SPMFs and DPMFs

There are certain GMTK objects that implement one-dimensional CPTs that have a special functions, including the probabilities for sparse CPTs, or the mixture coefficients (responsibilities) for Gaussian mixtures. This section describes the syntax for these objects, dense probability mass functions (DPMFs) and sparse probability mass functions (SPMFs).

17.3.1 Dense probability mass functions (DPMFs)

DPMFs are simple one-dimensional CPTs, meaning CPTs with no parents, so they implement, say, $p(A)$ for some random variable A . They are declared in the way described in Section 16.2. Here is an example of a couple of DPMFs inline in a master file.

```
DPMF_IN_FILE inline 3
0      % DPMF number
mydpmf1 % DPMF name 'mydpmf1'
1 1.0  % The number of probabilities (one) and the probability

1      % DPMF number
mydpmf2 % DPMF name 'mydpmf2'
% The number of probabilities (4) and the probabilities
4 0.25 0.25 0.25 0.25

2      % DPMF number
mydpmf3 % DPMF name 'mydpmf2'
% The number of probabilities (3) and the probabilities
3 0.2 0.2 0.6
```

As can be seen, a DPMF consists of its name, followed by the number of probabilities that will follow, and then followed by that many probabilities. The number of probabilities corresponds to, for example, the number of mixture components in a Gaussian mixture in which case the probabilities indicate the component responsibilities.

Note that the number of probabilities for a DPMF might change when it is being used for Gaussian mixture responsibilities. This is because GMTK has a Gaussian component splitting/vanishing algorithm (described in Section 18.5) which can split or remove a particular component, and therefore grow or shrink the size of a DPMF. Care, therefore, should be exercised not to use a DPMF both for mixture responsibilities and sparse CPTs at the same time. If this does occur for some reason, a run-time error will be reported.

17.3.2 Sparse probability mass functions (SPMFs)

Sparse PMFs are used only with sparse CPTs, and are therefore fully described in Section 17.2.3 along with their use. To summarize, SPMFs are like DPMFs except they contain two arrays, one array giving the values of the random variable that have non-zero probability, and another array (a reference to a DPMF) of the same size that gives the non-zero probabilities for those values.

17.4 Gaussians and Gaussian Mixtures

Since Gaussians and Gaussian mixtures are so widely used, and since during training the number of components in a mixture might grow or shrink (via GMTK's splitting and vanishing algorithm), both Gaussian and Gaussian mixtures are basic objects in GMTK. These distributions are the only way at the moment to specify distributions over continuous observation vectors.

A basic Gaussian component contains a mean and a diagonal covariance vector. More advanced Gaussian components might have means that are conditional on other portions of the feature vector (either before, during, or after the current time frame). Other Gaussians might have full covariance matrices specified in this way. We begin, however, by describing simple diagonal covariance Gaussian mixtures.

17.4.1 Mean and Diagonal-Covariance Vectors

Gaussian components require a dimensionality, mean vector, and a covariance vector.

Mean vectors are quite simple. They require a name, a dimensionality, and a vector with that many mean values. Here is an example of several mean vectors.

```
MEAN_IN_FILE inline 3 % 3 mean vectors
0 % mean 1
% name, dimensionality (2), and values
mean0 2 1.0 1.0

1 % mean 2
% name, dimensionality (3), and values
mean1 3 -3.0 2.0 1.5

2 % mean 3
% name, dimensionality (5), and values
mean2 5 0.0 0.0 0.0 0.0 0.0

MEAN_IN_FILE file.means binary
```

The example first declares 3 mean vectors, all inline in a master file, and then declares some number of additional means which presumably live in a binary file named `file.means`. Of the means that are declared inline, the first one (with name `mean0`) is 2-dimensional with unity values, the second one (with name `mean1`) is three-dimensional with mixed positive and negative values, and the third one (with name `mean2`) is a five-dimensional zero-mean vector.

Mean vectors can be any dimensionality, and as can be seen the dimensionality must be specified along with the definition of the mean vector.

Diagonal covariance matrices (or actually vectors in this case) are defined using a similar syntax. Here is an example of two diagonal covariance matrices:

```
COVAR_IN_FILE inline 2 % two diag covariance matrices
0 % covar 1
covar0 10 % name and dimensionality (10)
% values
1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0

1 covar1 5
20 20 20 20 20
```

The first matrix `covar0` is a ten-dimensional unity-covariance vector, and the second is a five-dimensional covariance vector each with value 20. Note again that the dimensionality of the vector must be specified along with the covariance vector itself.

17.4.2 Gaussian Components

A Gaussian component groups together a mean and a covariance vector of the same dimension, and gives that grouping a name which is used to refer to the Gaussian component being defined. For example, here is a list of two Gaussian components (GCs).

```

MC_IN_FILE inline 2

0 % first GC
26 % dimensionality of the GC
0 % the 'type' of the Gaussian component. 0 = just mean & covar
gc_0 % the name of the component, here 'gc_0'
mean_0 covar_0 % the mean and variance of this component

1 % second GC
26 % dimensionality of the GC
0 % the 'type' of the Gaussian component. 0 = just mean & covar
gc_1 % the name of the component, here 'gc_0'
mean_1 covar_1 % the mean and variance of this component

```

These components assume the existence of the corresponding mean and covariance vectors *which must have the same dimensionality*. A Gaussian component also specifies a type (0 in the examples above) which indicate which type of component it is. The type of component specifies what additional objects are required to complete the component's definition. A component of type 0 requires only a mean and a diagonal covariance object, as given in the examples above. More on the type of component occurs in Section 17.4.4.

17.4.3 Special Internal Names: Zero and Unity Score Components

GMTK implements two special internal “Gaussians” components. They are not actually Gaussian at all but can be useful in certain circumstances for doing multi-stream and multi-rate models.

The first special component is called `internal:UnityScore`. It produces a component that will always return the value 1 (unity) regardless of the values of its arguments. I.e., $p(x) = 1, \forall x$. This therefore is not a true density, and one must be careful when using this density to ensure that the resulting model is probabilistically valid. Nevertheless, having this component can be indispensable for specifying certain models.

Another internal component density is defined as `internal:ZeroScore`. It is identical to the unity-score Gaussian except rather than always returning unity, it always returns zero.

17.4.4 Mixtures of Gaussians

Gaussian components (of any type) along with a DPMF can be bundled together to form a Gaussian mixture. Note that since different types of components can exist in a mixture, a mixture is able to consist a heterogeneous collection of components if desired.

A mixture of Gaussians is declared as in the following:

```

MX_IN_FILE inline 2

0 % mixture number
26 % dimensionality of the mixture
gm0 % the name of the Gaussian mixture.
2 % the number of components in this mixture
mx0_dpmf % the name of the DPMF containing the component
            % responsibilities. This DPMF must have
            % dimensionality 2.

```

```

gc_0 gc_1 % a list of 2 components for this mixture

1   % mixture number
26  % dimensionality of the mixture
gm0 % the name of the Gaussian mixture.
3   % the number of components in this mixture
mx1_dpmf % the name of the DPMF containing the
           % component responsibilities
           % the dimensionality of this DPMF must have value 3
gc_2 gc_3 gc_4 % a list of 3 components for this mixture

```

A Gaussian mixture definition must be given a dimensionality, and that dimensionality must match that of all of its corresponding components (and therefore which must match all of the corresponding means and covariance matrices).

17.5 Dlink matrices and structures

Dlink matrices and dlink structures are the one place where true graphical model structure is specified outside of the normal textual structural file. The structure that dlinks determine are those over observation vectors. Because it is often the case that such structure will switch for different values of hidden variables, that the structure will be automatically learned in some way, and that the amount of information contained in observation structures is quite large, it was decided that such structure should be kept out of the main structure file and specified using normal GMTK objects. Let us start with with dlink structures.

Dlink structures specify the directed dependencies that are to exist between the individual elements of vector observation vectors. Dlink structures do not specify the implementation of these dependencies, and state only that such dependencies might exist in some way.

Given a particular element of an observation vector, a dependency might exist from a parent that exists either 1) in the past of the current element, 2) in the same frame as the current element, and 3) in the future relative to the current element. This is depicted in Figure 17.1. These dependencies can co-exist with each other, so that a particular element might have multiple parents in the past, present or future.

There are a number of features of GMTKs dlink structures. First, each individual element of an observation vector can have its *own set of dependencies* which can be to any set of parents in the past, present, or future. Note, however, that an element can not be its own parent (Bayesian networks do not allow for directed cycle, and this undefined dependency would be such a cycle). Note also that a given element can not specify the same parent twice, as that would indicate a double edge, also not allowed in a Bayesian network. Second, dependency patterns can be *sparse* since there is no restriction on the set or number of parents an element can have. Third, the dependency patterns can switch as a function of a hidden parent. When a hidden variable which is a parent of an observation vector changes value, the corresponding Gaussian mixture will also (potentially) change. The Gaussian mixture can be set up so that the pattern of dependencies changes (switches) for each hidden variable value.

The dlink structure facility is quite flexible, and it allows GMTK to support Gaussians that have either full, banded diagonal, and factored sparse covariance matrices. It also allows multi-stream processing and cross-stream dependency modeling. More generally, it allows the representation of buried Markov models [44]. Let us start with a simple example first.

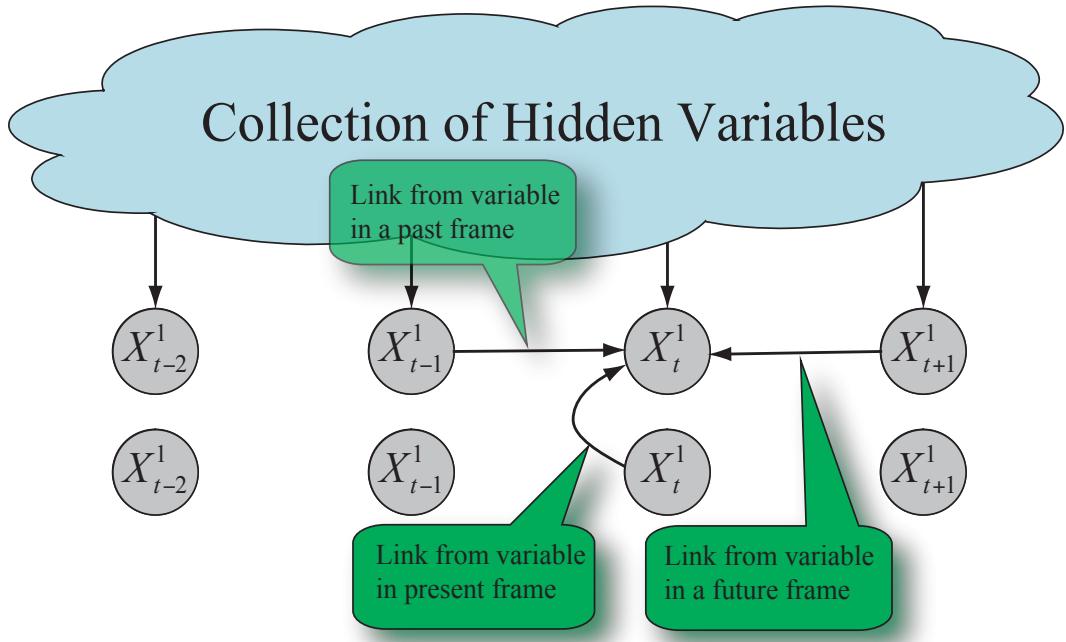


Figure 17.1: This figure shows a particular element of the current observation vector at time t can have a dependency either into the past, the present, or the future.

17.5.1 Full-covariance Gaussians

In Section TO-BE-INCLUDED (also given in [29]), it was shown how either a Gaussian can be represented by either a directed or an undirected graphical model. GMTK represents all Gaussians as directed graphical model. Repeating here part of the derivation given in Section ??, the inverse covariance matrix $K = \Sigma^{-1}$ of a Gaussian can be Cholesky factored as follows as $K = R'R$, where R is upper triangular, $D^{1/2} = \text{diag}(R)$ is the diagonal portion of R , and $R = D^{1/2}U$. A Gaussian density can therefore be represented as:

$$p(x) = (2\pi)^{-d/2} |D|^{1/2} e^{-\frac{1}{2}(x-\mu)'U'DU(x-\mu)}$$

the exponent of which can further be represented as:

$$(x - \mu)'U'DU(x - \mu) = (x - Bx - \tilde{\mu})'D(x - Bx - \tilde{\mu})$$

where $U = I - B$, I is the identity matrix, B is an upper triangular matrix with zeros along the diagonal. Therefore, a full covariance Gaussian can be specified by giving 1) a mean, 2) a diagonal covariance component, and 3) a B matrix. The B matrix contains the coefficients of the linear dependencies of a directed graphical view of a Gaussian. In particular, the i^{th} row of B specifies the regression coefficients for a Gaussian when it is factorized according to the chain rule:

$$p(x) = \prod_{i=1}^d p(x_i|x_{i+1:D}) \quad (17.1)$$

where d is the dimensionality of the Gaussian.

A dlink structure given so that it specifies parents for individual elements of a feature vector to come from the same vector (the present) can therefore be used to determine the structure of a B matrix. Moreover, since the structure can be sparse or banded, GMTK supports full, sparse, and banded diagonal matrices.

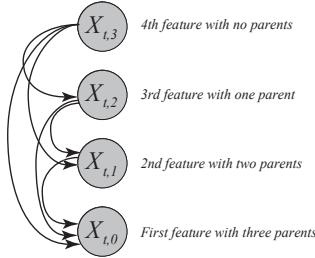


Figure 17.2: Figure showing one possible structure for the B matrix. In this case, we are defining a directed model over the individual features of a Gaussian. The first feature (feature number 0, or $X_{t,0}$) has three parents, $X_{t,1}$, $X_{t,2}$, and $X_{t,3}$. This given by the first structure specification line in the dlink structure file, which states $3\ 0\ 3\ 0\ 2\ 0\ 1$, meaning that there are 3 parents, the first parent has time lag 0 (so the current frame) and is to parent 3, the second parent has time lag 0 and is to parent 2, and the third parent has time lag 0 and is to parent 1. The specification of the parents for features 1, 2, and 3 are specified in a similar way. Note that to satisfy the chain rule of probability, care must be taken to ensure that no circularities are created in the dlink structure specification. For example, feature 3 in the figure has no parents, but if it did a circularity would exist.

Lets go over a few examples. Suppose we wish to produce a $d = 4$ dimensional full-covariance Gaussian. Furthermore, let us use the Gaussian factorization given in Equation ???. This means that element i has as parents all other elements in the observation vector at the same time that have higher element numbers, i.e., $i + 1, i + 2, \dots, d$. The dlink structure for this Gaussian is depicted in Figure 17.2. To specify the dlink structure for this Gaussian, one would specify (inline within a master file) the following:

```
DLINK_IN_FILE inline 1
0 % first dlink structure
dlink_str0 % name of the dlink structure 'dlink0'.
4      % Number of features for which this dlink structure
      % refers to. This must correspond to the dimensionality
      % of the Gaussian.
% The parents for feature element 0, which has 3 parents.
% The numbers consist of:
% <num links> <1st time lag> <1st offset> <2nd time lag> <2nd offset> ...
3 0 3 0 2 0 1
% Next, the dlinks for feature 1
2 0 3 0 2
% Next, the dlinks for feature 2
1 0 3
% Finally, the dlinks for feature 3, and there are no dependencies.
0
```

The dlink structure contains the following. First, the dlink structure name `dlink0`. Next comes an integer indicating the number of features that this dlink can be used with. When a Gaussian component uses this dlink structure, the dimensionality of the Gaussian must correspond to the number of features in the dlink structure. Therefore, we could call this number d in general. Next comes the specification of the structure itself. This consists of d lists of parents, one for each feature starting at the lowest feature of the dlink structure (e.g., feature 0) and ending at the highest feature number (e.g., feature $d - 1$).

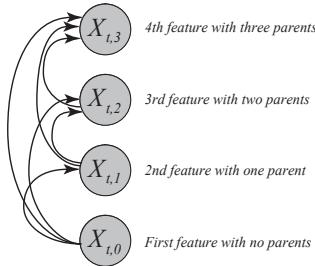


Figure 17.3: Figure showing another possible structure for the B matrix.

In the example above, feature element 0 has three parents, all from relative time lag 0 (meaning all parents come from the same frame). The three parents consists of feature elements 3, 2, and 1 respectively (see Figure 17.2). Feature element 1 has two parents in the same frame, features 3 and 2. Feature 2 has one parent, feature 3. Finally, the fourth feature, feature 3, has no parents.

The dlink structure is flexible, so different chain rule factorizations can be specified just as easily. For example, given the chain rule factorization of the Gaussian as follows:

$$p(x) = \prod_{i=1}^d p(x_i | x_{1:i-1}) \quad (17.2)$$

This can be specified as follows.

```
DLINK_IN_FILE inline 1
0 % first dlink structure
dlink_str1
4      % Number of features.
0
1 0 0
2 0 0 0 1
3 0 0 0 1 0 2
```

This says that feature 0 has no parents, feature 1 has one parent (at element 0), feature 2 has two parents (elements 0 and 1) and lastly feature 3 has three parents (elements 0, 1, and 2). This is depicted in Figure 17.3. Note that there are many other ways of factorizing a Gaussian and a dlink structure can represent all of them (there are $d!$ possible factorizations). Regardless of the order of factorization, a full-covariance Gaussian can be specified.

Once a dlink structure for a particular ordering of the factorization of a B matrix is specified, it is necessary to also specify the actual regression coefficients of the B matrix for the Gaussian, and this is done using a dlink matrix. A dlink matrix has a coefficient for each dependency specified in a dlink structure.

```
DLINK_MAT_IN_FILE inline 1
0 % first dlink matrix
dlink_mat0 % name of the dlink matrix.
dlink_str0 % Name of the dlink structure to be used with
            % this dlink matrix.
4          % number of features, must match dlink structure
% the regression coefficients for the dlink structure.
% For each vector element, an integer giving
```

```
% the number of coefficients and the coefficients themselves
% must be specified.
3 0.1 0.2 0.3
2 0.4 0.5
1 0.6
0
```

The dlink matrix first consists of its name `dlink_mat0`. Next, the name `dlink_str0` of the dlink structure to be used with this matrix is given. After that, the number of features is given. This number must match that of the number of features of the dlink structure. Finally, the list of coefficients is given, one for each feature element, meaning one each for the corresponding element given in the dlink structure.

We are at last ready to specify a full-covariance Gaussian. We will create a full-covariance Gaussian using the Cholesky factorization describe above, but we will use upper-triangular B matrices that are dense in the upper-triangular portion. This produces a full-covariance Gaussian as described in Section TO-BE-INCLUDED (also see [29]).

Lets use the factorization given in Equation 17.1 and specify the dlink structure, dlink matrix, a mean vector, a covariance vector, and a Gaussian component for a $d = 12$ dimensional full-covariance Gaussian component. What follows is the complete annotated specification.

```
% specify a 12-D zero mean vector
MEAN_IN_FILE inline 1 % 1 mean vector
0 % mean 1
mean0 12 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
COVAR_IN_FILE inline 1

% specify a 12-D unity diagonal covariance matrix
COVAR_IN_FILE inline 1 % 1 diagonal covariance matrix
0 % covar 1
covar0 12 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0

% specify a dlink structure corresponding to element
% i having parents all elements with index greater than i.
DLINK_IN_FILE inline 1
0 % dlink structurr 1
dlink0 % name of dlink structure = 'dlink0'
12 % dimensionality
11 0 11 0 10 0 9 0 8 0 7 0 6 0 5 0 4 0 3 0 2 0 1
10 0 11 0 10 0 9 0 8 0 7 0 6 0 5 0 4 0 3 0 2
9 0 11 0 10 0 9 0 8 0 7 0 6 0 5 0 4 0 3
8 0 11 0 10 0 9 0 8 0 7 0 6 0 5 0 4
7 0 11 0 10 0 9 0 8 0 7 0 6 0 5
6 0 11 0 10 0 9 0 8 0 7 0 6
5 0 11 0 10 0 9 0 8 0 7
4 0 11 0 10 0 9 0 8
3 0 11 0 10 0 9
2 0 11 0 10
1 0 11
0
```

```
% The dlink matrix corresponding to dlink structure dlink0
DLINK_MAT_IN_FILE inline 1
0      % dlink matrix 1
dlink_mat0 % name of dlink matrix
dlink0    % name of dlink structure to use
12       % dimensionality
% regression coefficients, all zeros for now.
11 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
10 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
9  0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
8  0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
7  0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
6  0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
5  0.0 0.0 0.0 0.0 0.0 0.0
4  0.0 0.0 0.0 0.0
3  0.0 0.0 0.0
2  0.0 0.0
1  0.0
0

% The gaussian component using the above.
MC_IN_FILE inline 1
0  % first GC
12 % dimensionality of the GC
1  % the 'type' of the Gaussian component. 1 = use dlink structure
gc_0 % the name of the component, here 'gc_0'
mean0 covar0 dlink_mat0 % the mean, variance and dlink matrix to use.
```

The above defines a component named `gc_0`. Note that the type of the component in this case is 1 (rather than 0 as was done in Section 17.4.2). Type 1 indicates that a mean, covariance, *and* dlink matrix should follow. Note that factorizations other than Equation 17.1 can be produced just as easily, by specifying a different dlink structure and matrix. Moreover, it is possible to produce a mixture of Gaussians that have components with using different factorizations. This feature will become more interesting in the next section.

17.5.2 Banded Diagonal and/or Sparse Factored Inverse Covariance Matrices

Dlink matrices and structures can do more than specify full-covariance matrices. They may also be used to specify a variety of different directed structures for the Gaussian densities. This can lead to a wide variety of possible sparse directed Gaussian structures.

The way this is done is similar to the way that full covariance matrix Gaussians are formed, but rather than giving the full set of possible parents for a given Gaussian factorization, instead only a subset is given. For example, as stated above, any density function (including a Gaussian) can be factorized according to the chain rule of probability. Suppose the dimensionality of the Gaussian is d , and that $\pi(i), i = 1 \dots d$ is an

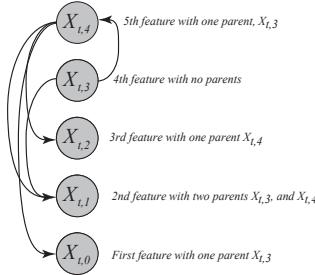


Figure 17.4: A directed model for a sparse 5-dimensional Gaussian.

arbitrary permutation of the integers from 1 to d inclusive. The chain rule states the following:

$$p(x) = \prod_{i=1}^d p(x_{\pi(i)} | x_{\pi(1:i-1)}) \quad (17.3)$$

where $\pi(1:j)$ represents the first j integers in the permutation, i.e., $\pi(1:j) = \{\pi(1), \pi(2), \dots, \pi(j)\}$. By making conditional independence assumptions about the distribution, we may change the above factorization as follows.

$$p(x) = \prod_{i=1}^d p(x_{\pi(i)} | x_{\text{pa}(i)}) \quad (17.4)$$

where $\text{pa}(i)$ denote the parents of element i , and since it is a directed model, we have that the parents are a subset, meaning that $\text{pa}(i) \subseteq \pi(1:i-1)$. This factorization corresponds to the set of independence statements for each i

$$X_{\pi(i)} \perp\!\!\!\perp (X_{\pi(1:i-1)} \setminus X_{\text{pa}(i)}) | X_{\text{pa}(i)}.$$

Note again that the permutation π can be arbitrary. GMTK supports all possible permutations in its representation of sparse Gaussians. Moreover, in light Equation ?? in Section ??, the sparseness corresponds to zeros in the B matrix. Therefore, such a sparse pattern can be set up using the dlink structure and matrix mechanism mentioned above. Lets run through a few examples.

Suppose that we have a 5-dimensional Gaussian distribution, and we wish to represent the sparse directed graph given in Figure 17.4. We need only to create a dlink structure and a corresponding matrix for this purpose, as follows:

```

DLINK_IN_FILE inline 1
0 % first dlink structure
sparse_dlink0 % name
5 % Number of features.
% The parents for feature element 0, 1 parent, number 4
% The numbers consist of:
% <num links> <1st time lag> <1st offset> <2nd time lag> <2nd offset> ...
1 0 4
% Next, the dlinks for feature 1, two parents, 3 and 4.
2 0 3 0 4
% Next, the dlinks for feature 2, one parent, feature 4
1 0 4
% Next, dlinks for feature 3, no dependencies.
0

```

```
% Finally, dlinks for feature 4, one parent, feature 3
1 0 3

% Next, we have a dlink matrix for the above structure.
DLINK_MAT_IN_FILE inline 1
0      % dlink matrix 1
sparse_dlink0 % name of dlink matrix
sparse_dlinkmat0    % name of dlink structure to use
5        % dimensionality
% regression coefficients, all zeros for now.
1 0.4
2 -0.4 1.3
1 1.2 1.0
0
1 -2.4
```

Note that in the above structure, there are dependencies not only from lower numbered elements to higher numbered ones (namely, element 0 and 2 both have element 4 as a parent, element 2 has elements 3 and 4 as parents) but an element can have a lower numbered parent as well (namely, element 4 has element 3 as a parent). Note that this is OK since the structure does not specify a directed cycle, meaning the structure is a directed acyclic graph (DAG). To see this, place $X_{t,3}$ at the top of the DAG, where it has children $X_{t,4}$ and $X_{t,1}$. $X_{t,4}$ has three children $X_{t,2}$, $X_{t,1}$, and $X_{t,0}$.

In general, any combination of child and set of parents is a valid configuration in GMTK, except that a child may not be a parent of itself (note that specifying self as parent will cause GMTK to signal an error). Also, note that it is important to produce a structure that corresponds to a valid permutation and therefore factorization as specified in Equation 17.3. Note that using the representation for dlinks given above it is not only possible to specify all permutations, but it is also possible to specify graphs that have directed cycles, as in the following dlink structure over a 3-dimensional Gaussian.

```
DLINK_IN_FILE inline 1 0 circularity_dlink 3
1 0 1
1 0 2
1 0 0
```

The above structure states that element 1 is a parent of element 0, element 2 is a parent of element 1, and element 0 is a parent of element 2. Such a model *does not* correspond to a valid chain rule factorization for any permutation, and is therefore not a DAG, but is nevertheless allowed in the current version of GMTK. Care must be taken, therefore, to ensure that your sparse structures do not contain such a circularity.²

17.5.3 Sparse Global B Matrix

It is possible with GMTK to set up a structure to learn a sparse global B matrix. Having and learning the parameters for such a B matrix is similar to learning a global scaling and rotation matrix that is applied to all feature vectors, similar to a matrix transform (such as a discrete cosine transform) which is applied to the feature vectors before they are used in the ASR system. In this case, however, the transform can be learned in a maximum likelihood setting.

²The authors would be interested to hear if anyone obtains useful results using such circularities, which are clearly invalid probabilistic structures. My guess is that this will result in numerical oddities, such as zero valued variances.

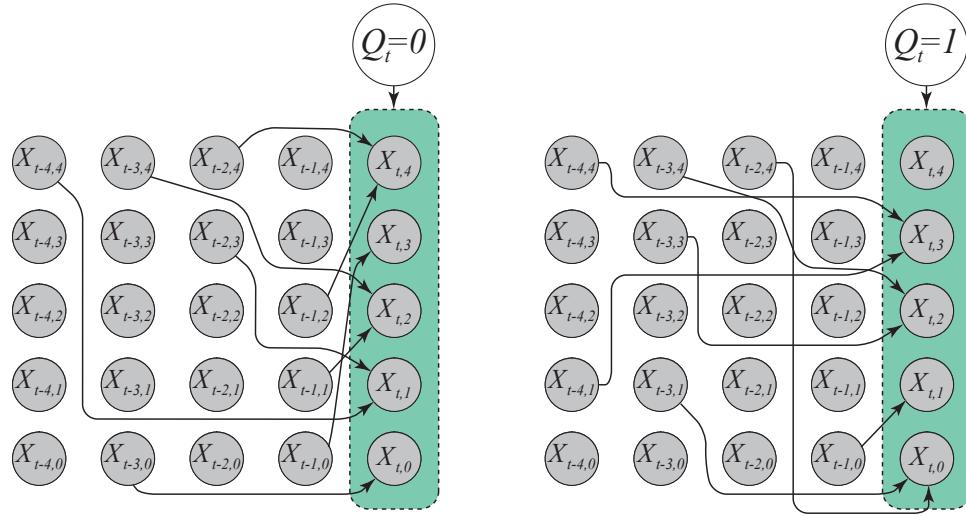


Figure 17.5: A simple BMM over 5-dimensional feature vectors. Note that the structure changes (switches) depending on the value of the hidden variable Q_t . Specifically, when the hidden variable $Q_t = 0$ the structure on the left (and its parameters) are active. when $Q_t = 1$ the structure on the right is active. The dlink structures and matrices for this graph is specified in the text.

17.5.4 Buried Markov Models with Linear Dependencies

Using the above structures, it is just as easy to specify structures corresponding to buried Markov models ([44]), but where the edge implementations between observation elements are linear.

In all of the above dlink structures, you might have noticed the zero that comes in between element indices. For example, in a line `3 0 3 0 2 0 1` defining a dlink structure, the zeros specify that the parents come from the the current frame. Specifically, these zeros are the current relative frame offset from the current frame at which to obtain the parent elements. If all of the offsets are always zero, then this will correspond to a certain permutation of a Gaussian factorization³. If, on the other hand, the offset is negative then the parent will come from the past relative to the current frame, and will implement a conditional Gaussian distribution, where the mean of the Gaussian is a function of the parents. If the offset is positive, the parents will come from the future relative to the current frame. There may be any mix between negative, zero, and positive temporal offsets.

Using dlinks structures and matrices, it is possible to derive BMMs where the structure switches as a function of one of the discrete parents. The parent simply needs to specify a Gaussian mixture that uses a different set of dependency links.

Lets consider the following BMM structure specified in Figure 17.5. This BMM uses 5-dimensional feature vectors, and has the sparse pattern as shown. Also shown is the fact that the BMM changes its structure depending on the value of the hidden variable at the time (lets assume that the hidden variable is binary for now). When the hidden variable is zero ($Q_t = 0$), we have the first sparse pattern on the left, and when it is one ($Q_t = 1$), we have the second pattern on the right.

These dlink structure for this case can be specified as follows. This requires several objects. First, two dlink structures are defined, one for each hidden variable value. Next, two dlink matrices are defined, one for each dlink structure. In general, that multiple dlink matrices can share the same dlink structure, which would correspond not to a switching structure, but rather switching parameter values. For illustrative purposes, in this example each dlink matrix uses a distinct dlink structure for each dlink matrix (and corresponding

³Again, assuming it is a valid factorization, see the previous section on sparse covariances

hidden variable value).

```

DLINK_IN_FILE inline 2
0 % first dlink structure
bmm_struct_0 % name [REDACTED]
5 % Number of features.
% note that the offsets are no longer zero.
1 -3 0
2 -2 3 -4 4
2 -3 4 -1 1
1 -1 0
2 -1 2 -2 4

1 % next dlink structure
bmm_struct_1 % name [REDACTED]
5 % Number of features.
2 -3 1 -2 4
1 -1 0
2 -3 3 -3 4
2 -4 1 -4 4
0

% Next, we have a dlink matrcies for the above structures.
DLINK_MAT_IN_FILE inline 2
0 % dlink matrix 1
bmm_mat_0 % name [REDACTED]
bmm_struct_0 % name of dlink structure to use
5 % dimensionality
% regression coefficients, all zeros for now.
1 0.3
2 0.4 -0.4
2 0.1 0.0
1 1.0
2 1.3 -1.4

1 % dlink matrix 2
bmm_mat_1 % name [REDACTED]
bmm_struct_1 % name of dlink structure to use
5 % dimensionality
% regression coefficients, all zeros for now.
2 -2.0 4.0
1 1.0
2 1.0 2.0
2 -3.0
0

```

Now that the dlink structures and matrices are set up, it is possible to specify two Gaussian components which utilize these two structures (we assume that the mean and variance objects have been defined

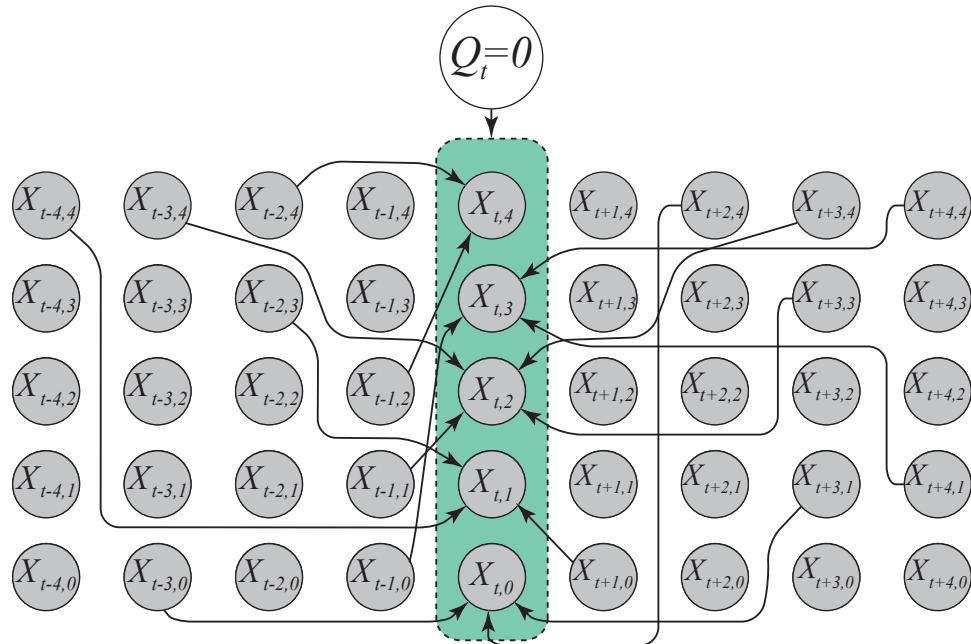


Figure 17.6: A BMM with parents both in the past and future relative to the the current frame.

somewhere, but are not specified in what follows).

```
MC_IN_FILE inline 2

0  % first GC
5  % dimensionality of the GC
1  % the 'type' of the Gaussian component. 1 = use dlink
gc_0 % the name of the component, here 'gc_0'
mean_0 covar_0 bmm_mat_0 % the mean, variance, and bmm mat of this component

1  % second GC
5  % dimensionality of the GC
1  % the 'type'
gc_1 % name of this component, here 'gc_1'
mean_1 covar_1 bmm_mat_0 % the mean and variance of this component
```

It is also interesting to note that parents need not come only from the past, but may also come from the future (or any combination of past, present, and future) as the following simple dlink structure demonstrates (as also given in Figure 17.6).

```
DLINK_IN_FILE inline 1
0 % first dlink structure
bmm_struct_0 % name
5 % Number of features.
% note that the offsets are no longer zero.
3 -3 0 3 1 2 4
3 -2 3 -4 4 1 0
```

```

4 -3 4 -1 1 3 3 3 4
3 -1 0 4 1 4 4
2 -1 2 -2 4

```

Note that for readability, it is of course always possible to reformat the above dlink structure specification. It is only white space that needs to separate the characters. For example:

```

DLINK_IN_FILE inline 1
0 % first dlink structure
bmm_struct_0 % name
5      % Number of features.
% note that the offsets are no longer zero.
3 -3 0
  3 1
  2 4
3 -2 3
 -4 4
  1 0
4 -3 4
 -1 1
  3 3
  3 4
3 -1 0
  4 1
  4 4
2 -1 2
 -2 4

```

Again, however, it is important to realize that when using structure with dependencies in the past, present, and future, the resulting graph might contain directed cycles. Care should therefore be taken to ensure that structures are not specified as such, or otherwise unexpected results may follow. Once again, GMTK does *not* signal this *a priori* as an error, and it is only after training might the results of such invalid structures be discovered. As mentioned above, the most likely result, however, will be numerical instabilities (e.g., IEEE floating point infinities, and/or messages about dividing by zero).

Exercise 159. Is it possible to define a linear conditional Gaussian $p(x|y)$ (with a DLINK under the global observation matrix) where "x" is a N -dimensional vector and "y" is a M -dimensional vector with N not equal to M . Also, is it possible to define $p(x|y, z)$? Assume that x , y and z are observed.

17.6 GMTK Parameter Sharing/Tying

GMTK supports a flexible array of options for parameter sharing and parameter tying. Any mean can be tied to any other mean (of the same dimension), any diagonal covariance may be tied to any other diagonal covariance (again of the same dimension), and any dlink matrix may be tied to any other dlink matrix (as long as the structures of the two dlink matrices are the same).

Sharing need not only be specified during testing, but may also be specified before training begins. GMTK uses a GEM algorithm for training in this, an algorithm that retains the convergence Guarantees that

normal EM possess, as long as training is done in the right way. The GEM algorithm is fully described in [49].

Parameter sharing is specified very simply in GMTK. When an object in GMTK is defined that consists of several constituent sub-objects, each of those sub-objects is specified using the sub-object name. If two objects wish to share the same sub-object, they need only specify the same name. For example, if two Gaussian components (which each normally consist only of a mean and a diagonal covariance vector) wish to share the same mean, they need only specify the same mean name. For example,

```
MC_IN_FILE inline 2
0 % first GC
26 % dimensionality of the GC
0 % the 'type' of the Gaussian component. 0 = standard
gc_0 % the name of the component, here 'gc_0'
mean_0 covar_0 % the mean and variance of this component

1 % second GC
26 % dimensionality of the GC
0 % the 'type' of the Gaussian component. 0 = standard
gc_1 % the name of the component, here 'gc_0'
mean_0 covar_1 % the mean and variance of this component
```

In the example above, `gc_0` and `gc_1` have unique diagonal covariance matrices `covar_0` and `covar_1` respectively), but they share the same mean vector `mean_0`. When the Gaussian is evaluated, that same mean will be used. Moreover, even Gaussians of different types can share a mean vector. This implies that a Gaussian of type '0' (having just a mean and diagonal covariance) and a Gaussian of type '1' (having a mean vector, diagonal covariance, and B matrix) can share the same mean (or diagonal covariance as well).

Note that during training, GMTK figures out when an object is shared, and decides based on the particular type of sharing if an EM algorithm can be used, or if for that particular object, a GEM algorithm must be run. This means that EM and GEM might be used simultaneously to train the set of parameters in GMTK, each will be used to match the appropriate parameter sharing. GMTK itself figures out when it should run EM and when it should run GEM based on the sharing pattern that is specified by the user.

The example above demonstrates only how mean vectors can be shared. GMTK however can share or tie many basic object in the same way. In particular, means, covariances, and dlink matrices can be shared in the same way simply by specifying the name of the sub-object so desired. When two or more different objects specify the same sub-object, then that sub-object is shared. Objects that can be shared this way include decision trees, named collections, dense probability mass functions (DPMFs), sparse probability mass functions (SPMFs), dlink structures, and Gaussian components (i.e., different Gaussian mixtures can share the same Gaussian component). Lets give a few examples.

First, here is an example where a decision tree, named `myDT`, is shared between two different deterministic CPTs. This can be done as follows:

```
DETERMINISTIC_CPT_IN_FILE inline 2
0 % first DETERMINISTIC_CPT
detcpt1 % name of the CPT
1 % num parents of the corresponding random variable
2 3 % cardinalities. cardinality of parent = 2, of self = 3
myDT % The name of an existing DT to implement the deterministic
```

```
% mapping

1      % second DETERMINISTIC_CPT
detcpt2 % name of the CPT
1      % num parents of the corresponding random variable
10 20  % cardinalities. cardinality of parent = 2, of self = 3
myDT  % The name of an existing DT to implement the deterministic
      % mapping
```

Note that the only difference between the two CPTs is that they have different cardinalities. The cardinalities therefore are associated with a CPT, not with a decision tree. The same decision tree can be used for any purpose where it might be useful. In the example above, the two deterministic CPTs implement the same function, but the first and the second requires a parent cardinality of 10 (and self cardinality of 20).

As our next example, suppose it is desired to share a dlink structure across multiple dlink matrices (this, for example, can be used to set of a system with a globally shared B matrix, as described above). This can be set up as follows.

```
% specify a dlink structure corresponding to element
% i having parents all elements with index greater than i.
DLINK_IN_FILE inline 1
0      % dlink structure 1
dlink0 % name
3      % dimensionality
2 0 2 0 1
1 0 2
0

% The dlink matrix corresponding to dlink structure dlink0
DLINK_MAT_IN_FILE inline 2
0      % dlink matrix 1
dlink_mat0 % name of dlink matrix
dlink0    % name of shared dlink structure to use
3      % dimensionality
% regression coefficients, all zeros for now.
2 0.0 0.0
1 0.0
0

1      % dlink matrix 1
dlink_mat1 % name of dlink matrix
dlink0    % name of shared dlink structure.
3      % dimensionality
% regression coefficients, all zeros for now.
2 0.0 0.0
1 0.0
0
```

In this example, the two dlink matrices `dlink_mat0` and `dlink_mat1` share the same dlink structure `dlink0`, simply by specifying the name `dlink0` in both instances.

Mixtures of Gaussians and the various forms of CPT objects (dense, sparse, and deterministic) can also be shared, but are done so in a different way, namely via the structure file and/or the current name collection object being used.

To share a CPT object of some sort, the name of the CPT is specified whenever it is needed. The following is an example:

```
variable : var1 {
    type: discrete hidden cardinality 2 ;
    conditionalparents :
        foo(0) using DenseCPT("sharedCPT");
}
variable : var2 {
    type: discrete hidden cardinality 2 ;
    conditionalparents :
        bar(0) using DenseCPT("sharedCPT");
}
```

In this case, the dense CPT named `sharedCPT` is specified twice by both random variables `var1` and `var2`. Note that the cardinalities of the variables that share a CPT and the cardinalities of the corresponding parents must be equivalent to each other, and must be the same as the corresponding CPT. For example, assuming that `foo` has cardinality 3, then `bar` must also have cardinality 3, and the shared CPT named `sharedCPT` must be a 3×2 table.

Mixture of Gaussian objects can also be shared between multiple sets of continuous observed random variables. Again, this is done via a combination of the structure file and the collection objects. Consider the following example, first a few definitions of collection and decision tree objects which is then followed by a structure file segment.

```
NAME_COLLECTION_IN_FILE inline 3
0 % first
col1 % The name of the collection, 'col1'
3 % The collection length, 3 entries.
gm1
gm2
gm3
1 % second
col2 % The name of the collection, 'col2'
3 % The collection length, 3 entries.
gm4
gm3
gm2
2 % third
col3 % The name of the collection, 'col3'
3 % The collection length, 3 entries.
gm3
gm5
gm4
```

```
DT_IN_FILE inline 3
0 % first
map1    % name of decision tree, 'map1'
1       % only one parent.
-1 (p0) % just copy value of parent
1 % second
map2    % name of decision tree, 'map2'
1       % only one parent.
-1 (p0) % just copy value of parent
2 % third
map3    % name of decision tree, 'map3'
1       % only one parent.
-1 (p0) % just copy value of parent
```

```
variable : obs1 {
    type: continuous observed 0:12 ;
    conditionalparents: foo(0) using mixture
        collection("col1")
        mapping("map1");
}
variable : obs2 {
    type: continuous observed 0:12 ;
    conditionalparents: bar(0) using mixture
        collection("col2")
        mapping("map2");
}
variable : obs3 {
    type: continuous observed 13:25 ;
    conditionalparents: baz(0) using mixture
        collection("col3")
        mapping("map3");
}
```

In this case there are three observed variables `obs1`, `obs2`, and `obs3`, each using its corresponding collection objects and mapping objects.

The first thing to note is that for continuous observation variables to share Gaussians, the (necessarily vector) variables must have the same dimensionality (vector length). In the case above, each of the variables have dimensionality 13, feature elements 0 through 12 for `obs1` and `obs2`, and feature elements 13 through 25 for `obs3`.

Each of the observation vectors use both a mapping decision tree and a collection object. The decision trees `map1`, `map2`, and `map3` map from the parent random variables, `foo`, `bar`, and `baz` respectively to positions within the collection objects (`col1`, `col2`, `col3` respectively).

Note that any of the variables should share either the collections or the maps, and in the example above, the definition of the mapping decision trees are in fact the same. For simplicity, the example shows each variable with its own collection and mapping objects.

As defined above, each of the collection objects have length three. That means that the decision tree

objects should never have a leaf node which has value greater than two (the values can only be between 0 and 2). As defined above, however, the decision trees copy the parent random variable value, which means that the corresponding random variables `foo`, `bar`, and `baz` should never have a value that is greater than 2. Note that in GMTK, there can be a value greater than 2 **only** if all such values have zero probability.

The three collection objects define lists of names of Gaussian mixtures, `gm1` through `gm5`. From the file above, it means, for example, that `col1`'s first (i.e., zero position) mixture is `gm1`, `col2`'s first mixture is `gm4`, and `col3`'s first mixture is `gm5`. The example above, for example, means that when `obs1`'s parent `foo` has value 2, it uses Gaussian mixture `gm3`. This same mixture is also used for `obs2` when its parent `bar` has value 1, and used for `obs3` when its parent `baz` has value 0. This, therefore, is the general way that Gaussian mixtures can be shared.

Note that there are actually two ways that mixtures could be shared, either via the collection object, or via the decision tree. In the above example, all of the sharing was done via the collection objects since the decision trees were identical and simply mapped the parent value to the leaf value. Other decision trees could be used, however, which maps collections of values of parent variables to the same position in the collection object.

Lastly, note that variable `obs3` shares Gaussians with variables `obs1` and `obs2`, even though `obs3` corresponds to a different feature range. This is allowed in GMTK, meaning that it is possible to share the same Gaussian distribution over a different and disjoint set of features. There can even be overlap between the two ranges, if the ranges as given overlap. It is moreover possible to share portions of a Gaussian, for example, by having the Gaussian components share the same means (as specified in Section xxx). In sum, these abilities give GMTK a great deal of flexibility in how it is able to share Gaussian densities or portions of Gaussian densities.

17.6.1 GEM training and parameter Sharing/Tying

GMTK supports both EM training and GEM training. The underlying equations for GEM training is described in Section TO-BE-WRITTEN. It should be noted here, however, that the GMTK general training program determines when either EM or GEM training is appropriate, depending on the degree and type of sharing that occurs. In fact, it might be that GEM and EM training are running internally at the same time, for different parameter values.

17.7 Virtual Evidence (VE) and time-inhomogeneous VECPTs

One way to impart virtual evidence to a hidden random variable is to specify an additional random variable which is observed to be unity, and then to add a factor relating the two random variables. If this is done over time, then the factor is time-homogeneous, meaning that the virtual evidence values do not change from one time frame to another.

Also mention here about hybrid systems, and give a small example on how to set up a hybrid system.

17.8 Deep Models, Virtual Evidence (VE) and DeepVECPTs

17.9 The Global Observation Matrix

GMTK uses a global observation matrix, which is read in for all files. The global observation matrix contains, say, T frames (for a T frame utterance) and some number M of features per frame. It can contain both continuous M_c and discrete M_d features, where $M = M_c + M_d$. The lower portion of the matrix contains only continuous features and the upper portion contains only discrete features. The matrix might

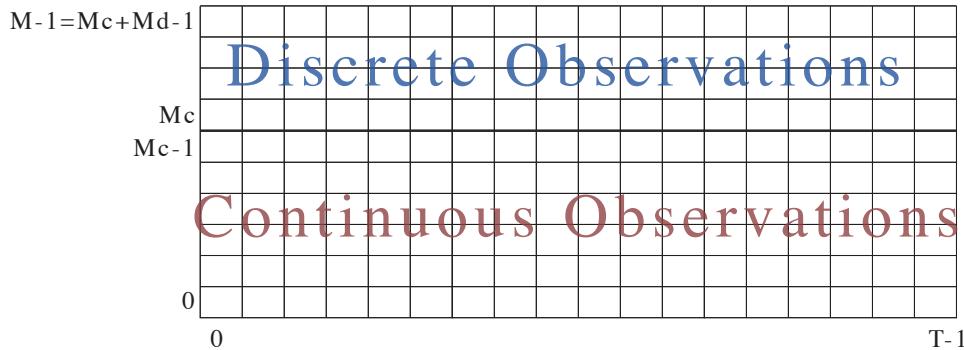


Figure 17.7: The Global Observation Matrix

have only continuous ($M_d = 0$) or only discrete features ($M_c = 0$). Figure 17.7 shows an example of an $M \times T$ sized matrix, and how it is organized.

Feature range specifications for observed random variables in a structure file determines the range in the global observation to obtain the observation values for the current frame. For example, once the template network is unrolled so that it is T frames long, each observed random variable at, say, frame t obtains its values from the elements within column t of the matrix. It is important therefore to specify feature ranges that map to the appropriate type of feature (i.e., a Gaussian should not specify a range that includes discrete features, and a discrete variable should not specify a continuous feature). If this occurs, GMTK will produce an error message.

17.9.1 Dlink Structures and the Global Observation Matrix

Another situation in which portions of columns of the global observation matrix is specified is when a dlink structure (see Section 17.5) is used. Consider the following example:

```
DLINK_IN_FILE inline 1
0
dlink_str0
4      % Number of features for which this dlink structure
3 0 3 0 2 0 1
% Next, the dlinks for feature 1
2 0 3 0 2
% Next, the dlinks for feature 2
1 0 3
% Finally, the dlinks for feature 3, and there are no dependencies.
0
```

For the specification of *feature* position elements in a dlink structure, the dlink structure specifies **absolute** positions within the global observation matrix rather than positions relative to the Gaussian that is using the Gaussian mixture. In other words, the dlink structure specifies the same dependencies regardless of what Gaussian uses it. For example, in the example above, the first feature of the dlink structure requires absolute feature position 3, 2, and 1 in the current frame of the observation matrix. If a Gaussian using this dlink structure was defined over features 0 through 3 (i.e., the structure file specifies the range 0 through 3), then this would correspond to a full covariance Gaussian, say $p(x)$ where x is the 4-dimensional vector corresponding to features 0 through 3 . If, on the other hand, a Gaussian using this dlink structure was

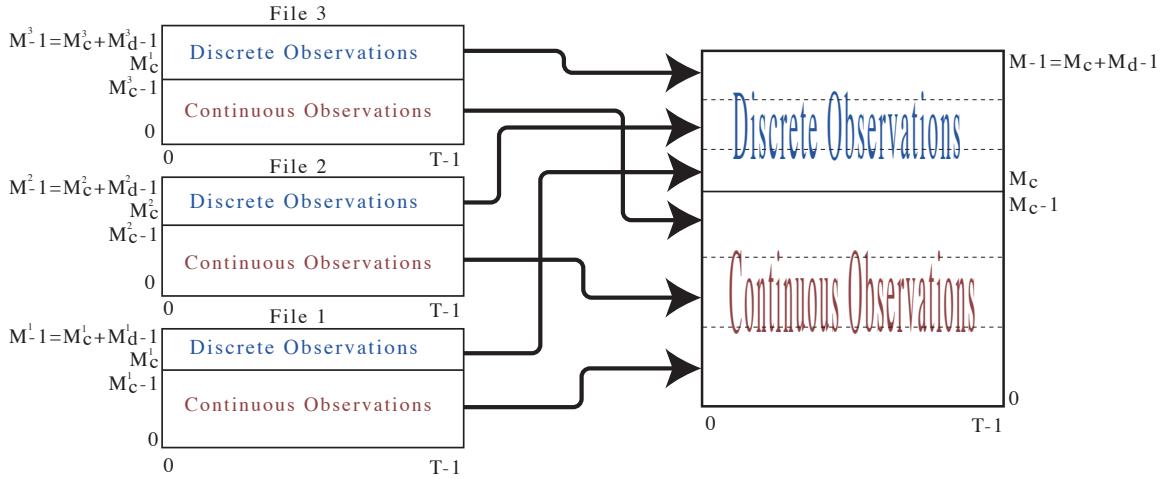


Figure 17.8: The combination of three feature files on the left consisting each of both continuous and discrete observations. On the right is the virtual global observation matrix, combined from the three feature files so that the continuous observations come first followed by the discrete observations.

defined over features 4 through 7, this would correspond to a linear conditional Gaussian, say $p(y|x)$ where y is a 4-dimensional vector over features 4 through 7 of the current frame, and x is again features 0 through 3 of the current frame.

Given the above, when specifying non-diagonal covariance Gaussians, care should be taken to ensure that the dlink structure used corresponds to the desired conditioning set. On the other hand, with this ability, it is easy to create a variety of conditional multi-stream Gaussian models, say $p(y|x)$ where y comes from one stream, and x comes from another stream. This is particularly easy using multiple observation files described below.

Note that for temporal elements, the dlink structure specifies **relative** rather than absolute positions within the observation matrix. For example, a dlink structure line of the form

```
3 0 3 0 2 0 1
```

would select feature elements (i.e., parents) 3, 2, and 1 in the current frame (the three zeros), but in

```
3 -1 3 -2 2 -3 1
```

this selects parent 3 in the previous frame (-1), parent 2 two frames in the past, and parent 1 three frames in the past.

17.9.2 Multiple Files and the Global Observation Matrix

In GMTK, the notion of a frame (or slice) will be the same as is typical in automatic speech recognition (ASR) systems.⁴ One should realize that GMTK's templates use sections, consisting of a prologue, chunk, and epilogue each of which can be any number of frames long.

The global observation matrix is formed from the vertical concatenation one or more observation files. There is a specific command line syntax to use, and is described in the section on the programs (see below).

⁴In ASR, this is often a 25ms sliding time window, with a 10ms time-step between windows. This is only a convention, however, any frame width and step can be used.

Each observation file used must contain the same number (say T) of frames, but the files may have as many features per frame (say M^i) as is desired. Moreover, each observation file may have any number of continuous M_c and/or discrete M_d features, where $M = M_c + M_d$. Supposing that some number of files are concatenated together, where the i 'th file has M^i features, the resulting virtual internal global observation matrix will have $\sum_i M^i$ features. The continuous and discrete features are reorganized, however (see Figure ??). In particular, if M_c^i (resp. M_d^i) is the number of continuous (resp. M_d^i) features in the i 'th file, then once the files are concatenated together, the first $M_c = \sum_i M_c^i$ features (i.e., features 0 through $M_c - 1$) are continuous, and the remaining $M_d = \sum_i M_d^i$ features are discrete (i.e., features M_c through $M_c + M_d - 1$). This way, it is always the case that the lower indexed features are continuous and the higher indexed features are discrete.

Note that selection of the feature stream files, and the orders that the feature streams are combined, can be changed on the fly using GMTK's command line arguments (see below). GMTK structure files and dlink structures, however, specify absolute locations in the virtual observation matrix. Therefore, if an experiment exists and a new file to merge is added at the beginning, the range of the feature values will change. For example, suppose there are two observation files at the moment, the first (file A) has 5 features and the second (file B) has 3 features. The global observation matrix will have 8 features, and the first five (features 0-4) will correspond to file A, and the last 3 (features 5-7) will correspond to file B. If another file (file C) is used as the first file (and suppose file C has two features) so that the file order used is now C A B, that will effectively shift up by 2 the indices of the features corresponding to file A and file B. This could significantly change the behavior of a program, especially if one is using Gaussians that have already been trained using just the file A and file B features.

In general, it is better to have any new feature files added at the end rather than the beginning (e.g., order A B C above) so that the ranges in the structures and dlinks that have already been specified will correspond to the same features and therefore still be valid.

17.9.3 Multiple Files and the Global Observation Matrix

17.10 Virtual Evidence (VE) and VECPTs

Also mention here about hybrid systems, and give a small example on how to set up a hybrid system. Refer to the section on virtual evidence, §5.5, and Hybrid systems and virtual evidence, §5.7.2.

Chapter 18

The Main GMTK Programs

GMTK consists of a number of different programs that allow you to do anything from train the parameters of a graphical model based system, to convert parameters from ASCII to binary format.

Instructions for submitting bugs, mention the track web page.

Instructions for being added to gmtk-users.

18.1 Integer range specifications

Integer ranges are used for a number of purposes in GMTK programs: decision tree range specifications, ranges of features to include in the global observation matrix, the set of utterances to include in training. Integer ranges are really just a way of specifying a general set of integers. GMTK uses a particular syntax for specifying these sets as follows:

In general, an integer range specifies a set of integers in the range 0 through $N - 1$, where N is the total number of possible integers. In some cases, N is known and there are shortcut range specifications that can be used (e.g., the number of utterances, the number of accumulator files, etc.). In other cases, N is unknown at the time the range is given (e.g., a decision tree range, since a given decision tree can be used with different random variables having different cardinalities). In the latter case, N is a very large value (larger than any cardinality).

An integer range specification consists of one of the following forms: b indicates element b with 0 the least and $N - 1$ the greatest possible value; $b-e$ or $b:e$ inclusive beginning at b ending at e ; $b:s:e$ inclusive from b striding by s and ending no more than e . Either b or e can be omitted specifying an implicit 0 or $N - 1$ (e.g., $b-$ indicates an inclusive range starting at b ending at $N - 1$). Either b or e can be preceded by a \wedge character indicating the value $N - n - 1$ (i.e., $\wedge 0$ is the last element $N - 1$, $\wedge 2$ is $N - 1 - 2$, etc.). A full range specification must indicate a sorted list of elements where each element is specified at most once. A range may also start with the $/$ character indicating a negated (i.e., complemented) range, i.e., all elements between zero and one less than N will be selected except for those that match the range specification following the $/$ character.

For example, if $N = 20$ the range $0, 2, 4-8, 9, 11:4:\wedge 0$ corresponds to the set $0, 2, 4, 5, 6, 7, 8, 9, 11, 15, 19$, the range $1:7:14, 10-12, \wedge 3-\wedge 0$ indicates $1, 8, 10, 11, 12, 16, 17, 18, 19$, and the range $/3, 5, 8-17$ indicates $1, 2, 4, 6, 7, 18, 19$. Also, $0:2:$ specifies the even elements and $1:2:$ specifies the odd ones. A range can also be `nil` or `none` specifying the empty set, or can be `all`, `full`, `-`, or `:` specifying the range $0 - (N - 1)$ (alternatively $0-\wedge 0$).

[Include text about max value for DTs is unknown, since a etc.. value to be equal to the cardinality of the random variable here. We can't do that, however, because the DT is generic, and could be used with multiple different RVs with different cardinalities.]

18.2 Observation/feature file formats

GMTK supports four different types of feature file formats, ICSI Pfiles, HTK files, lists of raw binary features, and lists of plain ASCII features. The type of file format used is determined by the command line arguments given to the program.

The format of ICSI Pfiles is not described in this document. Please refer to the ICSI Sprach distribution [142] for details about Pfiles. Similarly, HTK files are not described in this document; the format of these files is described in the HTK book [453]. It is worth noting here, however, that an HTK feature file is an ASCII file containing a list of files each of which contains the features of an utterance. The format of the feature files for each utterance is binary data consisting of a 12-byte header followed by the features. ICSI Pfiles place all of the utterances and any header information into one large file. Pfiles are advantageous when the training corpus is not very large, as the utterances for an entire training or test set can be very easily manipulated. HTK files are better for larger corpora.

Binary and ASCII files are similar to HTK files, except that the underlying feature files for each utterance are raw binary or ASCII respectively. In other words, an ASCII or binary file consists first of a single ASCII file that specifies a list of other files. Each file in that list must exist, and must contain a raw binary matrix of features (binary file) or a raw ASCII matrix of features. Note that ASCII and binary files can not be intermixed.

Several GMTK programs use the same command line arguments to specify these files, and so are described in this section. They take the form as follows:

<code>-of1 str</code>	<i>Observation File 1 {}</i>
<code>-nf1 unsigned</code>	<i>Number of floats in observation file 1 {0}</i>
<code>-nil unsigned</code>	<i>Number of ints in observation file 1 {0}</i>
<code>-fr1 str</code>	<i>Float range for observation file 1 {all}</i>
<code>-ir1 str</code>	<i>Int range for observation file 1 {all}</i>
<code>-fmt1 str</code>	<i>Format (htk,bin,asc,pfile) for observation file 1 {pfile}</i>
<code>-iswp1 bool</code>	<i>Endian swap condition for observation file 1 {F}</i>

Note that there is the digit “1” after each command line option. This means that these options are for the first file. There are other command line options with a suffix “2” and “3” corresponding to the second and third file respectively. The first, second, and third may be in different formats (i.e., Pfiles, HTK files, etc.) but each such file specification must have the same number of utterances, and the number of frames per utterance must also be the same across files. Some of these files might contain floating point (continuous) value features, others might contain discrete features, and still others might contain a combination of both. The features are combined together into one global observation matrix as described in Section 17.9.1.

We now describe the options in detail.

- `-of1 str` Specifies the file name of the first feature file.
- `-nf1 str` The number of “continuous” floating point values per time frame contained in the feature file.
- `-nil str` The number of integer values contained in the feature file. Note that some file types might have only integers, and others might have only floating point values. HTK files, for example, may contain only integers or floats, but not both. Pfiles may contain floats and/or integers. ASCII and binary files may also contain any number of either integers or floats. Because the program can not in general for all file types figure out how many values are in what format, the user must specify this on the command line.

- `-fr1 str` This option is a string specifying the subset of floating point values, contained in the file, that should be used and placed into the global observation matrix. The default option is to use all values. This means that any arbitrary (possibly sparse) subset of features may be chosen and placed together in consecutive array elements in the global matrix. The format of the range string is a matlab-like list of integers (see Section 18.1).
- `-irl str` This option is similar to `-fr1` except it chooses from the integers in the file.
- `-fmt1 str` This option specifies the format of the file: PFILE, an HTK file, an ASCII file, or a binary file.
- `-iswp1 bool` This option specifies whether to swap the byte order of the binary data as it is read in. This is needed as the data might have been generated on a machine with a different Endian as the one that is currently being used. This option allows files with differing byte orders to be used together. GMTK does not figure out automatically the Endian of the file (which would not be possible anyway for raw binary data).

18.3 Generic Options

In this section, we describe a set of GMTK command-line options that are common to all or most GMTK programs. Each of the separate programs, and their own idiosyncratic options and behaviors, will be described in a separate section below.

18.4 gmtkTriangulate

18.4.0.1 GMTK Triangulation Search Engine

The algorithms above were recently implemented into the GMTK system along with all aforementioned $J()$ functions. Still, a number of ways exist to triangulate a set of partitions. The GMTK triangulation engine solves this using multiple prioritized heuristics. The heuristics include clique size, fill-in, weight, temporal position, file position, user-supplied hint, and random. The heuristics are provided in order by the user. The highest-priority heuristic is used to determine an elimination order, with lower priority heuristics used only to break ties when they occur. GMTK also supports simulated annealing [245] and maximum cardinality search.

If chunks are small enough, it is possible even to exhaustively search all elimination orders. More interestingly, it is possible to produce an exhaustive search over *all* triangulations (the space of triangulations via elimination do not span the space of all triangulations of a graph). In this latter case, it is possible to produce constrained triangulation schemes that lie outside the space of unconstrained triangulations by elimination, sometimes very useful when deterministic and sparse implementations of dependency exist. GMTK supports both methods of exhaustive search.

Users of GMTK, however, often do not wish to concern themselves with the intricacies of graph triangulation. Therefore, GMTK supports a simple *anytime algorithm* where an amount of time is given (1 minute, 2 hours, 3 days, etc.), and the engine searches for the best triangulation possible in that amount of time. We have found this approach quite satisfying from the toolkit user's perspective — one can provide the time they are willing to spend triangulating (a 3-day weekend) before using the graph for research purposes.

18.5 gmtkEMtrain

gmtkEMtrain is the main EM training program for GMTK. It is used for training the parameters of a graphical model based on the data that is contained in a list of observation matrices. This section describes the command line options for this program in detail. Observation feature files and their command line options are described in Section 18.2. First, here is a complete list of gmtkEMtrain’s command line options and their default values.

<-of1 str>	Observation File 1 ()
<-inputMasterFile str>	Input file of multi-level master CPP processed GM input parameters {}
<-strFile str>	Graphical Model Structure File {}
[-nfl unsigned]	Number of floats in observation file 1 {0}
[-nil unsigned]	Number of ints in observation file 1 {0}
[-fr1 str]	Float range for observation file 1 {all}
[-ir1 str]	Int range for observation file 1 {all}
[-fmt1 str]	Format (htk,bin,asc,pfile) for observation file 1 {pfile}
[-iswp1 bool]	Endian swap condition for observation file 1 {F}
[-of2 str]	Observation File 2 ()
[-nf2 unsigned]	Number of floats in observation file 2 {0}
[-ni2 unsigned]	Number of ints in observation file 2 {0}
[-fr2 str]	Float range for observation file 2 {all}
[-ir2 str]	Int range for observation file 2 {all}
[-fmt2 str]	Format (htk,bin,asc,pfile) for observation file 2 {pfile}
[-iswp2 bool]	Endian swap condition for observation file 2 {F}
[-of3 str]	Observation File 3 ()
[-nf3 unsigned]	Number of floats in observation file 3 {0}
[-ni3 unsigned]	Number of ints in observation file 3 {0}
[-fr3 str]	Float range for observation file 3 {all}
[-ir3 str]	Int range for observation file 3 {all}
[-fmt3 str]	Format (htk,bin,asc,pfile) for observation file 3 {pfile}
[-iswp3 bool]	Endian swap condition for observation file 3 {F}
[-cppCommandOptions str]	Additional CPP command line {}
[-outputMasterFile str]	Output file to place master CPP processed GM output parameters {}
[-inputTrainableParameters str]	File of only and all trainable parameters {}
[-binInputTrainableParameters bool]	Binary condition of trainable parameters file {F}
[-objsNotToTrain str]	File listing trainable parameter objects to not train. {}
[-outputTrainableParameters str]	File to place only and all trainable output parameters {}
[-binOutputTrainableParameters bool]	Binary condition of output trainable parameters? {F}
[-wpaeei bool]	Write Parameters After Each EM Iteration Completes {T}
[-seed bool]	Seed the random number generator {F}
[-maxEmIters unsigned]	Max number of EM iterations to do {3}
[-beam float]	Beam width (less than max*exp(-beam) are pruned away) {1.000000e+10}
[-allocateDenseCpts]	Automatically allocate any undefined CPTs.
[-mcvr double]	arg = 1 means use random initial CPT values. arg = 2, use uniform values {0}
[-botForceVanish unsigned]	Mixture Coefficient Vanishing Ratio {1.000000e+20}
[-mcst double]	Number of bottom mixture components to force vanish {0}
[-topForceSplit unsigned]	Mixture Coefficient Splitting Ratio {1.000000e+10}
[meanCloneSTDfrac double]	Number of top mixture components to force split {0}
[covarCloneSTDfrac double]	Fraction of mean to use for STD in mean clone {1.000000e-01}
[dlinkCloneSTDfrac double]	Fraction of var to use for STD in covar clone {1.000000e-01}
[cloneShareMeans bool]	Fraction of var to use for STD in covar clone {0.000000e+00}
[cloneShareCovars bool]	Gaussian component clone shares parent mean {F}
[cloneShareBlinks bool]	Gaussian component clone shares parent covars {F}
[varFloor double]	Gaussian component clone shares parent dlinks {F}
[floorVarOnRead bool]	Variance Floor {1.000000e-10}
[lldp float]	Floor the variances to varFloor when they are read in {F}
[mlldp float]	Log Likelihood difference percentage for termination {1.000000e-03}
[trrrng str]	Absolute value of max negative Log Likelihood difference
[storeAccFile str]	percentage for termination {1.000000e-02}
[loadAccFile str]	Range to train over segment file {all}
[loadAccRange str]	Store accumulators file {}
[llStoreFile str]	Load accumulators file {}
[accfileIsBinary bool]	Load accumulators file range {}
[startSkip integer]	File to store previous sum LL's {}
[endSkip integer]	Binary accumulator files (def true) {T}
[cptNormThreshold double]	Frames to skip at beginning (i.e., first frame is buff[startSkip]) {0}
[random bool]	Frames to skip at end (i.e., last frame is buff[len-1-endSkip]) {0}
[enem bool]	Read error if Sum1-0 /card > norm_threshold {1.000000e-02}
[showCliques integer]	Randomeize the parameters {F}
[numSplits integer]	Run enumerative EM {F}
[baseCaseThreshold integer]	Show the cliques after the network has been unrolled k times. {0}
[gaussianCache bool]	Number of splits to use in logspace recursion (>=2). {3}
[argsFile <str>]	Base case threshold to end recursion (>=2). {1000}
	Cache Gaussians evaluations during EM training.
	Will speeds things up, but uses more memory. {T}
	File to obtain additional arguments from {}

- **-of1 str** The name of the first feature file. Since feature file arguments are described generically above in Section 18.2, this option, and the options for the other two feature files are not describe here.
- **-cppCommandOptions str** This is a string that gives additional commands to the CPP command

line. This makes it possible, for example, to make CPP defines on the GMTK command line. This can be useful for specifying paths of certain files, ranges to use, and so on. See your local `cpp` manual page to see the syntax of options it supports, but note that typically, if you want to define a CPP variable `FOO` to have value, say, 5, you would give the option `-DFOO=5`.

- `-outputMasterFile str` This option gives the file containing the output master file, as described in Section 16.4.3.
- `-inputTrainableParameters str` The file containing trainable parameters in a pre-defined format, as described in Section 16.4.3.
- `-binInputTrainableParameters bool` A Boolean specifying whether or not the input trainable parameters file should be in binary or ASCII format. Like all Boolean arguments, it can be T/F, 0/1, true/false and so on.
- `-objsNotToTrain str` This option specifies a file that contains a listing of all GMTK objects that should be held fixed during training, rather than be updated at the end of each training iteration. The format of this file is ASCII, and is pre-processed by CPP. The file itself consists of a collection of keywords followed by object names. The keywords specify the type of object (the namespace if you will), and the object name specifies the name of the object in that object name space to not train. The keywords may be any of the following, corresponding to the object types listed in Table 16.1.

```
DPMF
SPMF
MEAN
COVAR
DLINKMAT
WEIGHTMAT
GAUSSIANCOMPONENT
DENSECPT
SPARSECPT
DETERMINISTICCPT
MIXGAUSSIAN
```

The name of the object can either be the object name, or the special string `*` which states that all objects of that kind should not be trained. For example, in the following file

```
MEAN foo
COVAR bar
MEAN baz
DENSECPT *
```

it states that the mean objects called `foo` and `baz`, the covariance object called `bar`, and all dense CPTs should be held fixed during EM training.

- `-outputTrainableParameters str` This option specifies a file in which to place all *trainable* output parameters. The parameters are written in a specific order, as described in Section 16.4.3.
- `-binOutputTrainableParameters bool` A Boolean specifying if the output trainable file should be ASCII or binary.

- `-wpaei` `bool` A Boolean that determines if all of the parameters should be written after each EM iteration completes, rather than only once at the end after EM has converged.
- `-seed` `bool` Boolean stating if the random number should be seeded.
- `-maxEmIters` `unsigned` An integer stating the maximum number of EM iterations that should be used during training.
- `-beam` `float` The beam width parameter, which is a floating point value that controls how GMTK does pruning. During each message pass between cliques in a graphical model, values that are less than $m e^{-\text{beam}}$ are pruned away, where m is the maximum clique probability. This is a generalization of standard beam pruning in speech recognition systems.
- `-allocateDenseCpts` `unsigned` Normally, all CPTs must be declared in parameter files for them to be usable as objects in a structure file. With this option, it is possible to automatically allocate dense CPTs that do not exist in the parameter file, just by naming them in a structure file. For setting up initial experiments, this option therefore might make it easier to get going. The unsigned argument to this command determines how the parameter values are initialized. If the value is zero (0), then CPTs are not allocated (the default). If the value is one (1), it means that random initial CPT values should be used. If the argument is two (2), then uniform initial values are used. See also Section 18.10 for information on how to allocate initial values for these parameters without needing to run any EM training.
- `-mcvr` `double` This is the mixture coefficient vanishing ratio (MCVR) which determines when components in a Gaussian mixture should “vanish” when the component responsibility gets too small. The MCVR works as follows. Let the i^{th} component’s responsibility be p_i . At the end of an EM iteration, the component will vanish if $p_i < 1/(N \times \text{MCVR})$ where N is the number of components in the distribution. Therefore, each mixture has its own vanishing threshold depending on the number of components that it currently uses.

The larger MCVR becomes, the less chance that a vanish will occur. To turn off vanishing completely, set this parameter to something in the range of 10^{10} . A typical value to allow for some vanishing is to set MCVR to about 50. To allow for aggressive vanishing, set MCVR to about 5 or so.

- `-botForceVanish` `unsigned` This next option is another way to vanish Gaussian components. This option specifies the number of mixture components that should be forced to vanish, regardless of the vanishing ratio. The parameter that is specified here (say V) determines the number of components that should vanish, and it vanishes the components with the smallest V responsibilities.
- `-mcsr` `double` The mixture coefficient splitting ratio (MCSR) is the analog to the MCVR. GMTK’s splitting algorithm is similar to the HTK algorithm for mixing up Gaussian components. In GMTK, the criterion can be controlled by the MCSR. A Gaussian component is split if its responsibility becomes larger than a threshold. Again, if p_i is the i^{th} component’s responsibility, the component is split if $p_i > \text{MCSR}/N$ where again N is the current number of components.

Like the MCVR, the larger MCSR becomes, the less the chance will be that a split occurs. To turn off splitting, set MCSR to be about 10^{10} . To force a split of all Gaussian components, set MCSR to be something very small (or see the `-topForceSplit` option).

In general, the MCVR/MCSR ratios should be set differently for each EM training iteration. I.e., one should **not** set them to be the same for each such training iteration, as that would lead to an ever splitting/vanishing set of Gaussians. When training up a system, it is best to split at one EM iteration,

and then allow things to stabilize for a few EM iterations without having any splitting or vanishing occur (or perhaps having mild vanishing during this time). After things have stabilized, it is often useful to repeat the cycle: aggressive splitting for 1 EM iteration, no splitting and very mild vanishing for about 4-6 EM iterations, and then repeat. In general, the set of values that MCVR/MCSR are set to at each EM iteration is called the splitting/vanishing *schedule*.

Here is a schedule¹ that was successfully used on the Aurora 2.0 noisy speech task with a phone-based GMTK recognizer. The schedule is not “optimized” in any way (meaning that other schedules might do better) but this one seems to work fairly well.

Step 0: Start the training with 1 Gaussian component per mixture

Step 1: Train multiple EM iterations with no splitting or vanishing until you have reached 2% convergence. This means that the relative difference in the global log likelihood is less than 2%. You can get the global log-likelihood at the end of each EM iteration using the option `-llStoreFile` which will store that number (in ASCII) to a file.

Step 2: Do an iteration that splits all Gaussians, and then continue training with no splitting/vanishing until 2% convergence ratio is achieved.

Repeat steps 1 and 2 a few times (typically 5). At this point it becomes dangerous to split all Gaussians because some of them will start to have determinants that reach the minimum, which will kill the job. So, then we start vanishing:

Step 3: Run one EM iteration with `mcvr = 10` and `mcsr = 1`.

Step 4: Continue training with no splitting/vanishing until 2% convergence.

Repeat steps 3 and 4 a few times. After a while some of the Gaussians again might start to have determinants that hit the minimum, so we then do:

Step 5: Set `mcvr = 10, mcsr = 1` for 1 EM iteration.

Step 6: Set `mcvr = 10, mcsr = 1e10` for 1 iteration (just to kill off some of the weak Gaussians from step 5)

Step 7: Continue training with splitting/vanishing until 2% convergence.

Repeat steps 5, 6, and 7 a few times. Finally to do the final following steps to achieve convergence:

Step 8: Set `mcvr = 10, mcsr = 5` for 2 iterations.

Step 9: Train with no splitting/vanishing until 0.2% convergence.

It was found that one can pretty much go on and on with this schedule, and performance keeps improving at least until 15k Gaussians (or so) are used.

Note that testing for relative convergence between an EM iteration when splitting and/or vanishing is allowed will not work, because if the two iterations you’re comparing have different numbers of Gaussians then the EM likelihood is not guaranteed to increase (and is therefore not comparable). Therefore, when testing for relative convergence, make sure that it is between two EM iterations in which no splitting or vanishing has occurred.

In general, the splitting/vanishing schedule is more art than science. If you happen to discover a schedule that works well, then please let me ([JB](#)) know about it.

¹Courtesy of Karen Livescu from MIT

- `-topForceSplit unsigned` In some cases, it can be useful to have the components with the top (largest) responsibility split. This option does that, and forces, at the end of an EM iteration, the splitting of those most probable components.
- `-meanCloneSTDfrac double, -covarCloneSTDfrac double, -dlinkCloneSTDfrac double`. When a split of a component Gaussian occurs, the component splits into two components. One of the two new components is the same as the old component. The other new component (the cloned component) is a random function of the old component, and these options determine the parameters of the random transformation. Essentially a clone is made of the Gaussian G , so the original G is left alone and its clone G' is randomly perturbed. As mentioned in Section 17.5.1, GMTK represents Gaussians in terms of their means, variances, and dlink matrices. Based on this Gaussian decomposition, there are a number of options when this occurs.

The `-meanCloneSTDfrac double` option determines the fraction of the original mean to use for the standard deviation in a Gaussian that is used to form the new mean as a function of the old mean. In other words,

$$\mu' = \mu + \text{meanCloneSTDfrac} \mu \mathcal{N}(0, 1)$$

where μ' is the new mean, μ is the old mean, and $\mathcal{N}(0, 1)$ is a sample from a zero mean unity variance Gaussian. This means that setting `meanCloneSTDfrac` to zero causes the mean of the cloned Gaussian to be equal to the original. A typical value for this parameter that has worked in the past is anywhere from 0.1 to 0.25, but this will not necessarily be the “best” for a given application.

The option `-covarCloneSTDfrac double` is the fraction of the original variance to use for the standard deviation in a Gaussian that is used to form the new variance as a function of the old variance. Therefore, this option is similar to the case for the mean, specifically:

$$D' = D + \text{covarCloneSTDfrac} D \mathcal{N}(0, 1)$$

So far, the value of 0.0 has been tried for this parameter.

Lastly, for the dlink matrix B , the `-dlinkCloneSTDfrac double` option exists. Again, only value 0.0 has been tried for this parameter.

- `-cloneShareMeans bool, -cloneShareCovars bool, -cloneShareDlinks bool`. When a split occurs and a clone is made of a component, GMTK provides the option to have the mean, variance, and/or dlink matrix of the clone to be shared with the original component, and these options determine if any of the Gaussian portions should be shared after a clone. Of course, if a portion is shared, there is no random transformation between the old and new portion.

Note that if any of these options are on, that implies that the Gaussian component and its clone will have a portion of itself (i.e., either a mean, diagonal covariance matrix, or B matrix) tied for the remainder of the life of the two components. If one of the components vanishes, then the shared portion will remain attached to the component that survives.

In general, these options provide a way of sharing portions of Gaussians without having to explicitly or by-hand specify how to do sharing in the parameter files. With these options turned on, sharing can start occur while growing the Gaussians from one component to multi-component mixtures.

- `-varFloor double` Controls the variance floor, meaning that if any of the diagonal variances of a Gaussian falls below this value, then the variance is “floored” (prohibited from falling below the floor value). The variance, in this case, is set to the corresponding variance from the previous EM iteration.

- `-floorVarOnRead bool` In certain cases, it is useful to have the variances floored to the floor value right when the parameters are read in (if for example the parameters were trained using a lower `-varFloor` value, but a larger value is now desired). This option controls this behavior.

Note that neither `-varFloor` nor `-floorVarOnRead` will directly change and/or affect the vanishing algorithm. It is in general the case, however, that when the variances get very small, the corresponding mixture coefficient responsibility will have a very low value, and is more likely to be vanished. Therefore, if `varFloor` is decreased, that can have the indirect effect of allowing components to have smaller responsibilities (smaller at least than if `varFloor` was larger). This might then encourage more vanishing. One must be careful, however, as if `varFloor` is very small, and the variances themselves get too small, then other numerical issues will arise (GMTK will produce numerous warning messages about them).

Similarly, more aggressive vanishing (lower MCVR) can be performed to try to eliminate those small variance components, something that might allow for a larger `-varFloor`.

In general, the vanishing and variance floor options will interact with each other in interesting and mysterious ways.

- `-lldp float` This option determines the log likelihood difference percentage (`lldp`) for termination of an EM iteration. If the overall percent difference between the previous EM iteration's log likelihood and the current iteration's log likelihood falls below this threshold, EM training will end.
- `-mnlldp float` NOT CURRENTLY USED.
- `-trrng str` Gives an integer range of training utterance numbers to train over, using the standard integer range specification (Section 18.1).
- `-storeAccFile str` After an EM iteration or a partial EM iteration, this option gives the file name in which to store all the EM accumulators for all trainable objects. If this option is specified, then after forming the accumulators by doing a pass through the training data, the accumulators are written to a file, and then the program exits (i.e., it does not update the parameters). The reason for this is that the accumulators are then used in another run of the training program which accumulates together all of the accumulators that were created by different runs of `gmtkEMtrain`, and updates the parameters. This option is useful for parallel training, as described in Section 18.5.1.
- `-loadAccFile str` Before starting an EM iteration, it is possible to load initial accumulators. This option is used with parallel training.
- `-loadAccRange str` This option provides an integer range (Section 18.1) that is used to choose and automatically load a number of different accumulator files, each of which have been generated by different partial runs of `gmtkEMtrain`. If the file name given by the `-loadAccFile` option contains the string `@D` then for each integer given in the range, the `@D` is substituted by each range integer, and the corresponding accumulator file is loaded.

As an example, suppose the `-loadAccFile` file is given as `foo@D.acc` and that the range is given as `0 : 3`. Then the following four accumulator files will be assumed to exist and will be loaded: `foo0.acc`, `foo1.acc`, `foo2.acc`, `foo3.acc`

- `-llStoreFile str` Specifies the file to store the overall log likelihood of the training data. This is used to determine convergence during external and parallel EM training (see Section 18.5.1).
- `-accFileIsBinary bool` Set to true if the accumulator files are in binary rather than ASCII. Since the accumulator files can be large, this option should almost always be true (the default).

- `-startSkip integer` When using dlinks into the past, it is important that the parents of children at early frames do not reference to frames before the beginning of the global observation matrix. If a dlink dependency specifies a frame before the global matrix, then a run-time error will occur. This option says that some number of frames should be skipped at the beginning of each utterance. Those skipped frames may be parents in a dlink matrix (since elements in the global matrix exist for these now negative frame indices).
- `-endSkip integer` This is the analog to `-startSkip` except it operates on the end of the observation matrix, since dlinks are allowed to specify parents in the future as well as the past.
- `-cptNormThreshold double` GMTK will check to ensure that, after they are read in, the CPTs are appropriately normalized (i.e., sum to unity). This option controls the unity tolerance. A read error will occur if there exists a CPT such that $|S - 1.0|/C > \tau$ where S is the sum over the CPT probabilities (which should be unity), C is the cardinality of the random variable (which means that the tolerance automatically increases for greater cardinality variables), and τ is the threshold given as a command line argument.
- `-random bool` True if all parameters should be randomized after being read in.
- `-enem bool` True if enumerative EM should be run. This will be extremely slow as it does not use junction trees at all. This option is there only to verify probability values.
- `-showCliques integer` This option instructs GMTK to unroll the graph some number of times (given by the command line integer argument), triangulate the graph, and then print out the resulting cliques and their sizes. This can be useful to examine the cliques and thereby obtain information on the running time of the graph that you are using.
- `-numSplits integer` During log-space inference (Section XXX), this argument determines the number of splits to use when training. Essentially, this argument determines the base of the logarithm when we say that the running time goes from $O(ST)$ (and respectively space goes from $O(ST)$) during exact inference to $O(ST \log T)$ time (respectively $O(S \log T)$ space) in log-space inference. In general, the larger the value, the less space will be required.
- `-baseCaseThreshold integer` When the number of frames falls below a threshold, log-space inference starts using exact space inference. In other words, this option specifies the base case threshold to end the log-space recursion.
- `-gaussianCache bool` This option determines if Gaussian probability evaluations are cached during EM training. If this is true, the program will run faster, but will use more memory. The amount of extra memory is not that large, however, so there is almost no reason not to keep this set to true.
- `-argsFile <str>` Any of the command line arguments can be obtained from a file rather than from the command line. This option specifies a file from which options are to be obtained. The file consists of keyword-value pairs separated by a colon, where the keywords are the same as the command line parameters without the dash. Argument files may be interspersed with command line arguments, and the order of processing is sequentially through the command line, descending into files where argument files are given. For example, the following is a valid argument file:

```
strFile: foo.str
inputMasterFile: foo.master
of1: foo.htk
```

```
nf1: 39
fmt1: htk
```

18.5.1 gmtkEMtrain, EM iterations, and parallel training

As was alluded to above, gmtkEMtrain supports parallel EM training. In fact, there are two ways to run EM training, both internal to the program and external. In internal EM training, multiple EM iterations are run for one single invocation of the gmtkEMtrain program. In external EM training, each EM iteration corresponds to running gmtkEMtrain $M + 1$ times. It is run M times to create M different accumulator files, where each accumulator file corresponds to a different set of observation files, and is run a final time to bundle together all the accumulator files, update the parameters, and write the new parameters out to a file. In this latter mode of processing, each of the M runs of gmtkEMtrain can be run simultaneously on different processors, which is how parallel EM training works.

The following is a simple example of a script that runs one iteration of EM externally using gmtkEMtrain. It runs gmtkEMtrain $M + 1 = 3$ times (so that $M = 2$). It does not run the programs in parallel, but it can easily be seen where the parallelism arises.

Basically, you need to call gmtk multiple times to generate accumulator files, and then you call it one last time to accumulate the accumulator files together and do the parameter update. A typical call to to produce an accumulator file is:

```
gmtkEMtrainNew -of1 ./switchboard.train.features.v1.lst -nf1 39 -fmt1 htk \
-iswp1 true -inputMasterFile PARAMETERS/masterFile \
-strF PARAMETERS/lattice.str -random F -varFloor 1e-5 \
-checkTriFileCards F -accFileIsBinary true \
-maxEmIters 1 -storeAccFile acc_1.data -trrng 0:74
```

This will train only on utterances 0:74 and then produce accumulator file acc_1.data

To combine accumulator files together, do:

```
gmtkEMtrainNew -of1 ./switchboard.train.features.v1.lst -nf1 39 -fmt1 htk \
-iswp1 true -inputMasterFile PARAMETERS/masterFile \
-strF PARAMETERS/lattice.str -random F -varFloor 1e-5 -checkTriFileCards F \
-maxEmIters 1 -accFileIsBinary true -llStoreFile MISC/.ll.txt -mcsr 1e20 \
-mcvr 1e20 -loadAccRange 1:500 -loadAccFile MISC/acc_@D.data -trrng nil
```

Which will gather all accumulator files named acc_@D.data where the string "@D" is replaced by integers in the range 1 through 500. These are accumulated and the new parameters are produced and saved as normal.

18.5.2 gmtkEMtrain tips

In this section, we list a number of tips for using `gmtkEMtrain`.

18.5.2.1 Determinants becoming too small

In some cases, you might find that `gmtkEMtrain` reports that determinants of Gaussians become smaller than is possible for the finite precision IEEE floating point arithmetic to handle. This happens only when the determinant of a covariance matrix (something that `gmtkEMtrain` needs to compute for doing an EM update) becomes smaller than the minimum possible double precision floating point value (i.e., there is no threshold that can be adjusted in this case).

In general, when this happens there are several possible solutions:

- use more training utterances, as this often occurs when a component does not get enough counts.
- make sure the observations are scaled appropriately. For example, if you take a set of observation features, and divide each feature by, say, 1e10, there is no theoretical information loss, but the variances of Gaussians (and therefore the determinants) will become much smaller thereby potentially leading to a minimum variable problem. If such a thing happens, you can try simply multiplying all your features by, say, 1000 so that the features are in the range more appropriate for finite precision arithmetic.
- Make sure that the features were generated with the appropriate options. There are many options when using MFCCs, for example. Also, it is often the case that brand new feature sets (something that GMTK with BMMs and dlink matrices is uniquely suitable for) are not appropriately scaled at first.

18.6 obs-tools

18.7 gmtkViterbi

`gmtkViterbi` is the main GMTK decoding program. Many of the command line options are the same as for `gmtkEMtrain`, so only the ones that are unique to `gmtkViterbi` will be described here.

Currently, `gmtkViterbi` works by running a single pass of max-product inference over the network. This means that the procedure finds the single most probable configuration of all hidden variables. Once these hidden variables are instantiated (such as a word variable), the variables over time may optionally be written out to disk. The time frame that the instantiated hidden variable is written out may also be determined using another hidden variable (such as a transition variable). At the moment, GMTK does not allow for some variables to be integrated (via summation) and others to be maxed, but this will change in future versions of GMTK.

Here is the complete set of command line options.

```

<-of1 str>          Observation File 1 {}
<-strFile str>      GM Structure File {}
<-inputMasterFile str> Input file of multi-level master CPP processed GM input parameters {}
[-nf1 unsigned]      Number of floats in observation file 1 {0}
[-ni1 unsigned]      Number of ints in observation file 1 {0}
[-fr1 str]           Float range for observation file 1 {all}
[-ir1 str]           Int range for observation file 1 {all}
[-fmt1 str]          Format (htk,bin,asc,pfile) for observation file 1 {pfile}
[-iswp1 bool]         Endian swap condition for observation file 1 {F}
[-of2 str]           Observation File 2 {}
[-nf2 unsigned]      Number of floats in observation file 2 {0}
[-ni2 unsigned]      Number of ints in observation file 2 {0}
[-fr2 str]           Float range for observation file 2 {all}
[-ir2 str]           Int range for observation file 2 {all}
[-fmt2 str]          Format (htk,bin,asc,pfile) for observation file 2 {pfile}

```

```

[-iswp2 bool]           Endian swap condition for observation file 2 {F}
[-of3 str]              Observation File 3 {}
[-nf3 unsigned]         Number of floats in observation file 3 {0}
[-ni3 unsigned]         Number of ints in observation file 3 {0}
[-fr3 str]              Float range for observation file 3 {all}
[-ir3 str]              Int range for observation file 3 {all}
[-fmt3 str]             Format (htk,bin,asc,file) for observation file 3 {file}
[-iswp3 bool]            Endian swap condition for observation file 3 {F}
[-binInputTrainableFile str] File of only and all trainable parameters {}
[-inputTrainableFile str] Binary condition of trainable parameters file {F}
[-cppCommandOptions str] Command line options to give to cpp {}
[-varFloor double]       Variance Floor {1.000000e-10}
[-floorVarOnRead bool]   Floor the variances to varFloor when they are read in {F}
[-dcdrng str]            Range to decode over segment file {all}
[-beam float]            Beam width (less than max*exp(-beam) are pruned away) {1.000000e+10}
[-showVitVals bool]      Print the viterbi values?? {F}
[-printWordVar str]       Print the word var - which has this label {}
[-varMap str]             Use this file to map from word-index to string {}
[-transitionLabel str]   The label of the word transition variable {}
[-startSkip integer]     Frames to skip at beginning (i.e., first frame is buff[startSkip]) {0}
[-endSkip integer]       Frames to skip at end (i.e., last frame is buff[len-1-endSkip]) {0}
[-cptNormThreshold double] Read error if |Sum-1.0|/card > norm_threshold {1.000000e-02}
[-showCliques integer]   Show the cliques after the network has been unrolled k times. {0}
[-dumpNames str]          File containing the names of the variables to save to a file {}
[-filelist str]            List of filenames to dump the hidden variable values to {}
[-numSplits integer]      Number of splits to use in logspace recursion (>=2). {3}
[-baseCaseThreshold integer] Base case threshold to end recursion (>=2). {1000}
[-argsFile <str>]         File to obtain additional arguments from {}

```

Let us now go through each option. Again, the ones that are already described above are not repeated here.

- `-dcdrng str` This option provides the integer range (see Section 18.1) over the utterance file to decode.
- `-showVitVals bool` This is a Boolean that, if true, states the the Viterbi values (the maximum probability values) of the hidden variables should be printed out.
- `-printWordVar str` Gives the name of the variable (typically a word variable) that should be printed out.
- `-transitionLabel str` Gives the name of a binary transition variable that, when unity, determines when (i.e., for which frames) the word variable given by `-printWordVar` should be printed out.
- `-varMap str` Gives a name of an ASCII file that contains a list of words. This is the mapping from integer word variable value to actual textual word string, and these strings are used for printing rather than the integers.
- `-dumpNames str` The file that contains a list of the names of all the variables that should be saved to a file.
- `-filelist str` The name of the file that contains a list of filenames to which the variables values listed in `-dumpNames` should be written. The number of file names listed in the file should be the same as the number of utterances decoded.

18.8 gmtkScore

`gmtkScore` is the main GMTK re-scoring program, for producing a score from a given model. I.e., it simply integrates over all hidden variables, and computes $P(X)$, where X are all the observed variables, and then prints out $\log P(X)$.

Here is the complete list of `gmtkScore`'s options.

```

<-of1 str>          Observation File 1 {}
<-strFile str>        GM Structure File {}
<-inputMasterFile str> Input file of multi-level master CPP processed GM input parameters {}
[-nf1 unsigned]      Number of floats in observation file 1 {0}
[-nil unsigned]      Number of ints in observation file 1 {0}
[-fr1 str]           Float range for observation file 1 {all}
[-ir1 str]           Int range for observation file 1 {all}
[-fmt1 str]          Format (htk,bin,asc,pfile) for observation file 1 {pfile}
[-iswp1 bool]        Endian swap condition for observation file 1 {F}
[-of2 str]           Observation File 2 {}
[-nf2 unsigned]      Number of floats in observation file 2 {0}
[-ni2 unsigned]      Number of ints in observation file 2 {0}
[-fr2 str]           Float range for observation file 2 {all}
[-ir2 str]           Int range for observation file 2 {all}
[-fmt2 str]          Format (htk,bin,asc,pfile) for observation file 2 {pfile}
[-iswp2 bool]        Endian swap condition for observation file 2 {F}
[-of3 str]           Observation File 3 {}
[-nf3 unsigned]      Number of floats in observation file 3 {0}
[-ni3 unsigned]      Number of ints in observation file 3 {0}
[-fr3 str]           Float range for observation file 3 {all}
[-ir3 str]           Int range for observation file 3 {all}
[-fmt3 str]          Format (htk,bin,asc,pfile) for observation file 3 {pfile}
[-iswp3 bool]        Endian swap condition for observation file 3 {F}
[-inputTrainableFile str] File of only and all trainable parameters {}
[-binInputTrainableFile bool] Binary condition of trainable parameters file {F}
[-cppCommandOptions str] Command line options to give to cpp {}
[-varFloor double]   Variance Floor {1.000000e-10}
[-floorVarOnRead bool] Floor the variances to varFloor when they are read in {F}
[-ddcrng str]        Range to decode over segment file {all}
[-beam float]         Beam width (less than max*exp(-beam) are pruned away) {1.000000e+10}
[-startSkip integer] Frames to skip at beginning (i.e., first frame is buff[startSkip]) {0}
[-endSkip integer]   Frames to skip at end (i.e., last frame is buff[len-1-endSkip]) {0}
[-cptNormThreshold double] Read error if |Sum-1.0|/card > norm_threshold {1.000000e-02}
[-showCliques bool]  Show the cliques of the not-unrolled network {F}
[-numSplits integer] Number of splits to use in logspace recursion (>=2). {3}
[-baseCaseThreshold integer] Base case threshold to end recursion (>=2). {1000}
[-argsFile <str>]     File to obtain additional arguments from {}

```

All of the options for gmtkScore have already been described.

18.9 gmtkSample

gmtkSample allows you to sample from a graphical model. It currently only samples discrete variables. In the next release, GMTK will allow sampling of both discrete and continuous variables.

Here is the complete list of command line options.

```

Required: <>; Optional: [] ; Flagless arguments must be in order.
<-of1 str>          Observation File 1 {}
<-strFile str>        GM Structure File {}
<-inputMasterFile str> Input file of multi-level master CPP processed GM input parameters {}
<-dumpNames str>    File containing the names of the variables to save to a file {}
<-ofilelist str>    List of filenames to dump the hidden variable values to {}
[-nf1 unsigned]      Number of floats in observation file 1 {0}
[-nil unsigned]      Number of ints in observation file 1 {0}
[-fr1 str]           Float range for observation file 1 {all}
[-ir1 str]           Int range for observation file 1 {all}
[-fmt1 str]          Format (htk,bin,asc,pfile) for observation file 1 {pfile}
[-iswp1 bool]        Endian swap condition for observation file 1 {F}
[-of2 str]           Observation File 2 {}
[-nf2 unsigned]      Number of floats in observation file 2 {0}
[-ni2 unsigned]      Number of ints in observation file 2 {0}
[-fr2 str]           Float range for observation file 2 {all}
[-ir2 str]           Int range for observation file 2 {all}
[-fmt2 str]          Format (htk,bin,asc,pfile) for observation file 2 {pfile}
[-iswp2 bool]        Endian swap condition for observation file 2 {F}
[-of3 str]           Observation File 3 {}
[-nf3 unsigned]      Number of floats in observation file 3 {0}
[-ni3 unsigned]      Number of ints in observation file 3 {0}
[-fr3 str]           Float range for observation file 3 {all}
[-ir3 str]           Int range for observation file 3 {all}
[-fmt3 str]          Format (htk,bin,asc,pfile) for observation file 3 {pfile}
[-iswp3 bool]        Endian swap condition for observation file 3 {F}
[-inputTrainableFile str] File of only and all trainable parameters {}
[-binInputTrainableFile bool] Binary condition of trainable parameters file {F}
[-cppCommandOptions str] Command line options to give to cpp {}
[-samplerng str]     Range to decode over segment file {all}
[-startSkip integer] Frames to skip at beginning (i.e., first frame is buff[startSkip]) {0}
[-endSkip integer]   Frames to skip at end (i.e., last frame is buff[len-1-endSkip]) {0}
[-argsFile <str>]     File to obtain additional arguments from {}

```

All of these options have been described above for different programs.

18.10 gmtkParmConvert

gmtkParmConvert is the program that can convert ASCII to binary files and vice versa, and can automatically allocate and print dense CPTs for you.

Note that it is crucial to convert parameter files to binary as soon as possible since reading them in ASCII is doubly slow – ASCII files are first processed by CPP, and it further takes time to convert the parameter values from ASCII to binary. It is not unheard of, in fact, for CPP to take 20 minutes to process large ASCII files.

Once files are converted to binary, they can easily be reconverted to ASCII for quick viewing using gmtkParmConvert. Also, the special file name “-” indicates that that parameters should be written to (respectively, read from) standard output `stdout` (respectively, standard input `stdin`), depending on if the file is an input or output file.

As mentioned in Section 18.5, GMTK has the ability to automatically allocate the parameter objects for dense CPTs. Both the `gmtkEMtrain` and the `gmtkParmConvert` program can do this for you, using the `-allocateDenseCpts` option. `gmtkParmConvert` is a particularly convenient way of doing this as a script need not produce the parameter file objects needed for all of the dense CPTs.

Here is the complete set of command line options:

<code>[-inputMasterFile str]</code>	Input file of multi-level master CPP processed GM input parameters {}
<code>[-outputMasterFile str]</code>	Output file to place master CPP processed GM output parameters {}
<code>[-allocateDenseCpts]</code>	Automatically allocate any undefined CPTs.
<code>[-inputTrainableParameters str]</code>	File of only and all trainable parameters {}
<code>[-binInputTrainableParameters bool]</code>	Binary condition of trainable parameters file {F}
<code>[-outputTrainableParameters str]</code>	File to place only and all trainable output parameters {}
<code>[-binOutputTrainableParameters bool]</code>	Binary condition of output trainable parameters? {F}
<code>[-cppCommandOptions str]</code>	Command line options to give to cpp {}
<code>[-varFloor double]</code>	Variance Floor {1.000000e-10}
<code>[-floorVarOnRead bool]</code>	Floor the variances to varFloor when they are read in {F}
<code>[-cptNormThreshold double]</code>	Read error if Sum-1.0 /card > norm_threshold {1.000000e-02}
<code>[-argsFile <str>]</code>	File to obtain additional arguments from {}

All of these options have been described above in `gmtkEMtrain`.

Chapter 19

GMTK Under the hood

In this section, we describe features of GMTK that are most interesting to how to do inference.

- The new inference engine uses partition-based graph structures and partition based junction trees, as described in our previous status report (and paper [34]). The *partition* is a portion of a junction tree that corresponds to a piece of a graph that is triangulated separately. Rather than unrolling the graph up to length N and then triangulating it at the very end, the new GMTK system will first form a set of small partitions, will triangulate them separately, and then will then unroll the junction tree (in the form of a series of partitions) to the appropriate length of the given utterance or sentence. In addition to this, the best *boundary* between partitions is found using the boundary algorithm [34].
- The GMTK triangulation engine has been extended to include all of the theoretically motivated new triangulation procedures mentioned above (Section ??) which are appropriate for models with significant amounts of determinism, and as exists in most speech and language models.
- As is well known, finding the optimal triangulation is an NP-complete problem, and therefore heuristics must be used. The GMTK triangulation engine has been extended to include an *anytime* triangulation algorithm. In an anytime algorithm, the user specifies an amount of time she wishes to spend triangulating (say 20 seconds, an hour, a 3-day weekend, etc.), and the engine will return the best triangulation it can find in the slated amount of time.

The underlying goal here is to avoid requiring the speech/language researcher/user from needing to spend time thinking about how best to triangulate their graph (something which can be quite technically involved, and can easily distract the researcher from their main task). Having an anytime triangulation system enables the researcher to concentrate on their own goal (namely, finding a better model for speech/ language, and using the language of graphical models to express this new idea) rather than be distracted by the intricacies of graph triangulation. This anytime algorithm is now implemented and working.

- GMTK now has new customized data-structures for sparse-join procedures to avoid iterating over probabilities that are zero. A key issue for performing probabilistic inference efficiently in the presence of much determinism and/or pruning can be relatively simply described as finding the best data structure for performing the following table multiplication:

$$f(a, b, c, d, e, f) = \phi_1(a, b, c)\phi_2(b, c, d)\phi_3(d, c)\psi(a, b, c, d, e, f)$$

One way of doing this would be to iterate over all possible values of for the variables of the ψ function, or a, b, c, d, e, f . For each such value combination perform the stated multiplication. If each of the variables above had 100 possible values, this would require 100^6 operations. The key problem,

however, is that all of the functions $\phi_1, \phi_2, \phi_3, \psi$ are extremely sparse, containing many zeros, so the naive approach of iterating through all values and multiplying would be extremely wasteful, most of the computation spent multiplying by and filling in a f -function table with zero values.

Instead of the naive approach mentioned above, the new GMTK inference engine maintains a novel data structure, what we call a *separator-driven sparse-join*. A basic feature of this data structure is that only the non-zero entries of each of the functions are stored in memory. But in addition to this, there are two steps that achieve a significant speedup and memory reduction. First, the iteration of the variables is *separator driven*, meaning that the variable iteration proceeds in an order which is determined by an ordering of the ϕ functions. In addition to this, the ϕ functions are represented in such a way so that the set of variables for the ϕ functions are partitioned into two portions: 1) the set of variables for a ϕ function that intersects previous ϕ function's variables (according to the ϕ function order), and 2) the residual (or innovation) set of variables for the ϕ function, that are *not* contained in previous ϕ functions (again according to the ϕ order). For example, the representation for ϕ_2 above would contain a portion for b, c (since it is part of ϕ_1), and a portion just for d (since it is not part of ϕ_1). Each portion of the function consists of a shared data value table and also a hash table for quick access to that portion values. Therefore, when the ϕ functions are iterated, only the “compatible” (non-zero probability) sections of the the later ϕ functions are consulted. If a hash table lookup for the intersection variables of any ϕ function returns empty, then it is immediately known that *all* additional entries corresponding to the variables set so far are zero, and iteration then backs out to the previous ϕ function step. Thus, an enormous portion of the zeros (in fact all zeros that are implicitly represented by a ϕ function not containing an entry for those zeros), are removed before even touching the ψ function. This can lead to orders of magnitude reductions for some distributions.

Note that this representation is quite distinct from a typical sparse matrix array representation, as the internal representation of each ϕ function depends entirely on all of the other ϕ functions, and iteration is specifically separator (or ϕ -function) driven. A sparse matrix alone would not have these properties. As an added advantage, the separator driven procedure fully supports and takes advantage of pruning (both at the separator and clique level). And just as importantly, this procedure removes entries that are zero because of a deterministic or sparse CPT entry in GMTK.

The separator-driven sparse join data structure is now implemented in the new GMTK core engine, and is used in the collect evidence stage of probabilistic inference. This has resulted in significant speedups relative to the more naive clique- (or ψ -function) driven iteration mentioned above.

- Ancestral assignments of variables to cliques. A new algorithm to assign variables to cliques was developed. When a variable is assigned, and if it is deterministic, it is used to determine which combinations of parent and child values have zero probability (and therefore do not deserve any allocated memory). Note that variables assigned to a clique are used in this fashion regardless if they produce probability for that clique since benefit can be gained in both cases. Of course, each variable is only assigned to and gives probability to exactly one clique, for inference to be correct. The assignment then of variables to cliques is a heuristic that attempts to place variables at a location where its parents (if any) have been previously assigned in the junction tree. This is to encourage the deterministic variable to be able to use the parent values as often as possible via the separator values (and thus increase the detection and non-representation of zeros).
- Packed representation of clique values. In the new inference engine, clique values are represented in *packed* form rather than expanded form. The packing is done by a procedure that pre-allocates data structures making it possible to rapidly pack and unpack clique values. The data structure was designed to perform well on modern computer architectures, as it was set up to attempt to avoid branch mis-predicts, something costly in today’s highly-pipeline micro-architectures. The value of

a random variable is then packed in to $\lceil \log_2(C) \rceil$ bits, where C is the cardinality of the variable. The packing has actually has an enormously beneficial effect on both speed and memory consumption.

Speed benefits:

- There is less data traffic to and from main memory.
- As a result, there are fewer cache misses, so the benefit has a non-linear effect.
- Packed rather than unpacked values are hashed, making hash keys significantly smaller and faster to evaluate.
- There are fewer hash collisions in the hash table since in each key, each key bit is more significant to determining the key value. In unpacked representations, many of the bits of the key will always be zero zero, which means that they will be meaningless for determining hash-table key entry.

Memory benefits:

- Packed clique values are much smaller to store than unpacked values, so they take up much less space. In fact, many large cliques might have 20 or so variables. If they are unpacked, the number of bytes per clique entry would be $20 \times 4 = 80$ bytes. With tens or hundreds of thousands of clique entries, this can add up. With packed representation, however, many of these large clique values now fit in a single a single 4-byte machine word! This can decrease the memory requirements 20-fold.
- When clique values are in fact larger than one machine pointer (32bits on a 32-bit machine, 64-bits on a 64-bit machine, etc.), the packed clique values are stored in a clique-specific shared table that is also hashed for easy access. Therefore, if two cliques having the same set of variables shifted in time have the same set of values, those values will be stored only once in memory rather than multiple times for each clique instance. Because of packing, the hash table requires significantly less storage, and key computation is significantly faster.
- Many specific data-structures have been developed that are specific for GMTK and probabilistic inference, rather than using generic C++-STL data structures for everything. This includes data structures to better manage memory, managing separator-driven sparse-joins, custom fast hash tables, and so on.
- The old GMTK decision trees (DTs) were somewhat unwieldy to write and maintain. In order to help the user more easily specify such decision trees, GMTK now supports an *arbitrary* integer formula in leaf nodes of DTs. In addition to the standard arithmetic, logical, and bitwise functions, a number of common functions are now available including: max, min, median, mod, ceil, floor, round, C-style conditional evaluation, parent/child cardinality, frame number, and total number of frames in utterance. These new decision tree leaf node formulas significantly reduce the size of decision trees, and (as a result) can significantly improve their speed.

19.1 Some GMTK error messages

19.1.0.1 Warnings about not enough accumulated probability

19.1.0.2 Zero clique errors

GMTK's inference engine never stores a clique table corresponding to zero probability (it does implicit zero compression).

This is really a zero clique table error message. It is generally a user error, meaning the model is set up such that no entries of the clique

19.1.0.3 Can not successfully compute either a left or a right interface.

19.2 GMTK System Issues

Last year, it was reported that significant work had gone into a re-implementation of the core GMTK probabilistic inference engine. Since that time, we have achieved, in the general case, another factor of 2 to 3 speedup (and more in many cases) by further improving both data structures and algorithms, but also introducing still better triangulation heuristics. There have been both quite practical improvements, but also theoretical (the later of which in one case yielded a real-time wall-clock speedup of 50 on one large graph). In addition to this, a number of user visible enhancements have been added to GMTK, many of them quite useful.

These are outlines in the following list:

- **Systems/Algorithmic:** During triangulation, when converting from a directed to an undirected graph, GMTK now disconnects observed parents from their children in the corresponding undirected graph. The reason for this is that if the following structure exists $A \rightarrow B \rightarrow C$ where B is observed, then the directed model states that $A \perp\!\!\!\perp C | B$, a factorization that ideally should also be represented in the undirected model. By disconnecting B from C in the undirected model, this factorization is preserved. This can lead to significant speedups.
- In last year’s GMTK, if a variable being iterated in a clique also lived in one of its incoming separators, and that variable was assigned in the clique (meaning it existed with its parents), that variable was still considered and skipped at each iteration. Now, all such variables are removed prior to any iteration thus improving performance.
- GMTK now supports arbitrary static Bayesian networks, even if they are not Dynamic Bayesian networks. The reason for this is that since GMTK has such a powerful triangulation engine, it should be useful for use on non-dynamic problems. In addition, GMTK now also supports arbitrary disconnected networks, so one can use and compare a series of, say, naive Bayes classifiers (or even an i.i.d. set of variables) with models that do have a temporal dependence.
- The entire random variable hierarchy (RV) in GMTK was re-written so that each RV has its own C++ class, each class with the code specific for the type of RV (hidden, observed, discrete, continuous, switching, non-switching, etc.). This specialization of RVs has cut down on dynamic run-time checks that before needed to be performed, thus improving performance.
- New immediate constants for RV values, including `frameNum` (the current frame number), `numFrames` (the number of frames in the current segment), `segmentNum` (the current segment number out of all the segments currently being processed), and `numSegments`, the total number of segments currently being processed. Having these constants yields both speedups (for those graphs that utilize this information) since they do not need to be explicitly computed for each frame/segment.
- A new concept of “switching weights” have been added, a construct that generalizes the notion of word insertion penalties and language model scale factors in speech recognition. A “weight” in GMTK is a modification of a probability p according to the following formula:

$$p' = \text{penalty} * p^{\text{scale}} + \text{shift}$$

A “switching weight” is a weight that is applied depending on the value of some other random variable in a graph. For example, we might introduce a word insertion penalty of 100 by only applying a weight when a word transition occurs. The syntax for this change is as follows:

```

variable : word {
    type: discrete hidden cardinality VOCAB_SIZE ;
    switchingparents: wordTransition(-1)
        using mapping("directMappingWithOneParent");
    conditionalparents:
        word(-1) using DeterministicCPT("copyCPT")
        | word(-1) using DenseCPT("wordBigram");
    weight:
        nil
        | penalty -100 ;
}

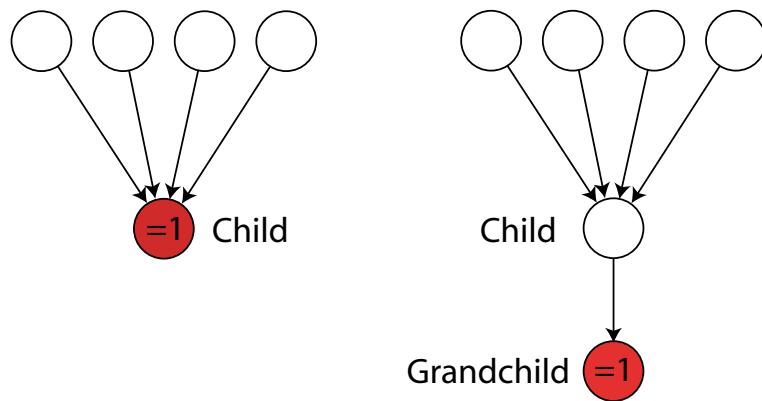
```

When the variable `wordTransition` is 0, the weight is `nil` (no weight) and when it is 1, the weight is -100 . GMTK therefore generalizes this notion to arbitrary graphical models.

- GMTK now supports “virtual evidence CPTs” (VECPTs), which are time-inhomogeneous conditional probability tables, namely ones that change depending on the time frame. One can use a VECPT to implement an child variable observed to be 1, and give a different value for each of the parents. Therefore, GMTK can easily represent standard hybrid artificial neural network (ANN)/HMM systems, but more generally any arbitrary hybrid ANN/DBN or SVM/DBN system. This gives GMTK an enormous amount of flexibility in this way.
- Many of GMTK’s error messages have been vastly improved, and are much more informative. All messages regarding files now include both file name and line number (unlike in previous versions). This significantly cuts down on user debugging time. Moreover, many enhancements have been made on GMTK’s verbose output, that allows the user to trace what the inference engine is really doing. It also includes messages that tell the user as soon as decoding hits zero probability where precisely it occurs (rather than waiting until the end of the segment).
- The main programming for timing GMTK inference `gmtkTime` has been improved where the GMTK process is forked off from the main process. This makes it possible to reduce the chance of thrashing and crashing when timing bad triangulations during an anytime triangulation search (see last year’s status report for more details).
- GMTK now supports in native mode both ARPA N-gram file formats (and so therein any arbitrary backoff procedure) and also supports factored language models (FLM) and generalized parallel back-off (GPB) procedures, using the exact same file formats that were added to SRILM. This means that it is now possible to build first-pass native mode FLM/GPB-based decoders, and to use these techniques to build sparse smoothed CPT on a variety of different types of data. In fact, we have started working on both dialog-act and part-of-speech tagging using these features, and have obtained impressive results (obtaining better performance than CRFs on the same data set, even though the DBNs have only been trained generatively). These results were submitted as two papers to ACL’05.
- **Theoretical/Practical:** In the last two years, it was described how finding the best boundary in a DBN triangulation was crucial to the performance of the resulting DBN inference. Our UAI publication from 2003 [34] includes a general exponential algorithm for finding the optimal boundary regardless of the boundary heuristic used. This year, we realized that for some (but not all) boundary heuristics, it is possible to solve them optimally using a max-flow algorithm (which therefore can run optimally, even for large M values). This algorithm, when combined with the new feature of disconnecting observed parents and with $M = 2$ yielded an additional factor of 50 speedup on one complicated

articulatory graph. A factor of 50 is quite significant!! We are now extending the boundary heuristic to submodular flow algorithms to be able to find the optimal boundary for a more general set of heuristics.

- GMTK now supports arbitrary higher-order Markov chains, not just first-order like before. This means that variables can have parents any number of frames into both the past and future. This makes it easy to test out a number of higher-order language models without needing to worry about any tricks to convert from a higher to a lower-order Markov chain.
- GMTK also now supports both forward and/or backward time edges. This allows still further novel models, such as certain coarticulatory or reverse-time effects.
- GMTK’s observation file handling has been completely redone. The observation file code now supports online up/down sampling; merging of multiple files; repetition of segments, sub-ranges, and frames; stretching of single observations to the entire duration of other files; and arbitrary subset selection. In addition, GMTK now supports online MVA processing [84, 83, 81].
- A newly written program has been created `gmtkviz`, which is a structure file visualizer for GMTK. `gmtkviz` is still in its early stages, but it is already quite useful. For example, one can display structure files, move nodes around so that they display better, add/subtract anchor points to edges (which are rendered as splines), print out postscript (for inclusion into latex, etc.), and save variable location information so that next time one opens a structure file, the variables will be in about the same place. This is all done so that nodes maintain their connectedness as appropriate.
- GMTK now supports clique printing, meaning it can print out clique contents after the collect or after both the collect/distribute evidence stage has completed. In the HMM world, this would mean that the software can print out the gamma $\gamma_t(i)$ probabilities (which is defined as the posterior probability of the state given the observations, or $\gamma_t(i) = p(Q_t = i | X_{1:T})$). GMTK can now do this for any arbitrary subset of variables, generalizing the gamma probabilities. It can also do the same during the collect evidence (forward) computation as well.
- GMTK’s decision trees (DTs) have been completely re-written so that they are both much faster and require much less memory. Basically, DTs now use a data structure that is much more packed, and each item being packed is as compressed as possible. The latest version also supports a new syntax for DTs which is much faster (especially to read for large vocabulary DTs). There are new internal data structures that in some cases make a DT access as cheap as a memory lookup.
- GMTK now supports ”Virtual Evidence Separators”. Basically, consider when GMTK finds a case of an induced-graph according to the following figure:



On the left, we have a child observed to always equal 1, and that child is deterministically related to parents. On the right, we have the grandchild observed at 1, and is randomly related to a child (the parent of grandchild), and if the child is deterministically related to parents (and otherwise the same conditions as the left hold). In both of these cases, we have that $C = f(P_{1:N})$ (i.e., the child is a deterministic function of the parents), and $G = \text{random}(C)$. What this really means is that there are a set of constraints imposed jointly on the values of the parents, and rather than iterate them all separately and independently from each other, GMTK can optionally use VE separator to deal with these implicit constraints. To do this, GMTK offline iterates through all parent values and builds a table of which parents satisfy the child (or grand child) with non-zero probability. Then when iterating through the variables during real inference (online), it iterates through all satisfying parents simultaneously (since they are dependent conditioned on the child (or grandchild)) rather than separately like in the old version. This set of parent values is either computed anew once each time GMTK is run, or is additionally optionally stored on disk so that next time GMTK runs it will load rather than re-generate the tables. The reason for the disk file is that when there are many parent values that need to be checked, computing this table can take a long time. This can have a big improvement in performance for certain graphs.

- **Heuristic:** GMTK supports three novel beam pruning options (corresponding to a forms of approximate inference). In the first form, once a clique is created, only the top k clique entries are retained, where k is a parameter (or it can be the top p percent of entries as well). This k parameter puts an upper bound on the amount of computation that will survive from one clique to the next. In the previous version, it was only possible to put a “beam” which is a ratio of log probabilities from the maximum clique value to the smallest that was retained. For very high-entropy cliques, it was difficult to set the beam to something that would allow any appreciable amount of pruning to occur.

Another new pruning option prunes cliques while they are being created. Basically, it does this by, from the two instances of a given clique at the two most recent time chunks, it estimates what the maximum value of the current clique will be. The estimates are obtained using a simple (but quite effective) 1-step linear extrapolation formula. If at any time during the creation of the clique, the partial clique probability falls below this estimate minus another beam, then that partial clique entry is not further expanded. This option is useful for early pruning based on Gaussian probabilities.

A third new beam pruning option occurs during EM training, when low-probability clique entries are not used for accumulating statistics (this trimming outliers).

- **Systems:** When variables are iterated within a clique (after all separator values have been set) there is a choice of order of the variables themselves. Recently, the ordering itself has been improved using a heuristic that works better in many cases. The ordering is of course topological within a clique, but within that constraint, it chooses variables first in decreasing priority according to: continuous, observed, and increasing real cardinality given the parents, and any ties are broken based on how many additional variables downstream are potentially influenced by knowing the value of the current variable. For example, if there are two variables a and b which have the same cardinality, but a is the parent of 2 deterministic children but b is the parent of only one deterministic child, then a is chosen next. This has also led to considerable additional speedups.
- **Systems/Theoretical:** Almost optimally packed clique representations. In the earlier version, there was a strong possibility that certain variables when packed in a clique would span machine word boundaries, thereby taking longer to pack and unpack. It turns out that the optimal way to pack a number of variables to minimize word-boundary crossings is an NP-hard problem. GMTK now implements a heuristic (inspired by approximate solutions to the *stock-cutting* problem) that in most

cases results in a packing that has zero word-boundary crossings. This has resulted in an additional appreciable speedup for all triangulations.

A number of novel features have been added to GMTK this year. These include the following: 1) new graph-based lattices with a publication submitted; 2) less memory needed for static graphs; 3) speed improvements by improving on GMTK’s hash tables; 4) new percentage-based beam options; 5) less memory needed since deterministic children need not be stored in a clique; 6) sample-based pruning; 7) undirected edges are now supported; 8) clique entropy can now be computed, useful for confidence measures; and 9) `gmtkKernel`. Each of these is now described a bit more fully.

- **Systems/Algorithmic:**

We have introduced new methodology to express word-lattices within a dynamic graphical model. There are in fact a variety of choices for doing this, including a technique to relax the time information associated with lattice nodes in a way that trades off hypothesis expansion with presumed segmentation boundary accuracy. Our approach uses a set of time-inhomogeneous and algorithmically expressed conditional probability tables to encode the lattice. The approach was implemented as part of GMTK, and word error rate improvements on the Switchboard corpus indicate that our technique is a viable means to incorporate large state space speech recognition systems into a graphical model. Moreover, we were able to decode lattices that standard hand-coded C++ implementations were unable to decode (e.g., NIST’s `latticedecode` tool) since the NIST tool ran out of memory.

Figure 19.1: A graphical model representation of a lattice

In the simplest case, a lattice is merely a representation of a stochastic sequencer, where words may follow other words regardless of what time it is, as long as they are within one of the word sequences represented by the lattice. While a lattice node n would ordinarily possess specific time point $\tau(n)$, in this view we ignore this information and only utilize the fact that a node is a branching point in the lattice where a set of new words may stochastically follow.

To represent this in a graphical model, we introduce two random variables (vertices) at each time frame, a vertex named “lattice node” and a vertex named “lattice link.” Let N_t be the lattice node vertex and L_t be the lattice link vertex at time frame t . $N_t = i$ indicates that at time t the lattice is currently at node i . Clearly, the two successive values $N_{t-1} = i, N_t = j$ determine a particular lattice link $L_t = \ell$ only if $(i, j) \in \mathcal{L}$ is link value ℓ . The meaning of a time “frame” depends on the current use of the lattice. For example, when used as a representation of a constrained search space in a speech recognition system, a time frame takes on its normal meaning (e.g., 10ms acoustic frame steps). When the lattice is used for word re-scoring, however, the time frames might correspond to words (more on this below). In either case, we introduce a third binary random variable at each time point $T_t \in \{0, 1\}$ that indicates node transition. In other words, if $T_t = 1$ then the hypothesis is that at time t , we should move from the current lattice node to one of its next valid nodes.

There are still, however, distinct ways of representing with these random variables the lattice node transition information. In one case, the target (or destination) node is made available right at transition time (i.e., when $T_t = 1$). Since both the source and target are known at this point, the target node along with the newly determined link is selected and carried forward in time throughout the duration of this new link. We call this the *known-target* approach; it is depicted in Figure 19.1 (we use the standard GMTK-style DBN description as described in [34]). If there is a transition, $T_t = 1$, then N_t is selected randomly based on a set of valid follow-on nodes starting from the node indicated by the value of N_{t-1} . Also, the value of L_t , which indicates a link, is determined based on N_{t-1} and N_t — thus, the new node is available right at transition time. If $T_t = 0$ then there is no transition, so we

just copy previous values, i.e., $L_t = L_{t-1}$ and $N_t = N_{t-1}$ with probability one, carrying this link information along throughout the duration of the link. At the beginning of the graph, there are several extra beginning-of-sentence nodes. The *first node* vertex in the figure is always equal to the identity of the starting node in the lattice. The *transition* vertex is observed to be one so that N_t is a random variable with values corresponding the second possible nodes in the lattice.

In a second case, it may be that the target node is revealed only at the *next* transition time, so the current link is not known until then. We have not implemented this approach, but it may be that it could be the more interesting of the two.

The work above has lead to a paper submission to the IEEE “spoken language technologies” conference, 2006.

- When running probability of evidence $p(x_E)$ on static graphs, it was possible to delete memory from cliques even if in the same partition when performing collect evidence. This now occurs in GMTK.
- GMTK’s hash tables have been completely re-written, and are now much faster (this alone gives approximately a 20-50% speedup). Customized hash functions were developed that were optimized for the type of hash keys that GMTK needed (i.e., packed clique values).
- A new beam-pruning option was developed (based on a suggestion by Andrew McCallum). Essentially, this option prunes away all clique entries below a threshold that is set such that all of the entries that are kept make up x -percent of the total clique mass. So far, however, after experimenting extensively with this option, it has not outperformed the k -best pruning that was described last year.
- A significant memory savings can occur for graphs with much determinism since deterministic children need not be stored in a clique, rather they can be constructed at a later time when needed (since they are deterministic function of their parents). This occurs, moreover, without a significant speed penalty (since there is only a constant speed hit at the time that we re-expand the clique entry). In addition, this approach is applied recursively, so that if there are children of deterministic children, they are also not saved. Last year, the clique packing algorithm was described, and that it occurred at 32-bit boundaries. Here, deterministic children are added back into the clique storage if it does not go over a 32-bit boundary (since that does not increase memory and only can increase speed).
- Another sample-based pruning option was been developed, where after pruning, a Monte Carlo method is used to sample from the portion of the clique that has been pruned away. This is being investigated further.
- Undirected edges are now supported in GMTK. Currently, undirected edges only effect the triangulation results, but in the long run, we will use them for producing hybrid Bayesian networks and undirected graphical models. The undirected models can the be used for expressing exponential distributions, conditional random fields, and regular constraints.
- The entropy of a clique can now be computed. This has been useful for confidence measures during the development of the work that lead to several of our recent papers [412, 414].
- `gmtkKernel`. As part of our bioinformatics needs, a new program was written called `gmtkKernel`. This program produces what we call the “accumulator kernel”, whereby a mapping from a variable length string (the DBN observations) to a fixed length string, of the length of the number of parameters, is done by computing the accumulators during an EM pass, but only during that one string. This fixed-length string can thus be thought of as a mapping from input to feature space in an SVM/kernel machine. We are currently investigating both theoretical and practical aspects associated with this kernel.

Part VII

Appendices

Appendix A

History and Development Milestones of GMTK

Below is the history of the GMTK project.

1990s: In the late 1990s, in some sense GMTK started out as two Ph.D. theses. The first thesis [48] (by Jeff Bilmes) introduced buried Markov models, or BMMs, which essentially are temporal Markov-chain controlled switching sparse conditional Gaussian mixture models for speech recognition, and also an associated set of algorithms to induce sparse and discriminative models, long before sparsity in machine learning became popular. In this work, the pattern of preferred sparsity was imposed a priori, before coefficient learning took place, and a number of discriminative heuristics were introduced to induce a pattern of sparsity that would be useful for discriminative tasks (pattern recognition). The goal of the sparse pattern was to produce dependencies that were inherently discriminative, hopefully making a discriminative loss function less important. This is in contrast to the way that sparsity is learned these days, where a loss function is decided upon (e.g., which can be either generative or discriminative, and can have a variety of forms), and a regularizer (such as ℓ_1 or a sparse group norm) is used to encourage sparsity while at the same time satisfying a loss function.

The second thesis [462] (by Geoff Zweig) was about building novel structures using dynamic Bayesian networks that were useful for speech recognition. This includes the classic triangle structure (that is mentioned below, see §??) and which allows one to easily specify a probabilistic sequencer directly using the facilities of a DBN, and where a standard inference algorithm can be used to implement the sequencer. One of the hypotheses proposed in this thesis was that one can build a variety of sequential structures right in the DBN by explicitly representing them in the graph, and then use a standard inference algorithm, without needing to resort to new C coding for each new feature. Geoff's master's thesis, on a similar topic, also proposed the frontier algorithm which was a novel way to produce a triangulation of a DBN directly based on the directed model, without having to first moralize it into an undirected graphical model or Markov random field.

2001: In 2001, Jeff and Geoff proposed a workshop as part of the Johns Hopkins University (JHU) summer workshop series. This workshop was accepted and in preparation for this workshop, Jeff and Geoff put the ideas from their two theses together to produce the first version of GMTK. Geoff was a research scientist at IBM at the time while Jeff was an assistant professor at the University of Washington, Seattle. GMTK was originally developed for speech recognition, and the results of the JHU workshop, other than the toolkit itself, were performance numbers showing that sparse discriminative models could help for some speech tasks. Other results showed that GMTK was a useful tool for articulatory modeling in speech recognition.

Another result of this was the further exposition of the idea of explicitly representing features of an ASR system right in the graph. As a concrete example, there are many speech recognition systems that do not allow for optional silence to be hypothesized during parameter training, meaning that either each word must follow another word, or all words must have silence interspersed between them. In any given speech utterance, it is usually only the word string that is transcribed as training data, not the location of those words in the acoustic utterance nor

The Aurora 2.0 training model that was developed as part of this workshop (and is used as one of the GMTK tutorials), however, allowed for optional silence to be hypothesized between each word, and this was done entirely via the DBN structure and the DBN framework without needing to modify any of the underlying inference algorithms or C++ code (unlike what would be required with a standard ASR system not having this facility). It was (and still is) felt that this general flexibility would be quite (and in fact is) useful for developing and quickly testing out novel ASR concepts.

- 2002: In 2002, the first version of the GMTK documentation was written. The documentation was written by Jeff, except at the beginning which included a copy of the 4-page ICASSP2002 paper [53] written by both Jeff and Geoff. Before this documentation existed, there were many questions that were repeatedly being answered by email, but after the documentation was complete, the number significantly died down (unsurprisingly). Also, at that time there, were a number of discussions about releasing the GMTK code open source but it was quite tricky due to various intricate and difficult legal and licensing issues associated with open source code.
- 2003: The 2001 version of GMTK used only one type of triangulation, and it did so implicitly via the use of the frontier algorithm (the actual junction tree produced in this early GMTK case was *always* a junction chain rather than a tree).

There are of course many other triangulation heuristics, so code for a new and completely separate GMTK triangulation engine was built for this purpose `gmtkTriangulate` by Jeff and Chris. This “engine” has been extended many times since 2003 (the last time in 2011), and it supports a wide variety of both standard and novel triangulation heuristics, including non-minimal [19] triangulation heuristics (useful when the model contains a lot of sparsity) and also an optimal exhaustive search triangulation procedure for those cases where finding the optimal triangulation is crucial (while this of course takes time exponential in the number of nodes, it can be useful if the resultant inference algorithm is used many times). At that point in time, however, GMTK was not able to take advantage of the triangulations that GMTK’s new triangulation engine produced since the inference code used only the junction chain produced by the frontier algorithm

- 2004: The main project in 2004 was a complete rewrite of all of the high- and low-level probabilistic inference code – this included a completely new sparse-join based inference methodology based on a given triangulation (produced by the aforementioned triangulation engine), and based partition based triangulation as reported in the UAI 2003 [51] paper by Bilmes and Bartels. There were many new ideas here, including a novel way of doing inference based on sparse joins, and which borrowed methods from the SAT and CSP communities. While it might not seem like SAT/CSP has a lot to do with DGMs, in fact it does, in particular when considering that a DGM can be triangulated via first finding a boundary, the `-M` and the `-S` option of `gmtkTriangulate` (which allows for large sparse cliques), the triangulation of the resulting chunks, and then the SAT/CSP methods can be used to expand the resultant cliques in each of the chunks. Of course since the chunks are repeated, and from information gleaned by expanding previous chunks could be used to facilitate expanding later chunks, there are quite a few avenues for extending beyond what is possible to do in standard SAT/CSP problems. Much of this was exploited in this new version. We note that some of these ideas are briefly

mentioned in [50] but as of yet, most of the ideas behind GMTK’s current inference algorithm are currently not yet published. Later sections of this documentation (see Section ??, however, include a description of how inference is performed in GMTK.

At this point, all of the code in GMTK was produced by the University of Washington (i.e., by Bilmes or his students) and all the first version old GMTK code was no longer a part of GMTK.

A few other things happened in 2004 including: 1) seekable and iterable decision trees (to allow deterministic mapping functions to change at each segment); 2) n -gram CPTs (to allow ARPA language models to be read in directly to a CPT), and 3) factored language models (again to allow them to be read into a CPT). At this point, GMTK became a useful tool for simple natural language processing (NLP) tasks such as tagging and chunking in addition to speech recognition.

- 2005: In 2005, a number of natural language processing features were added. This included the ability to read in HTK [453] lattices files directly. A lattice (in the context of speech recognition and not to be confused with a Birkhoff lattice in mathematics) is a way of representing an exponential (in the string length) number of string hypotheses in a sequential model using a data structure (the lattice) that takes only a linear amount of space. A lattice also (via dynamic programming algorithms) allows an algorithm to score all exponential number of string hypotheses without requiring exponential computation (in fact the computation is still linear in T). Lattices are widely used in speech recognition as a compressed and efficient representation of an N -best list output of an earlier stage speech recognizer. The lattice can then be “rescored” by a more powerful but more computationally complex model, but done so in a way that only the hypotheses represented in the lattice are considered in order to moderate the computational costs. With HTK lattice support in GMTK, it is possible to read a lattice directly into a GMTK CPT and start using it immediately, without needing any additional scripting or converting into a sparse CPT. It is also possible to read in one lattice for each segment (i.e., each observation sequence can be associated with its own set of lattices). This latter facility is useful, say in speech recognition, when one wants to rescore many lattices each corresponding to its own acoustic observation.

The other changes in 2005 included: 1) the ability to include Dirichlet and Laplace priors on dense and sparse CPTs during EM training; 2) the maxflow boundary finding algorithm, which finds a boundary separator between repeated periodic chunks in a DGM using a modular cost function (this is done by reducing the inherent vertex cut problem into an edge cut problem which is then solved by maxflow); 3) optimized hash tables for many of the internal data structures (including clique and separator indexing, symbol tables, and so on). Also added were command line controls over hash-table load factors which allow for user-control over some of the memory/speed tradeoffs associated with hash tables; 4) command line debugging output using the `-verb` command, so that one can get debugging and trace outputs for things like the inference, reading in parameters, and so on. This makes it easier to, for example, debug model errors (like zero clique errors); 6) Lastly, this year was the first year that the first version of the GMTK graph visualization tool `gmtkviz` went live.

- 2006: The main feature added in 2006 was the use of centered and non-centered regularizers during EM training of Gaussian parameters. This also allows for a form of adaptation (regularized adaptation), where one can train but regularize towards trained parameters from a previous system. The other accomplishment this year was to remove some textbook inference code that was added in 2004. This code exactly implemented the equations that are commonly used to describe graphical model inference, and were added for debugging and pedagogical purposes. At the time it was thought that the code was no longer useful, it being so slow, but it is now (in 2011) thought that this code should be added back in (due to its pedagogical utility).

Another important feature added in 2006 was the first version of the `gmtkTie` program, written by Simon King from Edinburgh University. This program allows one to cluster Gaussians together based on various heuristics and as a result merge states associated with those clusters. Many of these heuristics are common in the speech recognition community for use with HMMs, but they have not been so widely used in the context of a general DGM.

2007: It seems 2007 was a slow year, as I haven't found anything in the email logs about major GMTK events in 2007. Of course, this was the year my first son was born and my father died, within a few months of each other. In fact, I would like to take this opportunity to make a dedication both to my father Murray, and my (now two) sons Alexander and Sebastian.

2008: In 2008, the ability to produce a DBN-based Fisher Kernel was introduced. This was done by adding the program `gmtkKernel` which output the result of $f = \frac{d}{d\theta} \log p_\theta(\bar{x})$ for observed value \bar{x} and model parameters θ . This can be seen as a mapping from a variable number of random variable instantiations \bar{x} to a fixed number of parameters f , from which a kernel can be defined.

Another enhancement was the ability to better process HTK feature files (compressed files and sub-ranges). Still another GMTK accomplishment was a new symbol table facility, so that any random variable declaration could also ask to use a symbol table, essentially a mapping from $[0, c - 1]$ (where c is the cardinality of a random variable) to a set of text symbols that are used to print out random variable values rather than the integers.

2009: A big change occurred in 2009 which was a complete re-write of all the sequential data structures to support biological sequences. Biological sequences can be very long (hundreds of millions of frames) and before this modification, even the unrolling operation within GMTK (which was still using C++ STL classes) was much too slow. Before then, only speech and language sequences were used which were at most only a few tens of thousands of frames long. This new re-write made it such that graph unrolling never occurs to the length of the observations, and rather only a form of online inference is used to expand the low-level junction tree clique table data structures (that are memory optimized) would expand to the length of the sequence. Also, the island algorithm was re-written to be much cleaner, and used some new data structures to actually store the islands. These changes lead to enormous speedups on biological sequences, and moreover even achieved some more minor speedups on short sequences as well due to better hardware cache utilization.

A few other changes this year included two new pruning options, the predictive pruning with an continuation heuristic learnt online (this was described and published in Interspeech 2010 [52]), and also diversity pruning. Another new feature was the missing feature Gaussian, where a Gaussian would ignore a feature (treat it as hidden) if it had a NaN value. Also, Gaussians evaluations were further improved by vectorizing the code (the use of explicit loop unrolling, local register usage, and other PHiPAC-style tricks).

This year, also, the old-style Viterbi program front end was completely removed and redone. Printing Viterbi variables was done in a way that supported the above new data structures, and also in way that was a bit simpler. A caveat to this new printing code, however, was that it printed the modified chunks (based on the boundary finding algorithm) rather than the user-defined ones. While this normally wouldn't create a problem, it could lead to some print output that would need to be fixed by a user script (note that this was fixed in 2011, see below).

Lastly, at this point the code was so significantly different than the main CVS trunk, that this code version did not get merged into the trunk until late 2010 (so looking at the logs show that these changes occurred in 2010 even though they really occurred in the beginning of 2009).

2010: In 2010, the main achievement was moving the code to the mercurial revision controls system (which is much better than CVS which is what we were previously using). We also started using `trac`, bug tracking system which is fully integrated with mercurial, and which allows us to efficiently keep track of and fix bugs.

Also, in 2010 a new version of the documentation was released, but really this was still the 2002 documentation with only a few minor typo fixes. Thus, the 2010 documentation was still quite out of date in that it mentioned none of the new features mentioned above that were introduced since 2002.

Another big change that occurred in 2010 was the merging of three separate GMTK branches into one: 1) the main CVS trunk which had had a few minor changes and bug fixes; 2) the major new data structure modifications, Viterbi printing, and so on mentioned above in 2009; and 3) a branch that had some important changes done to the GMTK tie program and that also added observation support for compressed HTK files (and other HTK observation file compatibilities). While this merge was a bit painful and time consuming, it seems to have ended up all well and good. Note that the merging was done while the source was still in CVS. As soon as this was done, everything was then immediately converted over to mercurial.

Also, GMTK's lattice support was improved this year so that GMTK can read in a lattice where there are more than two links between two lattice nodes (before it required that only one link between nodes existed).

Lastly, significant improvements and bug-fixes occurred in `gmtkViz`, GMTK's graph visualizer tool. This also included fixes that make it quite easy to compile, and also made it up to date with respect to the `wxwidgets`, the graphics library that `gmtkViz` uses.

2011: This year, we moved entirely to a gnuconf compilation model, so compiling GMTK (which used to be a bit finicky) is now as simple as typing `configure` and then `make`.

Also, another important user-level change is a new Viterbi-printing code. The Viterbi printing is now the same regardless of the boundary used and regardless of the underlying triangulation, thereby making it much easier to parse the output of the Viterbi program (while this sounds easy, this was a non-trivial change).

This year, we discovered that GMTK's interpreted deterministic mapping facility (decision trees with integer leaf-node formulas) could be excruciatingly slow in some cases. Therefore, it is now possible to compile in C++ code to express a given deterministic mapping. This has resulted in some significant speedups (I can imagine that if this had been done years ago, a number of models would have had a significant speed boost as some of the decision trees I've seen people create were truly monstrous).

Inventory tutorial and regression testing data sets were also created this year. Many GMTK models have been developed and collected over the years, but they had not been inventoried or standardized. Much effort was spent in collecting these models together and unifying them. The purpose of these models collected in one unified collection is to a) be used as example tutorial material for users (it is a lot easier to start with a working model that is similar to what you wish to do than to start from scratch), and b) to be used as a regression test suite, so that when new versions of GMTK come out, it will be easy to test to make sure that it produces identical results to earlier versions. Some of these models are now mentioned in this documentation.

At this time, many GMTK models have been inventoried, categorized, tested, and normalized. A spreadsheet has been created that lists all of the models and their features, allowing the user to quickly find a model that is most relevant to them. We are continuing to expand this set by unifying disparate models submitted by GMTK users.

In 2011 this documentation that you are reading now was first drafted (might we say completed?? Probably not). While there are still many unfinished sections, it should still be partially useful to some.

2012: Need to add details here.

2013: Need to add details here.

2014: Need to add details here.

One other thing that happened this year is a source code release on the web page <http://melodi.ee.washington.edu/gmtk>.

2018: We had a major server crash and unfortunately lost our bug tracker information. We should have moved to github much earlier.

2020: We moved all the source code over to git (it was in mercurial), and also started using github (as we should have many years ago). We also made the source publicly available on github.

I should also mention that in each of the above years many many bugs were identified and fixed, and many speedups were implemented (some quite low level such as loop unrolling and software pipelining of the low-level Gaussian routines, approximately optimal packed clique table implementations, and so on). I can't thank enough the people who have used GMTK and reported bugs (thank you!). Due to the bug tracker introduced in 2010, remembering and documenting such bugs became much cleaner and more efficient (but then again, some of these bugs were lost in 2018 due to a major server crash).

Appendix B

GMTKL Grammar

The following is an annotated context-free grammar that describes the GMTKL syntax in some detail. In the example, lines that start with the % character are comments.

```
GM = "GRAPHICAL_MODEL" identifier FrameList ChunkSpecifier

% A list of frames.
FrameList = Frame FrameList | NULL

% A frame consists of a list of random variables.
Frame = "frame" ":" integer "{" RandomVariableList "}"

RandomVariableList = RandomVariable RandomVariableList | NULL

% A random variable starts with its name such as "variable : foo"
% followed by a list of random variable attributes.
RandomVariable = "variable" ":" name "{" RandomVariableAttributeList "}"
    | "factor" ":" name "{" FactorAttributeList "}"

RandomVariableAttributeList =
    RandomVariableAttribute RandomVariableAttributeList | NULL

% Random variable attributes can either specify the random
% variable type, or might specify parents and parameters.
RandomVariableAttribute = TypeAttribute |
    EliminationHintAttribute |
    WeightAttribute |
    ParentsAttribute

% If random variable may have a weight.
WeightAttribute = "weight" : WeightAttributeSpecList ";""

WeightAttributeSpecList =
    WeightAttributeSpec " | " WeightAttributeSpecList
```

```

| WeightAttributeSpec

WeightAttributeSpec =  WeightOptionList
| "nil"

% there are many types of weights.
WeightOptionList =
( WeightType WeightOption ) WeightOptionList
| ( WeightType WeightOption )

WeightType = "scale" | "penalty" | "shift"

% A weight can be a constant "value" for all instances
% of this random variable, or it can come from
% one of the observation matrix elements itself, giving
% a different weight for this random variable at each time frame.
% In the later case, the integer range must have the form
%      m:m
% meaning it specifies a single scalar value, and the
% index m must correspond to a floating point feature.
WeightOption = ( "value" number )
| ( integer ":" integer )

EliminationHintAttribute = "elimination_hint" : number ;"

TypeAttribute = "type" ":" RandomVariableType ;"

% A parent can be either switching or conditional. A conditional
% parent is identical to the normal notion of a parent
% in a Bayesian network.
ParentsAttribute =
( "switchingparents" ":" SwitchingParentAttribute ";" )
| ( "conditionalparents" ":" ConditionalParentSpecList ";" )
# Semantics requires that we have both
# switchingparents & conditionalparents in a RV. Note that
# the parse grammar allows this not to be the case.

% A random variable is either of discrete or continuous type.
RandomVariableType = RandomVariableDiscreteType |
                    RandomVariableContinuousType

% A discrete random variable is either
% 1) hidden
% 2) observed, in which case you must specify either

```

```

%      a) a observation range of size 1 corresponding
%          to the observed values of this variable. The
%          range takes the form n:n, where n is the index of the
%          element in the observation file to obtain the value
%          of this variable for each frame.
%
%      or
%
%      b) a fixed immediate and inline value of this observed
%          random variable.
%
% In either case, you also provide the RV cardinality.
RandomVariableDiscreteType =
    "discrete"
    ( "hidden" | "observed" (integer ":" integer | "value" integer) )
    "cardinality" integer

%
% A continuous random variable is either
% 1) hidden (not currently supported in GMTK)
% 2) observed, in which case you must specify
% an observation range. The observation range takes
% the form of n:m and determines the
% size of the continuous observation vector.
RandomVariableContinuousType =
    "continuous"
    ("hidden" | "observed" integer ":" integer)

%
% A variable may have a list of switching parents, or
% if no switching parents are desired, use just 'nil'
SwitchingParentAttribute = "nil" | ParentList "using" MappingSpec

%
% For each vector value in the joint state space of the set of
% switching variable values, a decision tree maps that vector
% value down to an integer. For each possible integer value,
% there is a collection of conditional parents.
ConditionalParentSpecList =
    ConditionalParentSpec "|" ConditionalParentSpecList
    | ConditionalParentSpec

%
% A set of conditional parents consists of a list
% of parent specifiers, and an implementation.
ConditionalParentSpec = ConditionalParentList using Implementation

ConditionalParentList = "nil" | ParentList

```

```

ParentList =
    Parent "," ParentList
    | Parent

% A parent consists of a variable identifier, and an integer
% frame offset.
Parent = identifier "(" integer ")"

Implementation = DiscreteImplementation | ContinuousImplementation

% A discrete implementation can be one of the below.
DiscreteImplementation = ( "DenseCPT" | "SparseCPT" | "DeterministicCPT"
                           "NGramCPT" | "FNGramCPT" )   "(" ListIndex ")"

ContinuousImplementation = ContObsDistType
(
    "(" ListIndex ")"
    |
    "collection" "(" ListIndex ")" MappingSpec
)
#
# A ContinuousImplementation has two cases:
# 1) the first case uses the syntax
#
#         "(" ListIndex ")"
#
# and is used if conditional parents
# are nil in which case we select only one dist. In
# this case, 'ListIndex' refers directly to
# a particular single gaussian mixture, using
# the normal 'ListIndex' format (either an int
# or a string name). Note that in this case,
# the number of conditional parents (as given
# in the 'conditionalParents' array, will be zero.
# This is analogous to the discrete case where you directly
# select a CPT with the appropriate parents
#
# 2) in the second case, which uses the syntax
#
#         "collection" "(" ListIndex ")" MappingSpec
#
# this is when we have multiple
# conditional parents, and we need another decision tree to map
# from the conditional parents values to the appropriate
# distribution. Therefore we use the mapping syntax.
# In this case, the 'MappingSpec' must refer to one of

```

```
# the decision trees. The collection refers to the
# collection of objects in global arrays, and the mappingspec
# gives the mapping from random variable parent indices to
# the relative offset in the collection array.
#
%
% At the moment, the current version of GMTK only supports
% a mixGaussian distribution (the others are not yet supported),
% but the underlying mixture components can be a variety of different
% types of distribution.
ContObsDistType = "mixture" | "gausSwitchMixture"
| "logitSwitchMixture" | "mlpSwitchMixture"

MappingSpec = "mapping" "(" ListIndex ")"
    # A MappingSpec always indexes into one of the decision trees.
    # The integer (or string) is used to index into a table
    # of decision trees to choose the decision tree
    # that will map from the switching parents to one of the
    # conditional parent lists.

% A chunk specifier is the range of frames (inclusive) corresponding to the chunk.
% Before the chunk is the prologue and after is the epilogue.
ChunkSpecifier = "chunk" integer ":" integer

% A list index is just the name (a string) of an internal GMTK object.
ListIndex = integer | string

number = integer | floating_point_value
```


Part VIII

Bibliography

Bibliography

- [1] S. M. Aji and R. J. McEliece. The generalized distributive law. *IEEE Transactions in Information Theory*, 46:325–343, March 2000.
- [2] S.B. Akers. Binary decision diagrams. *IEEE Transactions on computers*, 27(6):509–516, 1978.
- [3] R.G. Almond. *Graphical Belief Modeling*. Chapman & Hall, 1995.
- [4] M Schmidt and A Niculescu-Mizil and K Murphy. Learning graphical model structure using l1-regularization paths. In *Proceedings of The Twenty Second Conference on Artificial Intelligence, AAAI 2007*, July 2007. L1 regularization for structure learning.
- [5] H. Attias, L. Deng, A. Acero, and J.C. Platt. A new method for speech denoising and robust speech recognition using probabilistic models for clean speech and for noise. In *European Conf. on Speech Communication and Technology (Eurospeech)*, 7th, 2001.
- [6] Zafer Aydin, Ajit Singh, Jeff Bilmes, and William Noble. Learning sparse models for a dynamic bayesian network classifier of protein secondary structure. *BMC Bioinformatics*, 12(1):154, 2011.
- [7] F. Bacchus, S. Dalmao, and T. Pitassi. Algorithms and complexity results for #sat and bayesian inference. In *Foundations of Computer Science FOCS-2003*, pages 340–351, 2003.
- [8] F. Bacchus, S. Dalmao, and T. Pitassi. Value elimination: Bayesian inference via backtracking search. In *Uncertainty in Artificial Intelligence: Proceedings of the Nineteenth Conference (UAI-2003)*, pages 20–28. Morgan Kaufmann Publishers, 2003.
- [9] R.I. Bahar, E.A. Frohm, C.M. Gaona, G.D. Hachtel, E. Macii, A. Pardo, and F. Somenzi. Algebraic decision diagrams and their applications. *Formal methods in system design*, 10(2):171–206, 1997.
- [10] L.R. Bahl, P.F. Brown, P.V. de Souza, and R.L. Mercer. Maximum mutual information estimation of HMM parameters for speech recognition. In *Proc. IEEE Intl. Conf. on Acoustics, Speech, and Signal Processing*, pages 49–52, Tokyo, Japan, December 1986.
- [11] Gökhan H. Bakir, Thomas Hofmann, Bernhard Schölkopf, Alexander J. Smola, Ben Taskar, and S. V. N. Vishwanathan. *Predicting Structured Data (Neural Information Processing)*. The MIT Press, 2007.
- [12] R. Barra-Chicote, F. Fernández, S. Lutfi, J.M. Lucas-Cuesta, J. Macias-Guarasa, JM Montero, R. San-Segundo, and JM Pardo. Acoustic emotion recognition using dynamic bayesian networks and multi-space distributions. *Proc. Interspeech, Brighton*, pages 336–339, 2009.
- [13] C. Bartels and J. Bilmes. Elimination is not enough: Non-minimal triangulations for graphical models. Technical Report UWEETR-2004-0010, University of Washington, Dept. of Electrical Engineering, 2004.
- [14] C. Bartels and J. Bilmes. Non-minimal triangulations for mixed stochastic/deterministic graphical models. In *Uncertainty in Artificial Intelligence: Proceedings of the Twenty-first Conference (UAI-2006)*, Cambridge, MA, July 2006. AUAI.
- [15] Chris Bartels and Jeff Bilmes. Focused state transition information in ASR. In *Proc. of IEEE Automatic Speech Recognition and Understanding Workshop (ASRU)*, San Juan, Puerto Rico, November/December 2005.

BIBLIOGRAPHY

- [16] Chris Bartels and Jeff Bilmes. Use of syllable nuclei locations to improve ASR. In *Proc. of IEEE Automatic Speech Recognition and Understanding Workshop (ASRU)*, Kyoto, Japan, December 2007.
- [17] Chris Bartels and Jeff Bilmes. Using syllable nuclei locations to improve automatic speech recognition in the presence of burst noise. In *Proc. of Interspeech*, Brisbane, Australia, September 2008.
- [18] Chris Bartels and Jeff Bilmes. Graphical models for integrating syllabic information. *Computer Speech and Language*, 24(4):685–697, 2010.
- [19] Chris Bartels and Jeff A. Bilmes. Creating non-minimal triangulations for use in inference in mixed stochastic / deterministic graphical models. *Machine Learning Journal*, 84(3):249–289, 2011.
- [20] Chris Bartels, Kevin Duh, Jeff Bilmes, Katrin Kirchhoff, and Simon King. Genetic triangulation of graphical models for speech and language processing. In *Proc. of 9th European Conference on Speech Communication and Technology (Eurospeech’05)*, Lisbon, Portugal, September 2005.
- [21] L.E. Baum and T. Petrie. Statistical inference for probabilistic functions of finite state Markov chains. *Ann. Math. Statist.*, 37(6):1554—1563, 1966.
- [22] L.E. Baum, T. Petrie, G. Soules, and N. Weiss. A maximization technique occurring in the statistical analysis of probabilistic functions of Markov chains. *Ann. Math. Stat.*, 41(1):164—171, February 1970.
- [23] M.J. Beal, H. Attias, and N. Jojic. Audio-video sensor fusion with probabilistic graphical models. In *Proc. ECCV*, 2002.
- [24] Antony J. Bell and Terrence J. Sejnowski. An information maximisation approach to blind separation and blind deconvolution. *Neural Computation*, 7(6):1129–1159, 1995.
- [25] Y. Bengio. Learning deep architectures for ai. *Foundations and Trends® in Machine Learning*, 2(1):1–127, 2009.
- [26] A. L. Berger, S.A. Della Pietra, and V.J. Della Pietra. A maximum entropy approach to natural language processing. *Computational Linguistics*, 22(1):39–71, 1996.
- [27] P. Billingsley. *Probability and Measure*. Wiley, 1995.
- [28] J. Bilmes. *Natural Statistical Models for Automatic Speech Recognition*. PhD thesis, U.C. Berkeley, Dept. of EECS, CS Division, 1999.
- [29] J. Bilmes. Graphical models and automatic speech recognition. Technical Report UWEETR-2001-005, University of Washington, Dept. of EE, 2001.
- [30] J. Bilmes. What HMMs can do. Technical Report UWEETR-2002-003, University of Washington, Dept. of EE, 2002.
- [31] J. Bilmes. Buried markov models: A graphical modeling approach to automatic speech recognition. *Computer Speech and Language*, 17:213—231, April—July 2003.
- [32] J. Bilmes. On soft evidence in bayesian networks. Technical Report UWEETR-2004-0016, University of Washington, Dept. of EE, 2004.
- [33] J. Bilmes, K. Asanović, C.W. Chin, and J. Demmel. Optimizing matrix multiply using PHiPAC: a portable, high-performance, ANSI C coding methodology. In *Proceedings of the International Conference on Supercomputing*, Vienna, Austria, July 1997. ACM SIGARCH.

- [34] J. Bilmes and C. Bartels. On triangulating dynamic graphical models. In *Uncertainty in Artificial Intelligence: Proceedings of the Nineteenth Conference (UAI-2003)*, pages 47–56. Morgan Kaufmann Publishers, 2003.
- [35] J. Bilmes and C. Bartels. On triangulating dynamic graphical models. In *Uncertainty in Artificial Intelligence (UAI)*, pages 47–56, Acapulco, Mexico, 2003. Morgan Kaufmann Publishers.
- [36] J. Bilmes and K. Kirchhoff. Factored language models and generalized parallel backoff. In *Human Lang. Tech., North American Chapter of Assoc. Comp. Ling.*, Edmonton, Alberta, May/June 2003.
- [37] J. Bilmes, N. Morgan, S.-L. Wu, and H. Bourlard. Stochastic perceptual speech models with durational dependence. *Intl. Conference on Spoken Language Processing*, November 1996.
- [38] J. Bilmes and G. Zweig. The Graphical Models Toolkit: An open source software system for speech and time-series processing. *Proc. IEEE Intl. Conf. on Acoustics, Speech, and Signal Processing*, 2002.
- [39] J. Bilmes, G. Zweig, T. Richardson, K. Filali, K. Livescu, P. Xu, K. Jackson, Y. Brandman, E. Sandness, E. Holtz, J. Torres, and B. Byrne. Discriminatively structured graphical models for speech recognition: JHU-WS-2001 final workshop report. Technical report, CLSP, Johns Hopkins University, Baltimore MD, 2001. <http://www.clsp.jhu.edu/ws2001/groups/gmsr/GMRO-final-rpt.pdf>.
- [40] J. Bilmes, G. Zweig, T. Richardson, K. Filali, K. Livescu, P. Xu, K. Jackson, Y. Brandman, E. Sandness, E. Holtz, J. Torres, and B. Byrne. Discriminatively structured graphical models for speech recognition: JHU-WS-2001 final workshop report. Technical report, CLSP, Johns Hopkins University, Baltimore MD, 2001. <http://www.clsp.jhu.edu/ws2001/groups/gmsr/GMRO-final-rpt.pdf>.
- [41] J. A. Bilmes. Graphical models and automatic speech recognition. In R. Rosenfeld, M. Ostendorf, S. Khudanpur, and M. Johnson, editors, *Mathematical Foundations of Speech and Language Processing*. Springer-Verlag, New York, 2003.
- [42] J.A. Bilmes. A gentle tutorial of the EM algorithm and its application to parameter estimation for Gaussian mixture and hidden Markov models. Technical Report TR-97-021, ICSI, 1997.
- [43] J.A. Bilmes. Data-driven extensions to HMM statistical dependencies. In *Proc. Int. Conf. on Spoken Language Processing*, Sidney, Australia, December 1998.
- [44] J.A. Bilmes. Buried Markov models for speech recognition. In *Proc. IEEE Intl. Conf. on Acoustics, Speech, and Signal Processing*, Phoenix, AZ, March 1999.
- [45] J.A. Bilmes. Dynamic Bayesian Multinets. In *Proceedings of the 16th conf. on Uncertainty in Artificial Intelligence*. Morgan Kaufmann, 2000.
- [46] J.A. Bilmes. Factored sparse inverse covariance matrices. In *Proc. IEEE Intl. Conf. on Acoustics, Speech, and Signal Processing*, Istanbul, Turkey, 2000.
- [47] Jeff Bilmes. A gentle tutorial of the EM algorithm and its application to parameter estimation for Gaussian mixture and hidden Markov models. Technical Report TR-97-021, ICSI, 1997.
- [48] Jeff Bilmes. *Natural Statistical Models for Automatic Speech Recognition*. PhD thesis, U.C. Berkeley, Dept. of EECS, CS Division, 1999.

- [49] Jeff Bilmes. Gaussian models in automatic speech recognition. In David Havelock, Sonoko Kuwano, and Michael Vorlander, editors, *Handbook of Signal Processing in Acoustics*, pages 521–556. Springer Science+Business Media, LLC, 2008.
- [50] Jeff Bilmes. Dynamic graphical models. *IEEE Signal Processing Magazine*, 27(6):29–42, November 2010.
- [51] Jeff Bilmes and Chris Bartels. On triangulating dynamic graphical models. Technical Report UWEETR-2002-0007, University of Washington, Department of Electrical Engineering, 2002. <https://www.ee.washington.edu/techsite/papers/refer/UWEETR-2002-0007.html>.
- [52] Jeff Bilmes and Hui Lin. Online adaptive learning for speech recognition decoding. In *Proc. Annual Conference of the International Speech Communication Association (INTERSPEECH)*, Makuhari, Japan, September 2010.
- [53] Jeff Bilmes and Geoff Zweig. The graphical models toolkit: An open source software system for speech and time-series processing. *Proc. IEEE Intl. Conf. on Acoustics, Speech, and Signal Processing*, 2002.
- [54] Jeff A. Bilmes. What hmms can do. *IEICE - Transactions on Information and Systems*, E89-D(3):869–891, March 2006.
- [55] J. Binder, K. Murphy, and S. Russell. Space-efficient inference in dynamic probabilistic networks. *Int'l. Joint Conf. on Artificial Intelligence*, 1997.
- [56] G. Birkhoff. Lattice theory, rev. ed. In *Amer. Math. Soc. Colloq. Publ*, volume 25, 1948.
- [57] C. Bishop. *Neural Networks for Pattern Recognition*. Clarendon Press, Oxford, 1995.
- [58] C. Bishop. *Pattern Recognition and Machine Learning*. Springer, 2006.
- [59] D. Blackwell and L. Koopmans. On the identifiability problem for functions of finite markov chains. *Ann. Math. Stat.*, 28(4):1011—1015, 1957.
- [60] H. Bourlard and N. Morgan. *Connectionist Speech Recognition: A Hybrid Approach*. Kluwer Academic Publishers, 1994.
- [61] X. Boyen, N. Friedman, and D. Koller. Discovering the hidden structure of complex dynamic systems. *15th Conf. on Uncertainty in Artificial Intelligence*, 1999.
- [62] X. Boyen and D. Koller. Exploiting the architecture of dynamic systems. In *AAAI/IAAI*, pages 313–320, 1999.
- [63] X. Boyen and D. Koller. Tractable inference for complex stochastic processes. In *Proc. Fourteenth Conf. on Uncertainty in Artificial Intelligence (UAI-99)*, 1999.
- [64] Rodrigo De Salvo Braz, Eyal Amir, and Dan Roth. Lifted first-order probabilistic inference. In *In Proceedings of IJCAI-05, 19th International Joint Conference on Artificial Intelligence*, pages 1319–1325. Morgan Kaufmann, 2005.
- [65] A.S. Bregman. *Auditory Scene Analysis: The Perceptual Organization of Sound*. MIT Press, 1990.

- [66] P.F. Brown. *The Acoustic Modeling Problem in Automatic Speech Recognition*. PhD thesis, Carnegie Mellon University, 1987.
- [67] P.F. Brown, V.J.D. Pietra, S.A.D. Pietra, and R.L. Mercer. The mathematics of statistical machine translation: Parameter estimation. *Computational linguistics*, 19(2):263–311, 1993.
- [68] M. Buettner, R. Prasad, M. Philipose, and D. Wetherall. Recognizing daily activities with rfid-based sensors. In *Proceedings of the 11th international conference on Ubiquitous computing*, pages 51–60. ACM, 2009.
- [69] W. Buntine. A guide to the literature on learning probabilistic networks from data. *IEEE Trans. on Knowledge and Data Engineering*, 8:195–210, 1994.
- [70] K.P. Burnham and D.R. Anderson. *Model Selection and Inference : A Practical Information-Theoretic Approach*. Springer-Verlag, 1998.
- [71] E. Castillo, J.M. Gutierrez, and A.S. Hadi. *Expert Systems and Probabilistic Network Models*. Springer, 1997.
- [72] O. Çetin, H. Nock, K. Kirchhoff, J. Bilmes, and M. Ostendorf. The 2001 GMTK-based SPINE ASR system. In *In Proceedings ICSLP*, 2002.
- [73] Ö. Çetin, A. Kantor, S. King, C. Bartels, M. Magimai-Doss, J. Frankel, and K. Livescu. An articulatory feature-based tandem approach and factored observation modeling. In *Proc. of the IEEE Int. Conf. on Acoustics, Speech, and Signal Processing (ICASSP)*, Honolulu, HI, April 2007.
- [74] O. Cetin and M. Ostendorf. Cross-stream observation dependencies for multi-stream speech recognition. In *In Proc. of European Conf. on Speech Communication and Technology*, pages 2517–2520, 2003.
- [75] O. Cetin and M. Ostendorf. Multi-rate and variable-rate modeling of speech at phone and syllable time scales. In *In Proc. of Rich Transcription Fall Workshop*, Palisades, NY, 2004.
- [76] O. Cetin and M. Ostendorf. Multi-rate hidden markov models and their application to machining tool-wear classification. In *Acoustics, Speech, and Signal Processing, 2004. Proceedings. (ICASSP '04). IEEE International Conference on*, volume 5, pages V–837–40vol.5, 17-21 May 2004.
- [77] O. Cetin and M. Ostendorf. Multi-rate and variable-rate modeling of speech at phone and syllable time scales. In *Acoustics, Speech, and Signal Processing, 2005. Proceedings. (ICASSP '05). IEEE International Conference on*, volume 1, pages 665–668, March 18-23, 2005.
- [78] Özgür Çetin, Mathew Magimai-Doss, Karen Livescu, Arthur Kantor, Simon King, Chris Bartels, and Joe Frankel. Monolingual and crosslingual comparison of Tandem features derived from articulatory and phone MLPs. In *Proc. of IEEE Automatic Speech Recognition and Understanding Workshop (ASRU)*, Kyoto, Japan, December 2007.
- [79] Ozgur Cetin, Harriet Nock, Katrin Kirchhoff, Jeff Bilmes, and Mari Ostendorf. The 2001 GMTK-based SPINE ASR system. In *International Conference on Spoken Language Processing (ICSLP)*, Denver, CO, 2002.
- [80] C. Chen, K. Filali, and J. Bilmes. Frontend post-processing and backend model enhancement on the Aurora 2.0/3.0 databases. In *Intl. Conf. on Spoken Language Proc.*, 2002.

BIBLIOGRAPHY

- [81] Chia-Ping Chen and Jeff Bilmes. Low-resource noise-robust feature post-processing on aurora 2.0. In *Proc. Int. Conf. on Spoken Language Processing*, Denver, Colorado, 2002.
- [82] Chia-Ping Chen, Jeff Bilmes, and Dan Ellis. Speech feature smoothing for robust ASR. In *IEEE Int. Conf. on Acoustics, Speech, and Signal Processing*, Philadelphia, PA, March 2005.
- [83] Chia-Ping Chen, Karim Filali, and Jeff Bilmes. Frontend post-processing and backend model enhancement on the aurora 2.0/3.0 databases. In *Proc. Int. Conf. on Spoken Language Processing*, Denver, Colorado, 2002.
- [84] Chia-Ping Chen, Katrin Kirchhoff, and Jeff Bilmes. Towards simple methods of noise-robustness. Technical Report UWEETR-2002-0002, University of Washington, Dept. of EE, 2001. <https://www.ee.washington.edu/techsite/papers/refer/UWEETR-2002-0002.html>.
- [85] D. Chen, D. Jiang, I. Ravyse, and H. Sahli. Audio-visual emotion recognition based on a dbn model with constrained asynchrony. In *2009 Fifth International Conference on Image and Graphics*, pages 912–916. IEEE, 2009.
- [86] Francine R. Chen. Identification of contextual factors for pronunciation networks. *Proc. IEEE Int. Conf. on Acoustics, Speech, and Signal Processing*, pages 753–756, 1990.
- [87] S. F. Chen and J. Goodman. An empirical study of smoothing techniques for language modeling. In Arivind Joshi and Martha Palmer, editors, *Proceedings of the Thirty-Fourth Annual Meeting of the Association for Computational Linguistics*, pages 310–318, San Francisco, 1996. Association for Computational Linguistics, Morgan Kaufmann Publishers.
- [88] Xiaoyu Chen, Michael M. Hoffman, Jeff A. Bilmes, Jay R. Hesselberth, and William S. Noble. A dynamic Bayesian network for identifying protein-binding footprints from single molecule-based sequencing data. *Bioinformatics*, 26(12):i334–i342, 2010.
- [89] Y. Chen. Research on Audio-Visual Asynchronous Correlation for Speaker Identification Based on DBN. *Future Intelligent Information Systems*, pages 15–21, 2011.
- [90] Y. Chen and M. Liu. Audio-visual speaker identification with asynchronous articulatory feature. *Electronics letters*, 46(3):255–256, 2010.
- [91] Y.T. Chiang, K.C. Hsu, C.H. Lu, L.C. Fu, and J.Y.J. Hsu. Interaction models for multiple-resident activity recognition in a smart home. In *Intelligent Robots and Systems (IROS), 2010 IEEE/RSJ International Conference on*, pages 3753–3758. IEEE, 2010.
- [92] D. M. Chickering and C. Meek. Finding optimal Bayesian networks. In *18th Conf. on Uncertainty in Artificial Intelligence*, pages 94–102, 2002.
- [93] C.K. Chow and C.N. Liu. Approximating discrete probability distributions with dependence trees. *IEEE Trans. on Info. Theory*, 14, 1968.
- [94] William W. Cohen. Stacked sequential learning. In *International Joint Conference on Artificial Intelligence*, pages 671–676, 2005.
- [95] A Andrew R Conn, Katya Scheinberg, and Luis N Vicente. *Introduction to derivative-free optimization*, volume 8. SIAM, 2009.
- [96] T.H. Cormen, C.E. Leiserson, and R.L. Rivest. *Introduction to Algorithms*. McGraw Hill, 1990.

- [97] T.M. Cover and J.A. Thomas. *Elements of Information Theory*. Wiley, 1991.
- [98] R. G. Cowell, A. P. Dawid, S. L. Lauritzen, and D. J. Spiegelhalter. *Probabilistic Networks and Expert Systems*. Springer, 1999.
- [99] S.J. Cox. Hidden Markov Models for automatic speech recognition: Theory and application. In C. Wheddon and R. Linggard, editors, *Speech and Language Processing*, pages 209–230, 1990.
- [100] Jeff Bilmes Danny Wyatt, Tanzeem Choudhury and Henry Kautz. A privacy sensitive approach to modeling multi-person conversations. In *Twentieth International Joint Conference on Artificial Intelligence (IJCAI07)*, Hyderabad, India, January 2007.
- [101] K. Daoudi, D. Fohr, and C. Antoine. A new approach for multi-band speech recognition based on probabilistic graphical models. In *Proc. Int. Conf. on Spoken Language Processing*, Beijing, China, November 2000.
- [102] K. Daoudi, D. Fohr, and C. Antoine. Continuous multi-band speech recognition using bayesian networks. In *Proc. IEEE Automatic Speech Recognition and Understanding Workshop (ASRU)*, Trento, Italy Barbara, December 2001.
- [103] K. Daoudi, D. Fohr, and C. Antoine. Dynamic Bayesian networks for multi-band automatic speech recognition. *Computer Speech and Language*, 17:263—285, April—July 2003.
- [104] A. Darwiche. Recursive conditioning. *Artificial Intelligence*, 126(1-2):5–41, 2001.
- [105] A. Darwiche. A differential approach to inference in Bayesian networks. *Journal of the ACM (JACM)*, 50(3):305, 2003.
- [106] Data mining and knowledge discovery. Kluwer Academic Publishers. Maritime Institute of Technology, Maryland.
- [107] T. Dean and K. Kanazawa. Probabilistic temporal reasoning. *AAAI*, pages 524–528, 1988.
- [108] R. Dechter. *Constraint Processing*. Morgan Kaufmann, 2003.
- [109] J.R. Deller, J.G. Proakis, and J.H.L. Hansen. *Discrete-time Processing of Speech Signals*. MacMillan, 1993.
- [110] A.P. Dempster, N.M. Laird, and D.B. Rubin. Maximum-likelihood from incomplete data via the EM algorithm. *J. Royal Statist. Soc. Ser. B.*, 39, 1977.
- [111] L. Deng and K. Erler. Structural design of hidden markov model speech recognizer using multivalued phonetic features: Comparison with segmental speech units. *Journal of the Acoustical Society of America*, 92(6):3058–3067, Dec 1992.
- [112] L. Deng and H. Sameti. Transitional speech units and their representation by regressive Markov states: Applications to speech recognition. *IEEE Transactions on speech and audio processing*, 4(4):301–306, July 1996.
- [113] M. Deviren. Dynamic bayesian networks for speech recognition. In *Proceedings of The Eighteenth National Conference on Artificial Intelligence, AAAI 2002*, Edmonton, CA, July/August 2002.
- [114] M. Deviren and K. Daoudi. Continuous speech recognition using dynamic bayesian networks : A fast decoding algorithm. In *First European Workshop on Probabilistic Graphical Models (PGM'02)*, Cuenca, Spain, November 2002.

BIBLIOGRAPHY

- [115] M. Deviren and K. Daoudi. Continuous speech recognition using structural learning of dynamic bayesian networks. In *Proceedings of 11th European Signal Processing Conference (EUSIPCO'2002)*, Toulouse, France, September 2002.
- [116] M. Deviren and Khalid Daoudi. Structure learning of dynamic Bayesian networks in speech recognition. In *European Conf. on Speech Communication and Technology (Eurospeech)*, 2001.
- [117] M.M. Deza and E. Deza. *Encyclopedia of distances*. Springer Verlag, 2009.
- [118] A. Dielmann. *Automatic recognition of multiparty human interactions using dynamic Bayesian networks*. PhD thesis, The University of Edinburgh, 2009.
- [119] A. Dielmann and S. Renals. Dynamic bayesian networks for meeting structuring. In *Acoustics, Speech, and Signal Processing, 2004. Proceedings. (ICASSP '04). IEEE International Conference on*, volume 5, pages V–629–32vol.5, 17-21 May 2004.
- [120] A. Dielmann and S. Renals. Automatic dialogue act recognition using a dynamic Bayesian network. In S. Renals, S. Bengio, and J. Fiscus, editors, *Proc. Multimodal Interaction and Related Machine Learning Algorithms Workshop (MLMI-06)*, pages 178–189. Springer, 2007.
- [121] A. Dielmann and S. Renals. DBN based joint dialogue act recognition of multiparty meetings. In *Proc. IEEE ICASSP*, volume 4, pages 133–136, 2007.
- [122] A. Dielmann and S. Renals. Recognition of dialogue acts in multiparty meetings using a switching dbn. *Audio, Speech, and Language Processing, IEEE Transactions on*, 16(7):1303–1314, 2008.
- [123] Alfred Dielmann and Steve Renals. Automatic meeting segmentation using dynamic Bayesian networks. *IEEE Transactions on Multimedia*, 9(1):25–36, 2007.
- [124] T. Dietterich. Machine learning for sequential data: A review. *Structural, syntactic, and statistical pattern recognition*, pages 227–246, 2002.
- [125] V. Digalakis, M. Ostendorf, and J.R. Rohlicek. Improvements in the stochastic segment model for phoneme recognition. *Proc. DARPA Workshop on Speech and Natural Language*, 1989.
- [126] J. L. Doob. *Stochastic Processes*. Wiley, 1953.
- [127] Arnaud Doucet, Nando de Freitas, and Neil Gordon, editors. *Sequential Monte Carlo Methods in Practice*. Springer-Verlag, 2001.
- [128] J. Droppo, L. Deng, and A. Acero. Evaluation of the splice algorithm on the aurora2 database. In *European Conf. on Speech Communication and Technology (Eurospeech)*, volume 1, pages 217—220, Aalborg, Denmark, September 2001.
- [129] M. Droste, W. Kuich, and H. Vogler, editors. *Handbook of Weighted Automata : Monographs in Theoretical Computer Science*. Springer-Verlag, Berlin Heidelberg, 2009.
- [130] R.O. Duda and P.E. Hart. *Pattern Classification and Scene Analysis*. John Wiley and Sons, Inc., 1973.
- [131] R.O. Duda, P.E. Hart, and D.G. Stork. *Pattern Classification*. John Wiley and Sons, Inc., 2000.
- [132] K. Duh. Jointly labeling multiple sequences: A factorial HMM approach. In *43rd Annual Meeting of the Assoc. for Computational Linguistics (ACL 2005), Student Research Workshop*, Ann Arbor, Michigan, June 2005.

- [133] K. Duh and K. Kirchhoff. Pos tagging of dialectal arabic: a minimally supervised approach. In *Proceedings of the ACL Workshop on Computational Approaches to Semitic Languages*, 2005.
- [134] R. Durbin, S. Eddy, A. Krogh, and G. Mitchison. *Biological sequence analysis*. Cambridge University Press, 1998.
- [135] D. Edwards. *Introduction to Graphical Modeling: 2nd Edition*. Springer, 2000.
- [136] E. Eide. Automatic modeling of pronunciation variations. In *European Conf. on Speech Communication and Technology (Eurospeech)*, 6th, 1999.
- [137] K. Elenius and M. Blomberg. Effects of emphasizing transitional or stationary parts of the speech signal in a discrete utterance recognition system. In *Proc. IEEE Intl. Conf. on Acoustics, Speech, and Signal Processing*, pages 535–538, 1982.
- [138] Y. Ephraim. Hidden markov processes. *IEEE Trans. Info. Theory*, 48(6):1518–1569, June 2002.
- [139] Y. Ephraim, A. Dembo, and L. Rabiner. A minimum discrimination information approach for HMM. *IEEE Trans. Info. Theory*, 35(5):1001–1013, September 1989.
- [140] Y. Ephraim and L. Rabiner. On the relations between modeling approaches for information sources. In *Proc. IEEE Intl. Conf. on Acoustics, Speech, and Signal Processing*, pages 24–27, 1988.
- [141] Y. Ephraim and L. Rabiner. On the relations between modeling approaches for speech recognition. *IEEE Trans. Info. Theory*, 36(2):372–380, September 1990.
- [142] Ellis et. al. ICSI Sprach tools. <http://www1.icsi.berkeley.edu/dpwe/projects/sprach/sprachcore.html>.
- [143] Karim Filali and Jeff Bilmes. A dynamic Bayesian framework to model context and memory in edit distance learning: An application to pronunciation classification. In *Proceedings of the 43rd Annual Meeting of the Association for Computational Linguistics (ACL'05)*, pages 338–345, Ann Arbor, Michigan, June 2005. Association for Computational Linguistics.
- [144] Karim Filali and Jeff Bilmes. A dynamic bayesian framework to model context and memory in edit distance learning: An application to pronunciation classification. In *Proceedings of the Association for Computational Linguistics (ACL)*, 43, University of Michigan, Ann Arbor, 2005.
- [145] Karim Filali and Jeff Bilmes. Multi-dynamic bayesian networks. In *Neural Information Processing Society (NeurIPS, formerly NIPS)*, Vancouver, Canada, December 2006.
- [146] R.A. Fisher. The use of multiple measurements in taxonomic problems. *Ann. Eugen.*, 7:179–188, 1936.
- [147] M. Fleischman and D. Roy. Grounded language modeling for automatic speech recognition of sports video. *Proceedings of ACL-08: HLT*, pages 121–129, 2008.
- [148] V. Fontaine, C. Ris, and J.M.Boite. Nonlinear discriminant analysis for improved speech recognition. In *European Conf. on Speech Communication and Technology (Eurospeech)*, 5th, pages 2071–2074, 1997.
- [149] J. Forbes, T. Huang, K. Kanazawa, and S. J. Russell. The BATmobile: Towards a bayesian automated taxi. In *IJCAI*, pages 1878–1885, 1995.

BIBLIOGRAPHY

- [150] E. Fosler-Lussier. *Dynamic Pronunciation Models for Automatic Speech Recognition*. PhD thesis, University of California, Berkeley., 1999.
- [151] J. Frankel, M. Wester, and S. King. Articulatory feature recognition using dynamic Bayesian networks. In *Proc. ICSLP*, September 2004.
- [152] J. Frankel, M. Wester, and S. King. Articulatory feature recognition using dynamic Bayesian networks. *Computer Speech & Language*, 21(4):620–640, October 2007.
- [153] Joe Frankel and Simon King. A hybrid ANN/DBN approach to articulatory feature recognition. In *Proc. of 9th European Conference on Speech Communication and Technology (Eurospeech'05)*, Lisbon, Portugal, September 2005.
- [154] N. Friedman. The Bayesian structural EM algorithm. *14th Conf. on Uncertainty in Artificial Intelligence*, 1998.
- [155] N. Friedman, D. Geiger, and M. Goldszmidt. Bayesian network classifiers. *Machine Learning*, 29:131–163, 1997.
- [156] N. Friedman and D. Koller. Learning Bayesian networks from data. In *NIPS 2001 Tutorial Notes*. Neural Information Processing Systems, Vancouver, B.C. Canada, 2001.
- [157] N. Friedman, K. Murphy, and S. Russell. Learning the structure of dynamic probabilistic networks. *14th Conf. on Uncertainty in Artificial Intelligence*, 1998.
- [158] Nir Friedman, Lise Getoor, Daphne Koller, and Avi Pfeffer. Learning probabilistic relational models. In *IJCAI*, pages 1300–1309, 1999.
- [159] S. Fujishige. *Submodular Functions and Optimization*. Number 58 in Annals of Discrete Mathematics. Elsevier Science, 2nd edition, 2005.
- [160] K. Fukunaga. *Introduction to Statistical Pattern Recognition*, 2nd Ed. Academic Press, 1990.
- [161] S. Furui. Cepstral analysis technique for automatic speaker verification. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, 29(2):254–272, April 1981.
- [162] S. Furui. Speaker-independent isolated word recognition using dynamic features of speech spectrum. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, 34(1):52–59, February 1986.
- [163] Sadaoki Furui. On the role of spectral transition for speech perception. *Journal of the Acoustical Society of America*, 80(4):1016–1025, October 1986.
- [164] Harold N. Gabow. Using expander graphs to find vertex connectivity. *J. ACM*, 53(5):800–844, September 2006.
- [165] M.J.F. Gales. Semi-tied covariance matrices for hidden Markov models. *IEEE Transactions on Speech and Audio Processing*, 7(3):272–281, May 1999.
- [166] M.J.F. Gales and S.J. Young. Segmental hidden Markov models. In *European Conf. on Speech Communication and Technology (Eurospeech)*, 3rd, pages 1579–1582, 1993.
- [167] A. Garg, Y. Pavlovic, and J.M. Rehg. Audio-visual speaker detection using dynamic bayesian networks. In *Fourth IEEE International Conference on Automatic Face and Gesture Recognition*, 2000., pages 384—390, Grenoble, France, 2000.

-
- [168] D. Geiger and D. Heckerman. Knowledge representation and inference in similarity networks and Bayesian multinets. *Artificial Intelligence*, 82:45–74, 1996.
- [169] Z. Ghahramani. *Lecture Notes in Artificial Intelligence*, chapter Learning Dynamic Bayesian Networks, pages 168–197. Springer-Verlag, 1998.
- [170] Z. Ghahramani and M. Jordan. Factorial hidden Markov models. *Machine Learning*, 29, 1997.
- [171] E.J. Gilbert. On the identifiability problem for functions of finite Markov chains. *Ann. Math. Statist.*, 30:688—697, 1959.
- [172] W. R. Gilks, S. Richardson, and D.J. Spiegelhalter, editors. *Markov Chain Monte Carlo in Practice*. Oxford University Press, 1995.
- [173] J. Glass, J. Chang, M. McCandless, et al. A probabilistic framework for feature-based speech recognition. In *Fourth International Conference on Spoken Language Processing*. Citeseer, 1996.
- [174] G.H. Golub and C.F. Van Loan. *Matrix Computations*. Johns Hopkins, 1996.
- [175] P.S. Gopalakrishnan and L.R. Bahl. Fast match techniques. In C.-H. Lee, F.K. Soong, and K.K. Paliwal, editors, *Automatic Speech and Speaker Recognition: Advanced Topics*, pages 413–428. Kluwer, 1996.
- [176] Olga Goubanova and Simon King. Predicting consonant duration with Bayesian belief networks. In *Proc. Interspeech 2005*, Lisbon, Portugal, 2005.
- [177] J. N. Gowdy, A. Subramanya, C. Bartels, and J. Bilmes. DBN-based multi-stream models for audio-visual speech recognition. *Proc. IEEE Intl. Conf. on Acoustics, Speech, and Signal Processing*, May 2004.
- [178] G. Gravier, M. Sigelle, and G. Chollet. A markov random field based multi-band model. In *Proc. IEEE Intl. Conf. on Acoustics, Speech, and Signal Processing*, 2000.
- [179] G.R. Grimmett and D.R. Stirzaker. *Probability and Random Processes*. Oxford Science Publications, 1991.
- [180] G.R. Grimmett and D.R. Stirzaker. *Probability and Random Processes, 3rd Ed.* Oxford Science Publications, 2001.
- [181] X.F. Guo, W.B. Zhu, Q. Shi, S. Chen, and R. Gopinath. The IBM LVCSR system used for 1998 mandarin broadcast news transcription evaluation. In *The 1999 DARPA Broadcast News Workshop*, 1999.
- [182] D. Gusfield. *Algorithms on strings, trees, and sequences: computer science and computational biology*. Cambridge Univ Press, 1997.
- [183] T.E. Harris. On chains of infinite order. *Pacific J. Math.*, 5:707—724, 1955.
- [184] D.A. Harville. *Matrix Algebra from a Statistician’s Perspective*. Springer, 1997.
- [185] M. Hasegawa-Johnson, J. Baker, S. Borys, K. Chen, E. Coogan, S. Greenberg, A. Juneja, K. Kirchhoff, K. Livescu, S. Mohan, J. Muller, K. Sonmez, and Tianyu Wang. Landmark-based speech recognition: Report of the 2004 Johns Hopkins summer workshop. In *Acoustics, Speech, and Signal Processing, 2005. Proceedings. (ICASSP ’05). IEEE International Conference on*, volume 1, pages 213–216, March 18-23, 2005.

BIBLIOGRAPHY

- [186] T. Hastie and R. Tibshirani. Discriminant analysis by Gaussian mixtures. *Journal of the Royal Statistical Society series B*, 58:158–176, 1996.
- [187] T. Hastie, R. Tibshirani, and J. Friedman. *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*. Springer Series in Statistics. Springer, 2001.
- [188] D. Heckerman. *Probabilistic Similarity Networks*. MIT Press, 1991.
- [189] D. Heckerman. A tutorial on learning with Bayesian networks. Technical Report MSR-TR-95-06, Microsoft, 1995.
- [190] D. Heckerman, D. Geiger, and D.M. Chickering. Learning Bayesian networks: The combination of knowledge and statistical data. Technical Report MSR-TR-94-09, Microsoft, 1994.
- [191] D. Heckerman, A. Mamdani, and M. Wellman. Real-world applications of Bayesian networks. *Communications of the ACM*, 38, 1995.
- [192] D. Heckerman and C. Meek. Models and selection criteria for regression and classification. In *Proceedings of Thirteenth Conference on Uncertainty in Artificial Intelligence*, Providence, RI. Morgan Kaufmann, August 1997.
- [193] H. Hermansky, D. Ellis, and S. Sharma. Tandem connectionist feature stream extraction for conventional hmm systems. In *Proc. IEEE Intl. Conf. on Acoustics, Speech, and Signal Processing*, Istanbul, Turkey, 2000.
- [194] T. Hesterberg, N. Choi, L. Meier, and C. Fraley. Least Angle and L1 Regression: a Review. *Statistics Surveys*, 2:61–93, 2008. L1 regularization - focuses on regression.
- [195] K-U. Höffgen. Learning and robust learning of product distributions. In *Proceedings of the sixth annual conference on Computational learning theory*, pages 77–83. ACM Press, 1993.
- [196] Michael M. Hoffman, Orion J. Buske, Jie Wang, Zhiping Weng, Jeff A. Bilmes, and William Stafford Noble. Segway: a dynamic Bayesian network method for segmenting genomic data. Submitted, 2011.
- [197] J. Hopcroft and J. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison Wesley, 1979.
- [198] J.E. Hopcroft, R. Motwani, and J.D. Ullman. *Introduction to automata theory, languages, and computation, 2nd Ed.* Addison-wesley, 2007.
- [199] W. Hu, Y. Zhang, Q. Diao, and S. Huang. An efficient viterbi algorithm on dbns. In *European Conf. on Speech Communication and Technology (Eurospeech)*, 7th, Geneva, Switzerland, 2003.
- [200] X.D. Huang, A. Acero, and H.-W. Hon. *Spoken Language Processing: A Guide to Theory, Algorithm, and System Development*. Prentice Hall, 2001.
- [201] X.D. Huang, Y. Ariki, and M. Jack. *Hidden Markov Models for Speech Recognition*. Edinburgh University Press, 1990.
- [202] Thomas J. Murray IV, Panayiotis Georgiou, and Shrikanth Narayanan. Knowledge as a constraint on uncertainty for unsupervised classification: A study in part-of-speech tagging. In *In Proceedings of ICML workshop on Prior Knowledge for Text and Language Processing*, Helsinki, Finland, July 2008.

- [203] Rishabh Iyer and Jeff Bilmes. Algorithms for approximate minimization of the difference between submodular functions, with applications. In *Uncertainty in Artificial Intelligence (UAI)*, Catalina Island, USA, July 2012. AUAI.
- [204] Stefanie Jegelka and Jeff A. Bilmes. Submodularity beyond submodular energies: coupling edges in graph cuts. In *Computer Vision and Pattern Recognition (CVPR)*, Colorado Springs, CO, June 2011.
- [205] F. Jelinek. Continuous speech recognition by statistical methods. *Proceedings of the IEEE*, 64(4):532–556, 1976.
- [206] F. Jelinek. *Statistical Methods for Speech Recognition*. MIT Press, 1997.
- [207] F.V. Jensen. *An Introduction to Bayesian Networks*. Springer, 1996.
- [208] F.V. Jensen. *Bayesian Networks and Decision Graphs*. Springer, 2001.
- [209] Gang Ji and Jeff Bilmes. Multi-speaker language modeling. In *Proc. of Human Language Technology Conference*, Boston, MA, 2004 2004.
- [210] Gang Ji and Jeff Bilmes. Dialog act tagging using graphical models. In *Proc. of IEEE Int. Conf. on Acoustics, Speech, and Signal Processing*, 2005.
- [211] Gang Ji and Jeff Bilmes. Backoff model training using partially observed data: Application to dialog act tagging. In *Human Language Technology Conference/North American chapter of the Association for Computational Linguistics (HLT/NAACL-2006)*, New York, NY, June 2006.
- [212] Gang Ji and Jeff Bilmes. Backoff model training using partially observed data: Application to dialog act tagging. In *Human Language Technology/American chapter of the Association for Computational Linguistics (HLT/NAACL'06)*, New York, NY, June 2006.
- [213] Gang Ji, Jeff Bilmes, Jeff Michels, Katrin Kirchhoff, and Chris Manning. Graphical model representations of word lattices. In *EEE/ACL 2006 Workshop on Spoken Language Technology (SLT2006)*, Palm Beach, Aruba, December 2006.
- [214] Y. Jia and J. Li. Relax frame independence assumption for standard hmms by state dependent auto-regressive feature models. In *Proc. IEEE Intl. Conf. on Acoustics, Speech, and Signal Processing*, Orlando, Florida, 2001.
- [215] D. Jiang, P. Liu, I. Ravyse, H. Sahli, and W. Verhelst. Video Realistic Mouth Animation Based on an Audio Visual DBN Model with Articulatory Features and Constrained Asynchrony. In *2009 Fifth International Conference on Image and Graphics*, pages 658–662. IEEE, 2009.
- [216] Dongmei Jiang, Guoyun Lv, Ilse Ravyse, Xiaoyue Jiang, Yanning Zhang, Hichem Sahli, , and Rongchun Zhao. *Audio Visual Speech Recognition and Segmentation Based on DBN Models*. I-Tech, 2007.
- [217] M.I. Jordan, editor. *Learning in Graphical Models*. Kluwer Academic Publishers, 1998.
- [218] M.I. Jordan. Graphical models. *Statistical Science*, 19(1):140–155, 2004.
- [219] M.I. Jordan. *An Introduction to Graphical Models*. to be published, 20XX.
- [220] Roberto J. Bayardo Jr. and Robert Schrag. Using CSP look-back techniques to solve exceptionally hard SAT instances. In *Principles and Practice of Constraint Programming*, pages 46–60, 1996.

BIBLIOGRAPHY

- [221] Roberto J. Bayardo Jr. and Robert C. Schrag. Using CSP look-back techniques to solve real-world SAT instances. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence (AAAI'97)*, pages 203–208, Providence, Rhode Island, 1997.
- [222] B.-H. Juang, W. Chou, and C.-H. Lee. Minimum classification error rate methods for speech recognition. *IEEE Trans. on Speech and Audio Signal Processing*, 5(3):257–265, May 1997.
- [223] B-H Juang and S. Katagiri. Discriminative learning for minimum error classification. *IEEE Trans. on Signal Processing*, 40(12):3043–3054, December 1992.
- [224] B.-H. Juang and L.R. Rabiner. Mixture autoregressive hidden Markov models for speech signals. *IEEE Trans. Acoustics, Speech, and Signal Processing*, 33(6):1404–1413, December 1985.
- [225] B.H. Juang. On the hidden markov model and dynamic time warping for speech recognition - a unified view. *AT&T Bell Laboratories Technical Journal*, 63:1213—1243, 1984.
- [226] D. Jurafsky and J. H. Martin. *Speech and Language Processing*. Prentice Hall, 2000.
- [227] D. Jurafsky and J. H. Martin. *Speech and Language Processing, 2nd Ed.* Prentice Hall, 2008.
- [228] F. Jurcicek, J. Svec, and L. Muller. Extension of HVS semantic parser by allowing left-right branching. *IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 4993–4996, April 2008.
- [229] G. Sagerer K. Kirchhoff, G.A. Fink. Conversational speech recognition using acoustic and articulatory input. In *Proceedings of ICASSP 2000*, 2000.
- [230] S. Kakade, Y. W. Teh, and S. Roweis. An alternate objective function for Markovian fields. In *Proceedings of the International Conference on Machine Learning*, volume 19, 2002.
- [231] A. Kantor. *Pronunciation modeling for large vocabulary speech recognition*. PhD thesis, UIUC, 2011.
- [232] A. Kantor and A. Hasegawa-Johnson. Stream weight tuning in dynamic bayesian networks. In *Acoustics, Speech and Signal Processing, 2008. ICASSP 2008. IEEE International Conference on*, pages 4525–4528. IEEE, 2008.
- [233] J. Karhunen. Neural approaches to independent component analysis and source separation. In *Proc 4th European Symposium on Artificial Neural Networks (ESANN '96)*, 1996.
- [234] S. Katagiri, B.-H. Juang, and C.-H.-Lee. Pattern recognition using a family of design algorithms based upon the generalized probabilistic descent method. *Proceedings of the IEEE*, 1998.
- [235] P. Kenny, M. Lennig, and P. Mermelstein. A linear predictive HMM for vector-valued observations with applications to speech recognition. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, 38(2):220–225, February 1990.
- [236] S. King, J. Frankel, K. Livescu, E. McDermott, K. Richmond, , and M. Wester. Speech production knowledge in automatic speech recognition. *Journal of the Acoustical Society of America*, 121(2):723–742, February 2007.
- [237] K. Kirchhoff. Syllable-level desynchronisation of phonetic features for speech recognition. In *Proc. Int. Conf. on Spoken Language Processing*, 1996.

- [238] K. Kirchhoff. Combining acoustic and articulatory information for speech recognition in noisy and reverberant environments. In *Proceedings of the International Conference on Spoken Language Processing*, 1998.
- [239] K. Kirchhoff. *Robust Speech Recognition Using Articulatory Information*. PhD thesis, University of Bielefeld, Germany, 1999.
- [240] K. Kirchhoff, S. Parandekar, and J. Bilmes. Mixed-memory markov models for automatic language identification. In *Proceedings of ICASSP*, Orlando, Florida, 2002.
- [241] K. Kirchhoff, Joe Shmo, and Jim Smim. Novel approaches to arabic speech recognition: Report from the 2002 johns-hopkins summer workshop. In *Proc. IEEE Intl. Conf. on Acoustics, Speech, and Signal Processing*, Hong Kong, 2003.
- [242] K. Kirchhoff and D. Vergyri. Cross-dialectal acoustic data sharing for arabic speech recognition. In *Acoustics, Speech, and Signal Processing, 2004. Proceedings. (ICASSP '04). IEEE International Conference on*, volume 1, pages I–765–8vol.1, 17-21 May 2004.
- [243] U. Kjærulff. Triangulation of graphs - algorithms giving small total space. Technical Report R90-09, Department of Mathematics and Computer Science. Aalborg University, 1990.
- [244] U. Kjærulff. A computational scheme for reasoning in dynamic probabilistic networks. In *Proceedings of the 8th Conference on Uncertainty in Artificial Intelligence (UAI-92)*, pages 121–129, San Francisco, 1992. Morgan Kaufmann Publishers.
- [245] U. Kjærulff. Optimal decomposition of probabilistic networks by simulated annealing. *Statistics and Computing*, 2(7-17), 1992.
- [246] U. Kjærulff. dHugin: A computational system for dynamic time-sliced Bayesian networks. *International Journal of Forecasting, Special Issue on Probability Forcasting*, 1995. Also, Aalborg University. Technical report.
- [247] Aaron A. Klammer, Sheila M. Reynolds, Jeff A. Bilmes, Michael J. MacCoss, and William Stafford Noble. Modeling peptide fragmentation with dynamic Bayesian networks for peptide identification. *Bioinformatics*, 24(13):i348–356, 2008.
- [248] D. Klein and C.D. Manning. Conditional structure versus conditional estimation in nlp models. In *Proceedings of the ACL-02 conference on Empirical methods in natural language processing-Volume 10*, pages 9–16. Association for Computational Linguistics, 2002.
- [249] D. Koller and N. Friedman. *Probabilistic Graphical Models: Principles and Techniques*. MIT Press, August 2009.
- [250] Y. Konig. *REMAP: Recursive Estimation and Maximization of A Posterior Probabilities in Transition-based Speech Recognition*. PhD thesis, U.C. Berkeley, 1996.
- [251] P. Krause. Learning probabilistic networks. *Philips Research Labs Tech. Report*, 1998.
- [252] A. Krogh and S. K. Riis. Hidden neural networks. *Neural Computation*, 11(2):541–563, 1998.
- [253] A. Krogh and J. Vedelsby. Neural network ensembles, cross validation, and active learning. In *Advances in Neural Information Processing Systems 7*. MIT Press, 1995.

BIBLIOGRAPHY

- [254] F. R. Kschischang, B. Frey, and H.-A. Loeliger. Factor graphs and the sum-product algorithm. *IEEE Trans. Inform. Theory*, 47(2):498–519, 2001.
- [255] N. Kumar. *Investigation of Silicon Auditory Models and Generalization of Linear Discriminant Analysis for Improved Speech Recognition*. PhD thesis, Johns Hopkins University, 1997.
- [256] John Lafferty, Andrew McCallum, and Fernando Pereira. Conditional random fields: Probabilistic models for segmenting and labeling sequence data. In *Proc. 18th International Conf. on Machine Learning*, pages 282–289. Morgan Kaufmann, San Francisco, CA, 2001.
- [257] Harlan Lane and Bernard Tranel. The Lombard sign and the role of hearing in speech. *Journal of Speech and Hearing Research*, 14:677–709, 1971.
- [258] S.L. Lauritzen. *Graphical Models*. Oxford Science Publications, 1996.
- [259] Eugene Lawler. *Combinatorial optimization: networks and matroids*. Holt, Rinehart, and Winston, 1976.
- [260] C.-H. Lee, E. Giachin, L.R. Rabiner, R. Pieraccini, and A.E. Rosenberg. Improved acoustic modeling for speaker independent large vocabulary continuous speech recognition. *Proc. IEEE Intl. Conf. on Acoustics, Speech, and Signal Processing*, 1991.
- [261] Su-In Lee, Varun Ganapathi, and Daphne Koller. Efficient structure learning of markov networks using l1-regularization. *Neural Information Processing Society (NIPS)*, 2006. L1 regularization for structure learning.
- [262] E. Levin. Word recognition using hidden control neural architecture. In *Proc. IEEE Intl. Conf. on Acoustics, Speech, and Signal Processing*, pages 433–436. IEEE, 1990.
- [263] E. Levin. Hidden control neural architecture modeling of nonlinear time varying systems and its applications. *IEEE Trans. on Neural Networks*, 4(1):109–116, January 1992.
- [264] S.E. Levinson. Continuously variable duration hidden Markov models for automatic speech recognition. *Computer Speech and Language*, I:29–45, 1986.
- [265] S.E. Levinson, L.R. Rabiner, and M.M. Sondhi. An introduction to the application of the theory of probabilistic functions of a Markov process to automatic speech recognition. *The Bell System Technical Journal*, pages 1035–1073, 1983.
- [266] Xiao Li. *Regularized Adaptation: Theory, Algorithms and Applications*. PhD thesis, University of Washington, 2007.
- [267] Xiao Li, Jonathan Malkin, and Jeff A. Bilmes. A graphical model approach to pitch tracking. In *International Conference on Spoken Language Processing (ICSLP)*, October 2004.
- [268] Hui Lin and Jeff Bilmes. Polyphase speech recognition. In *IEEE Int. Conf. on Acoustics, Speech, and Signal Processing*, Las Vegas, NV, April 2008.
- [269] Hui Lin and Jeff Bilmes. Polyphase speech recognition. In *Proc. IEEE Intl. Conf. on Acoustics, Speech, and Signal Processing*, Las Vegas, NV, April 2008.
- [270] Hui Lin, Jeff Bilmes, Dimitra Vergyri, and Katrin Kirchhoff. OOV detection by joint word/phone lattice alignment. In *IEEE Automatic Speech Recognition and Understanding Workshop (ASRU)*, Kyoto, Japan, December 2007.

- [271] Hui Lin and Jeff A. Bilmes. How to select a good training-data subset for transcription: Submodular active selection for sequences. In *Proc. Annual Conference of the International Speech Communication Association (INTERSPEECH)*, Brighton, UK, September 2009.
- [272] Hui Lin, Alex Stupakov, and Jeff Bilmes. Spoken keyword spotting via multi-lattice alignment. In *Proceedings of Interspeech*, Brisbane, Australia, September 2008.
- [273] Hui Lin, Alex Stupakov, and Jeff Bilmes. Spoken keyword spotting via multi-lattice alignment. In *Proc. Annual Conference of the International Speech Communication Association (INTERSPEECH)*, Brisbane, Australia, September 2008.
- [274] Hui Lin, Alex Stupakov, and Jeff Bilmes. Improving multi-lattice-alignment based spoken keyword spotting. In *Proc. IEEE Intl. Conf. on Acoustics, Speech, and Signal Processing*, Taipei, Taiwan, 2009.
- [275] H. Linhart and W. Zucchini. *Model Selection*. Wiley, 1986.
- [276] K. Livescu, J. Glass, , and J. Bilmes. Hidden feature modeling for speech recognition using dynamic bayesian networks. In *Proc. EUROSPEECH*, Geneva, Switzerland, August-September 2003.
- [277] K. Livescu and J. Glass. Feature-based pronunciation modeling for speech recognition. In *Proc. HLT/NAACL*, Boston, Massachusetts, May 2004.
- [278] K. Livescu and J. Glass. Feature-based pronunciation modeling with trainable asynchrony probabilities. In *Proc. ICSLP*, Jeju, Korea, October 2004.
- [279] K. Livescu and J. Glass. Feature-based pronunciation modeling with trainable asynchrony probabilities. In *Proc. Int. Conf. on Spoken Language Processing*, Jeju, South Korea, October 2004.
- [280] K. Livescu, J. Glass, and J. Bilmes. Hidden feature models for speech recognition using dynamic bayesian networks. In *European Conf. on Speech Communication and Technology (Eurospeech)*, 7th, Geneva, Switzerland, 2003.
- [281] Karen Livescu, James Glass, and Jeff Bilmes. Hidden feature models for speech recognition using dynamic bayesian networks. In *European Conf. on Speech Communication and Technology (Eurospeech)*, 8th, Geneva, Switzerland, 2003.
- [282] Karen Livescu, Özgür Çetin, Mark Hasegawa-Johnson, Simon King, Chris Bartels, Nash Borges, Arthur Kantor, Partha Lal, Lisa Yung, Ari Bezman, Stephen Dawson-Haggerty, and Bronwyn Woods. Articulatory feature-based methods for acoustic and audio-visual speech recognition: Summary from the 2006 JHU summer workshop. In *Proc. of the IEEE Int. Conf. on Acoustics, Speech, and Signal Processing (ICASSP)*, Honolulu, HI, April 2007.
- [283] B.T. Logan and P.J. Moreno. Factorial HMMs for acoustic modeling. *Proc. IEEE Int. Conf. on Acoustics, Speech, and Signal Processing*, 1998.
- [284] B.T. Logan and A.J. Robinson. Enhancement and recognition of noisy speech within an autoregressive hidden markov model framework using noise estimates from the noisy signal. *Proc. IEEE Int. Conf. on Acoustics, Speech, and Signal Processing*, 2:843—846, 1998.
- [285] Guoyun Lv and Dongmei Jiangand Rongchun Zhao. Single stream DBN model based triphone for continuous speech recognition. In *Ninth IEEE International Symposium on Multimedia Workshops*, pages 240–245, December 2007.

BIBLIOGRAPHY

- [286] Ning Ma, Chris D. Bartels, Jeff A. Bilmes, and Phil D. Green. Modelling the prepausal lengthening effect for speech recognition: A dynamic bayesian network approach. In *Proc. IEEE Intl. Conf. on Acoustics, Speech, and Signal Processing*, Taipei, Taiwan, 2009.
- [287] Shuangge Ma and Jian Huang. Penalized feature selection and classification in bioinformatics. *Briefings in Bioinformatics*, June 2008. L1 regularization - focuses on classification.
- [288] I.L. MacDonald and W. Zucchini. *Hidden Markov and Other Models for Discrete-valued Time Series*. Chapman and Hall, 1997.
- [289] C. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *Design Automation Conference (DAC)*, Las Vegas, June 2001.
- [290] A.L. Madsen and D. Nilsson. Solving influence diagrams using HUGIN, Shafer-Shenoy and Lazy propagation. In *Uncertainty in Artificial Intelligence*, volume 17, pages 337–345. Citeseer, 2001.
- [291] F. Mairesse, M. Gašić, F. Jurčíček, S. Keizer, B. Thomson, K. Yu, and S. Young. Phrase-based statistical language generation using graphical models and active learning. In *Proceedings of the 48th Annual Meeting of the Association for Computational Linguistics*, pages 1552–1561. Association for Computational Linguistics, 2010.
- [292] Jon Malkin, Xiao Li, and Jeff Bilmes. A graphical model for formant tracking. In *Proc. of IEEE Int. Conf. on Acoustics, Speech, and Signal Processing*, Philadelphia, PA, March 2005.
- [293] A. Mandal, K.R.P. Kumar, G. Athithan, and C.C. Sekhar. A graphical model based decoder for recognition of loss-concealed voip speech. In *Advances in Pattern Recognition, 2009. ICAPR'09. Seventh International Conference on*, pages 179–182. IEEE, 2009.
- [294] C. D. Manning and H. Schütze. *Foundations of Statistical Natural Language Processing*. MIT Press, 1999.
- [295] K.V. Mardia, J.T. Kent, and J.M. Bibby. *Multivariate Analysis*. Academic Press, 1979.
- [296] D. McAllester, M. Collins, and F. Pereira. Case-factor diagrams for structured probabilistic modeling. *Journal of Computer and System Sciences*, 74(1):84–96, 2008.
- [297] A. McCallum, D. Freitag, and F. Pereira. Maximum entropy markov models for information extraction and segmentation. In *Proceedings of the Seventeenth International Conference on Machine Learning*, pages 591–598, 2000.
- [298] G.J. McLachlan. *Finite Mixture Models*. Wiley Series in Probability and Statistics, 2000.
- [299] G.J. McLachlan and T. Krishnan. *The EM Algorithm and Extensions*. Wiley Series in Probability and Statistics, 1997.
- [300] M. Meilă. *Learning with Mixtures of Trees*. PhD thesis, MIT, 1999.
- [301] Marie-Jean Meurs, Fabrice Lefevre, and Renato de Mori. Spoken language interpretation: On the use of dynamic bayesian networks for semantic composition. *Acoustics, Speech, and Signal Processing, IEEE International Conference on*, 0:4773–4776, 2009.
- [302] M.J. Meurs, F. Lefèvre, and R. De Mori. Learning bayesian networks for semantic frame composition in a spoken dialog system. In *Proceedings of Human Language Technologies: The 2009 Annual Conference of the North American Chapter of the Association for Computational Linguistics, Companion Volume: Short Papers*, pages 61–64. Association for Computational Linguistics, 2009.

- [303] M.J. Meurs, F. Lefèvre, and R.D. Mori. A bayesian approach to semantic composition for spoken language interpretation. In *Ninth Annual Conference of the International Speech Communication Association*, 2008.
- [304] N. Mirghafori and N. Morgan. Combining connectionist multi-band and full-band probability streams for speech recognition of natural numbers. In *Proceedings of the International Conference on Spoken Language Processing*, pages 743–746, 1998.
- [305] M. Mohri. Weighted automata algorithms. In M. Droste, W. Kuich, and H. Vogler, editors, *Handbook of Weighted Automata : Monographs in Theoretical Computer Science*. Springer-Verlag, Berlin Heidelberg, 2009.
- [306] M. Mohri, F. Pereira, and M. Riley. Weighted finite-state transducers in speech recognition. *Computer Speech and Language*, 16(1):69–88, 2002.
- [307] M. Mohri, F. C. N. Pereira, and M. Riley. The design principles of a weighted finite-state transducer library. *Theoretical Computer Science*, 231(1):17–32, 2000.
- [308] Mehryar Mohri. Finite-state transducers in language and speech processing. *Comput. Linguist.*, 23(2):269–311, 1997.
- [309] M. Montemerlo, S. Thrun, D. Koller, and B. Wegbreit. FastSLAM: A factored solution to the simultaneous localization and mapping problem. In *Proceedings of the AAAI National Conference on Artificial Intelligence*, Edmonton, Canada, 2002. AAAI.
- [310] N. Morgan and H. Bourlard. Continuous speech recognition. *IEEE Signal Processing Magazine*, 12(3), May 1995.
- [311] N. Morgan and B. Gold. *Speech and Audio Signal Processing*. John Wiley and Sons, 1999.
- [312] K. Murphy. *Dynamic Bayesian Networks: Representation, Inference and Learning*. PhD thesis, U.C. Berkeley, Dept. of EECS, CS Division, 2002.
- [313] K. Murphy. Dynamic Bayesian Networks, 2003. from book: Probabilistic Graphical Models, M. Jordan (to appear).
- [314] Kevin P Murphy. *Machine learning: a probabilistic perspective*. Cambridge, MA, 2012.
- [315] N.-S.-Kim and C.-K.-Un. Frame-correlated hidden markov model based on extended logarithmic pool. *tsap*, 5(2), March 1997.
- [316] P. Nabende. Comparison of applying pair hmms and dbn models in transliteration identification. *Computational Linguistics in the Netherlands 2010*, page 107, 2010.
- [317] H. Nagamochi and T. Ibaraki. *Algorithmic Aspects of Graph Connectivity*. Cambridge, 2008.
- [318] M. Narasimhan and J. Bilmes. PAC-learning bounded tree-width graphical models. In *Uncertainty in Artificial Intelligence: Proceedings of the Twentieth Conference (UAI-2004)*. Morgan Kaufmann Publishers, 2004.
- [319] Mukund Narasimhan and Jeff Bilmes. A submodular-supermodular procedure with applications to discriminative structure learning. In *Uncertainty in Artificial Intelligence (UAI)*, Edinburgh, Scotland, July 2005. Morgan Kaufmann Publishers.

BIBLIOGRAPHY

- [320] R.E. Neapolitan. *Learning Bayesian Networks*. Prentice Hall, 2003.
- [321] R.E. Neapolitan. *Probabilistic Reasoning in Expert Systems*. John Wiley and Sons, 1990.
- [322] A. Nefian, L. Liang, X. Pi, and K. Murphy. Dynamic bayesian networks for audio-visual speech recognition. *EURASIP, Journal of Applied Signal Processing*, 11:1—15, 2002.
- [323] A. Nefian, L. Liang, X. Pi, L. Xiaoxiang, C. Mao, and K. Murphy. A coupled hmm for audio-visual speech recognition. In *Proc. IEEE Intl. Conf. on Acoustics, Speech, and Signal Processing*, 2002.
- [324] H. Ney, U. Essen, and R. Kneser. On structuring probabilistic dependencies in stochastic language modelling. *Computer Speech and Language*, 8:1–38, 1994.
- [325] H. Ney, S. Ortmanns, and T.H. Aachen. Progress in dynamic programming search for LVCSR. *Proceedings of the IEEE*, 88(8):1224–1240, 2000.
- [326] A. Ng and M. Jordan. On discriminative vs. generative classifiers: A comparison of logistic regression and naive bayes. In *Neural Information Processing Systems (NIPS)*, 14, Vancouver, Canada, December 2002.
- [327] P. Nguyen. Techware: Speech recognition software and resources on the web [Best of the Web]. *Signal Processing Magazine, IEEE*, 26(3):102–105, 2009.
- [328] H.J. Nock and S.J. Young. Loosely-coupled HMMs for ASR. In *Proc. Int. Conf. on Spoken Language Processing*, Beijing, China, 2000.
- [329] H. Noda and M.N. Shirazi. A MRF-based parallel processing algorithm for speech recognition using linear predictive HMM. *Proc. IEEE Intl. Conf. on Acoustics, Speech, and Signal Processing*, 1994.
- [330] M. Ostendorf, V. Digalakis, and O. Kimball. From HMM’s to segment models: A unified view of stochastic modeling for speech recognition. *IEEE Trans. Speech and Audio Proc.*, 4(5), September 1996.
- [331] M. Ostendorf, A. Kannan, O. Kimball, and J. Rohlicek. Continuous word recognition based on the stochastic segment model. *Proc. DARPA Workshop CSR*, 1992.
- [332] C. Pal, C. Sutton, and A. McCallum. Sparse forward-backward using minimum divergence beams for fast training of conditional random fields. In *ICASSP 2006*, 2006.
- [333] K.K. Paliwal. Use of temporal correlations between successive frames in a hidden Markov model based speech recognizer. *Proc. IEEE Intl. Conf. on Acoustics, Speech, and Signal Processing*, pages II–215/18, 1993.
- [334] A. Papoulis. *Probability, Random Variables, and Stochastic Processes, 3rd Edition*. McGraw Hill, 1991.
- [335] D. B. Paul. An efficient A* stack decoder algorithm for continuous speech recognition with a stochastic language model. *Proc. IEEE Intl. Conf. on Acoustics, Speech, and Signal Processing*, 1992.
- [336] J. Pearl. *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. Morgan Kaufmann, 2nd printing edition, 1988.
- [337] J. Pearl. Jeffrey’s rule, passage of experience, and neo-bayesianism. In H. E. Kyburg, R. P. Loui, and G. N. Carlson, editors, *Knowledge Representation and Defeasible Reasoning*, pages 245–266. Kluwer, Boston, 1990.

- [338] J. Pearl. *Causality*. Cambridge, 2000.
- [339] Judea Pearl. *Causality University Press*. Cambridge, 2nd edition, 2009.
- [340] Franz Pernkopf and Jeff Bilmes. Discriminative versus generative parameter and structure learning of bayesian network classifiers. In *Proc. of International Conference on Machine Learning*, Bonn, Germany, 2005.
- [341] Franz Pernkopf and Jeff Bilmes. Efficient heuristics for discriminative structure learning of bayesian network classifiers. *JMLR: Journal of Machine Learning Research*, 11:2323–2360, August 2010.
- [342] S.D. Pietra, V.D. Pietra, and J. Lafferty. Inducing features of random fields. Technical Report CMU-CS-95-144, CMU, May 1995.
- [343] Walter Pirovano and Jaap Heringa. Multiple sequence alignment. In *Bioinformatics*, pages 143–161. Springer, 2008.
- [344] A.B. Poritz. Linear predictive hidden Markov models and the speech signal. *Proc. IEEE Intl. Conf. on Acoustics, Speech, and Signal Processing*, pages 1291–1294, 1982.
- [345] A.B. Poritz. Hidden Markov models: A guided tour. *Proc. IEEE Intl. Conf. on Acoustics, Speech, and Signal Processing*, pages 7–13, 1988.
- [346] B. Pytlak, A. Ghoshal, D. Karakos, and S. Khudanpur. TRECVID 2005 experiments at johns hopkins univeristy: Using hidden markov models for video retrieval. In *Proceedings of the 2005 TREC Video Retrieval Evaluation Workshop*, November 2005.
- [347] L.R. Rabiner and B.-H. Juang. *Fundamentals of Speech Recognition*. Prentice Hall Signal Processing Series, 1993.
- [348] L.R. Rabiner and B.H. Juang. An introduction to hidden Markov models. *IEEE ASSP Magazine*, 1986.
- [349] L.R. Rabiner, B.H. Juang, S.E. Levinson, and M.M. Sondhi. Some properties of continuous hidden markov model representations. *AT&T Technical Journal*, 64:1251—1270, 1985.
- [350] L.R. Rabiner and R.W. Schafer. *Digital Processing of Speech Signals*. Prentice Hall, 1978.
- [351] Alvin Raj, Amarnag Subramanya, Jeff Bilmes, and Dieter Fox. Rao-blackwellized particle filters for recognizing activities and spatial context from wearable sensors. *Experimental Robotics: The 10th International Symposium, Springer Tracts in Advanced Robotics (STAR)*, 2006.
- [352] W. Ran and R. Wang. A framework and token passing model for continuous speech recognition with dynamic bayesian networks. In *Proceedings of the Fifth IASTED International Conference on Signal Processing, Pattern Recognition, and Applications*. Acta Press Inc.,# 80, 4500-16 Avenue N. W, Calgary, AB, T 3 B 0 M 6, Canada,, 2008.
- [353] M. Reyes-Gomez, B. Raj, and D. Ellis. Multi-channel source separation by factorial hmms. In *Proc. IEEE Intl. Conf. on Acoustics, Speech, and Signal Processing*, Hong Kong, April 2003.
- [354] Sheila Reynolds, Jeff Bilmes, and William Noble. Predicting nucleosome positioning using multiple evidence tracks. In *Proceedings of the International Conference on Research in Computational Molecular Biology (RECOMB)*, Lisbon, Portugal, April 2010.

BIBLIOGRAPHY

- [355] Sheila M. Reynolds and Jeff A. Bilmes. Part-of-speech tagging using virtual evidence and negative training. In *Human Language Technology (HLT) Conference/ Conference on Empirical Methods in Natural Language Processing*, October 2005.
- [356] Sheila M. Reynolds, Lukas Kall, Michael E. Riffle, Jeff A. Bilmes, and William Noble. Transmembrane topology and signal peptide prediction using dynamic bayesian networks. *PLoS Computational Biology*, 4(11), November 2008.
- [357] M. Richardson, J. Bilmes, and C. Diorio. Hidden-articulator markov models for speech recognition. In *Proc. of the ISCA ITRW ASR2000 Workshop*, Paris, France, 2000. LIMSI-CNRS.
- [358] M. Richardson, J. Bilmes, and C. Diorio. Hidden-articulator markov models: Performance improvements and robustness to noise. In *Proc. Int. Conf. on Spoken Language Processing*, Beijing, China, 2000.
- [359] M. Richardson, J. Bilmes, and C. Diorio. Hidden-articulator Markov models for speech recognition. *Speech Communication*, 41:511, October 2003.
- [360] M. Richardson and P. Domingos. Markov logic networks. *Machine Learning*, 62(1):107–136, 2006.
- [361] T. S. Richardson. *Learning in Graphical Models*, chapter Chain Graphs and Symmetric Associations. Kluwer Academic Publishers, 1998.
- [362] Carsten Riggelsen, Matthias Ohrnberger, and Frank Scherbaum. Dynamic Bayesian networks for real-time classification of seismic signals. In *11th European Conference on Principles and Practice of Knowledge Discovery in Databases*, Warsaw, Poland, September 2007.
- [363] Michael D. Riley. A statistical model for generating pronunciation networks. *Proc. IEEE Intl. Conf. on Acoustics, Speech, and Signal Processing*, pages 737–740, 1991.
- [364] C. Robert and G. Casella. *Monte Carlo statistical methods*. Springer Verlag, 2004.
- [365] Christian P. Robert. *The Bayesian Choice*. Springer, 2001.
- [366] R. T. Rockafellar. *Convex Analysis*. Princeton, 1970.
- [367] D. J. Rose, R. E. Tarjan, and G. S. Lueker. Algorithmic aspects of vertex elimination on graphs. *SIAM Journal Computing*, 5(2):266–282, 1976.
- [368] D.J. Rose. A graph-theoretic study of the numerical solution of sparse positive definite systems of linear equations. In R.C. Reid, editor, *Graph-Theory and Computing*. Academic Press, N.Y., 1972.
- [369] R. Rosenfeld. *Adaptive Statistical Language Modeling: A Maximum Entropy Approach*. PhD thesis, School of Computer Science, CMU, Pittsburgh, PA, April 1994.
- [370] R. Rosenfeld. Two decades of statistical language modeling: Where do we go from here? *Proceedings of the IEEE*, 88(8), 2000.
- [371] D. B. Rowe. *Multivariate Bayesian Statistics: Models for Source Separation and Signal Unmixing*. CRC Press, Boca Raton, FL, 2002.
- [372] S. Roweis and Z. Ghahramani. A unifying review of linear gaussian models. *Neural Computation*, 11:305–345, 1999.
- [373] S.J. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*, 2nd Ed. Prentice Hall, 2003.

-
- [374] K. Saenko, K. Livescu, J. Glass, and T. Darrell. Production domain modeling of pronunciation for visual speech recognition. In *Proc. ICASSP*, Philadelphia, 2005.
 - [375] K. Saenko, K. Livescu, M. Siracusa, K. Wilson, J. Glass, and T. Darrell. Visual speech recognition with loosely synchronized feature streams. In *Proc. ICCV*, October 2005.
 - [376] Kate Saenko, Karen Livescu, James Glass, and Trevor Darrell. Multistream articulatory feature-based models for visual speech recognition. *IEEE TRANSACTIONS ON PATTERN ANALYSIS AND MACHINE INTELLIGENCE*, 31(9), 2009.
 - [377] L. Sang, Z. Wu, Y. Yang, and W. Zhang. Automatic speaker recognition using dynamic bayesian network. In *Proc. IEEE Intl. Conf. on Acoustics, Speech, and Signal Processing*, 2003.
 - [378] S. Sanghai, P. Domingos, and D. Weld. Dynamic probabilistic relational models. In *IJCAI-03*, 2003.
 - [379] L.K. Saul and M.I. Jordan. Boltzmann chains and hidden Markov models. *Advances in neural information processing systems*, pages 435–442, 1995.
 - [380] Virginia Savova and Leon Peshkin. Bayesian networks for the recursive recovery of syntactic dependencies. In *In Proceedings of CogSci*, 2005.
 - [381] Virginia Savova and Leon Peshkin. Dependency parsing with dynamic bayesian network. In *In Proceedings of AAAI*, 2005.
 - [382] Virginia Savova and Leonid Peshkin. Dependency parsing with dynamic bayesian network. *Clinical Orthopaedics and Related Research*, abs/cs/0703135, 2007.
 - [383] J. Schenk, B. Hörnler, A. Braun, and G. Rigoll. Graphical models: Statistical inference vs. determination. *Acoustics, Speech, and Signal Processing, IEEE International Conference on*, 0:1717–1720, 2009.
 - [384] J. Schenk, B. Hörnler, B. Schuller, A. Braun, and G. Rigoll. GMs in On-Line Handwritten Whiteboard Note Recognition: The Influence of Implementation and Modeling. In *2009 10th International Conference on Document Analysis and Recognition*, pages 877–880. IEEE, 2009.
 - [385] B. Schölkopf and A. Smola. *Learning with Kernels: Support Vector Machines, Regularization, Optimization and Beyond*. MIT Press, 2002.
 - [386] Alexander Schrijver. *Combinatorial optimization: polyhedra and efficiency*, volume 24. Springer Verlag, 2003.
 - [387] S. Schwarzler, J. Geiger, J. Schenk, M. Al-Hames, B. Hornler, G. Ruske, and G. Rigoll. Combining statistical and syntactical systems for spoken language understanding with graphical models. In *Ninth Annual Conference of the International Speech Communication Association*, 2008.
 - [388] S. Schwärzler, G. Ruske, F. Wallhoff, and G. Rigoll. Using graphical models for an intelligent mixed-initiative dialog management system. *Human Interface and the Management of Information. Information and Interaction*, pages 201–209, 2009.
 - [389] Frank Seide, Gang Li, and Dong Yu. Conversational speech transcription using context-dependent deep neural networks. In *INTERSPEECH*, pages 437–440, 2011.

- [390] Tevfik Metin Sezgin and Randall Davis. Temporal sketch recognition in interspersed drawings. In *SBIM '07: Proceedings of the 4th Eurographics workshop on Sketch-based interfaces and modeling*, pages 15–22, New York, NY, USA, 2007. ACM.
- [391] TM Sezgin and R. Davis. Sketch recognition in interspersed drawings using time-based graphical models. *Computers & Graphics*, 32(5):500–510, 2008.
- [392] R.D. Shachter. Bayes-ball: The rational pastime for determining irrelevance and requisite information in belief networks and influence diagrams. In *Uncertainty in Artificial Intelligence*, 1998.
- [393] C.E. Shannon. A mathematical theory of communication. *Bell Systems Tech. Journal*, 27:379–423, 1948.
- [394] Pradeep Shenoy and Rajesh P. N. Rao. Dynamic bayesian networks for brain-computer interfaces. In *Advances in NIPS 17*, 2005.
- [395] T. Shinozaki and S. Furui. Hidden mode HMM using bayesian network for modeling speaking rate fluctuation. In *Automatic Speech Recognition and Understanding, 2003. ASRU '03. 2003 IEEE Workshop on*, pages 417–422, 30 Nov.-3 Dec. 2003.
- [396] Takahiro Shinozaki and Sadaoki Furui. Time adjustable mixtureweights for speaking rate fluctuation. In *Proc. Eurospeech*, 2003.
- [397] F. J. Smith, J. Ming, P. O’Boyle, and A.D. Irvine. A hidden markov model with optimized inter-frame dependence. In *icassp*, volume I, pages 209—212, Detroit, MI, May 1995.
- [398] P. Smyth, D. Heckerman, and M.I. Jordan. Probabilistic independence networks for hidden Markov probability models. Technical Report A.I. Memo No. 1565, C.B.C.L. Memo No. 132, MIT AI Lab and CBCL, 1996.
- [399] T. Stephenson, H. Bourlard, S. Bengio, and A. Morris. Automatic speech recognition using dynamic bayesian networks with both acoustic and articulatory variables. In *Proc. Int. Conf. on Spoken Language Processing*, pages 951–954, Beijing, China, 2000.
- [400] T. Stephenson, J. Escofet, M. Magimai-Doss, and H. Bourlard. Dynamic Bayesian network based speech recognition with pitch and energy as auxiliary variables. In *2002 IEEE International Workshop on Neural Networks for Signal Processing (NNSP 2002)*, pages 637–646, Martigny, Switzerland, September 2002.
- [401] T. A. Stephenson, M. Magimai-Doss, and H. Bourlard. Auxiliary variables in conditional Gaussian mixtures for automatic speech recognition. In *Seventh International Conference on Spoken Language Processing (ICSLP 2002)*, volume 4, pages 2665–2668, Denver, CO, USA, September 2002.
- [402] Todd A. Stephenson, M. Mathew, and Hervé Bourlard. Modeling auxiliary information in Bayesian network based ASR. In *7th European Conference on Speech Communication and Technology (Eurospeech 2001)*, volume 4, pages 2765–2768, Aalborg, Denmark, September 2001. IDIAP-RR 01-11.
- [403] D. Stirzaker. *Elementary Probability*. Cambridge, 1994.
- [404] G. Strang. *Linear Algebra and its applications, 3rd Edition*. Saunders College Publishing, 1988.
- [405] A. Subramanya, J. Bilmes, and C. Chen. Focused word segmentation for ASR. In *9th European Conf. on Speech Communication and Technology (Eurospeech)*, 2005.

- [406] A. Subramanya, J. Gowdy, C. Bartels, and J. Bilmes. DBN based multi-stream models for audio-visual speech recognition. In *Proc. ICASSP*, Montreal, Canada, 2004.
- [407] Amar Subramanya, Chris Bartels, Jeff Bilmes, and Patrick Nguyen. Uncertainty in training large vocabulary speech recognizers. In *Proc. of IEEE Automatic Speech Recognition and Understanding Workshop (ASRU)*, Kyoto, Japan, December 2007.
- [408] Amarnag Subramanya and Jeff Bilmes. Virtual evidence for training speech recognizers using partially labeled data. In *Human Language Technology Conference — North American Association for Computational Linguistics (NAACL-HLT-07)*, Rochester, NY, April 2007.
- [409] Amarnag Subramanya and Jeff Bilmes. Applications of virtual-evidence based speech recognizer training. In *Proceedings of Interspeech*, Brisbane, Australia, September 2008.
- [410] Amarnag Subramanya, Alvin Raj, Jeff Bilmes, and Dieter Fox. Hierarchical models for activity recognition. In *Proc. of IEEE Multimedia Signal Processing (MMSP) Conference*, October 2006.
- [411] Amarnag Subramanya, Alvin Raj, Jeff Bilmes, and Dieter Fox. Hierarchical models for activity recognition. In *IEEE Multimedia Signal Processing Conference (MMSP)*, Victoria, British Columbia, October 2006.
- [412] Amarnag Subramanya, Alvin Raj, Jeff Bilmes, and Dieter Fox. Hierarchical models for activity recognition. In *IEEE Multimedia Signal Processing (MMSP) Conference*, Victoria, CA, October 2006.
- [413] Amarnag Subramanya, Alvin Raj, Jeff Bilmes, and Dieter Fox. Recognizing activities and spatial context using wearable sensors. In *Proc. of 21st Conference on Uncertainty in Artificial Intelligence (UAI06)*, July 2006.
- [414] Amarnag Subramanya, Alvin Raj, Jeff Bilmes, and Dieter Fox. Recognizing activities and spatial context using wearable sensors. In *21st Conference on Uncertainty in Artificial Intelligence (UAI06)*, Cambridge, MA, July 2006.
- [415] Y. Sun, J.F. Gemmeke, B. Cranen, L. Bosch, and L. Boves. Using a DBN to integrate Sparse Classification and GMM-based ASR. In *Eleventh Annual Conference of the International Speech Communication Association*, Makuhari, Chiba, Japan, September 2010.
- [416] Y. Sun, J.F. Gemmeke, B. Cranen, L. ten Bosch, and L. Boves. Improvements of a dual-input DBN for noise robust ASR. *Proc. Interspeech 2011*, 2011.
- [417] J. Švec and F. Jurčíček. Extended hidden vector state parser. In *Text, Speech and Dialogue*, pages 403–410. Springer, 2009.
- [418] S. Takahashi, T. Matsuoka, Y. Minami, and K. Shikano. Phoneme HMMs constrained by frame correlations. *Proc. IEEE Intl. Conf. on Acoustics, Speech, and Signal Processing*, 1993.
- [419] E. Talvitie and S. Singh. Simple local models for complex dynamical systems. In *Proceedings of NIPS*, 2008.
- [420] B. Taskar. *Learning Structured Prediction Models: A Large Margin Approach*. PhD thesis, Stanford University, 2004.
- [421] B. Taskar, C. Guestrin, and D. Koller. Max margin markov networks. In *Neural Information Processing Systems (NIPS)*, 16, Vancouver, Canada, December 2003.

BIBLIOGRAPHY

- [422] L. Terry. *Audio-Visual Asynchrony Modeling and Analysis for Speech Alignment and Recognition*. PhD thesis, NORTHWESTERN UNIVERSITY, 2011.
- [423] L. Terry and A.K. Katsaggelos. A phone-viseme dynamic bayesian network for audio-visual automatic speech recognition. In *Pattern Recognition, 2008. ICPR 2008. 19th International Conference on*, pages 1–4. IEEE, 2008.
- [424] L.H. Terry, K. Livescu, J.B. Pierrehumbert, and A.K. Katsaggelos. Audio-visual anticipatory coarticulation modeling by human and machine. In *Eleventh Annual Conference of the International Speech Communication Association*, 2010.
- [425] M.E. Tipping and C.M. Bishop. Probabilistic principal component analysis. *Journal of the Royal Statistical Society, Series B*, 61(3):611–622, 1999.
- [426] D.M. Titterington, A.F.M. Smith, and U.E. Makov. *Statistical Analysis of Finite Mixture Distributions*. John Wiley and Sons, 1985.
- [427] R. Davis T.M. Sezgin. Sketch recognition in interspersed drawings using time-based graphical models. *Computers & Graphics*, May 2008.
- [428] V. Vapnik. *Statistical Learning Theory*. Wiley, 1998.
- [429] A.P. Varga and R.K. Moore. Hidden Markov model decomposition of speech and noise. In *Proc. IEEE Intl. Conf. on Acoustics, Speech, and Signal Processing*, pages 845–848, Alburquerque, April 1990.
- [430] A.P. Varga and R.K. Moore. Simultaneous recognition of concurrent speech signals using hidden makov model decomposition. In *European Conf. on Speech Communication and Technology (Eurospeech)*, 2nd, 1991.
- [431] D. Vergyri and K. Kirchhoff. Automatic diacritization of arabic for acoustic modeling in speech recognition. In *COLING Workshop on Arabic-script Based Languages*, Geneva, Switzerland,, 2004.
- [432] A.J. Viterbi. Error bounds for convolutional codes and an symptotically optimum decodings algorithm. *IEEE Transactions on Information Theory*, IT-13:260–267, 1967.
- [433] Martin Wainwright, Pradeep Ravikumar, and John Lafferty. High-dimensional graphical model selection using ℓ_1 -regularized logistic regression. *Neural Information Processing Society (NIPS)*, 2006. L1 regularization for structure learning.
- [434] M.J. Wainwright and M.I. Jordan. Graphical models, exponential families, and variational inference. *Foundations and Trends® in Machine Learning*, 1(1-2):1–305, 2008.
- [435] Y. Wang, Z. Y. Wu, L. H. Cai, and H. Meng. Modeling the synchrony between audio and visual modalities for speaker identification. In *Proceedings of the 8th Phonetic Conference of China and the International Symposium on Phonetic Frontiers (PCC2008)*, Beijing, China, April 2008.
- [436] C.J. Wellekens. Explicit time correlation in hidden Markov models for speech recognition. *Proc. IEEE Intl. Conf. on Acoustics, Speech, and Signal Processing*, pages 384–386, 1987.
- [437] M. Wester, J. Frankel, and S. King. Asynchronous articulatory feature recognition using dynamic Bayesian networks. In *Proc. IEICI Beyond HMM Workshop*, Kyoto, December 2004.
- [438] J. Whittaker. *Graphical Models in Applied Multivariate Statistics*. John Wiley and Son Ltd., 1990.

- [439] J.G. Wilpon, C.-H. Lee, and L.R. Rabiner. Improvements in connected digit recognition using higher order spectral and energy features. *Proc. IEEE Intl. Conf. on Acoustics, Speech, and Signal Processing*, 1991.
- [440] Shuly Winter. Formal language theory. In A. Clark, C. Fox, and S. Lappin, editors, *Handbook of Computational Linguistics and Natural Language Processing*, pages 12–42. Wiley-Blackwell, 2010.
- [441] M. Wöllmer, F. Eyben, A. Graves, B. Schuller, and G. Rigoll. Bidirectional lstm networks for context-sensitive keyword detection in a cognitive virtual agent framework. *Cognitive Computation*, 2(3):180–190, 2010.
- [442] M. Wollmer, B. Schuller, F. Eyben, and G. Rigoll. Combining long short-term memory and dynamic bayesian networks for incremental emotion-sensitive artificial listening. *Selected Topics in Signal Processing, IEEE Journal of*, 4(5):867–881, 2010.
- [443] P.C. Woodland. Optimizing hidden Markov models using discriminative output distributions. In *Proc. IEEE Intl. Conf. on Acoustics, Speech, and Signal Processing*, 1991.
- [444] P.C. Woodland. Hidden Markov models using vector linear prediction and discriminative output distributions. In *Proc. IEEE Intl. Conf. on Acoustics, Speech, and Signal Processing*, pages I–509–512, 1992.
- [445] P.C. Woodland and D. Povey. Large scale discriminative training for speech recognition. In *ICSA ITRW ASR2000*, 2000.
- [446] T. Wu, J.Y. Hsu, and Y. Chiang. Continuous recognition of daily activities from multiple heterogeneous sensors. In *Proceedings of AAAI*, 2009.
- [447] Danny Wyatt, Tanzeem Choudhury, and Jeff Bilmes. Discovering long range properties of social networks with multi-valued time-inhomogeneous models. In *Twenty-Fifth AAAI Conference on Artificial Intelligence (AAAI-10)*, Atlanta, GA., July 2010.
- [448] Yang Xiang. Temporally invariant junction tree for inference in dynamic Bayesian network. *Lecture Notes in Computer Science*, 1600:473–487, 1999.
- [449] L. Xin, G. Ji, J. Bilmes, and M. Ostendorf. DBN multistream models for Mandarin toneme recognition. In *Proc. ICASSP*, 2005.
- [450] S. Yaman, D. Hakkani-Tür, and G. Tur. Social role discovery from spoken language using dynamic bayesian networks. In *Eleventh Annual Conference of the International Speech Communication Association*, 2010.
- [451] M.A. Yatbaz and D. Yuret. Unsupervised part of speech tagging using unambiguous substitutes from a statistical language model. In *Proceedings of the 23rd International Conference on Computational Linguistics: Posters*, pages 1391–1398. Association for Computational Linguistics, 2010.
- [452] S. Young. A review of large-vocabulary continuous-speech recognition. *IEEE Signal Processing Magazine*, 13(5):45–56, September 1996.
- [453] S. Young, J. Jansen, J. Odell, D. Ollason, and P. Woodland. *The HTK Book*. Entropic Labs and Cambridge University, 2.1 edition, 1990’s.
- [454] K.H. Yeo and H.C. Wang. Joint estimation of feature transformation parameters and gaussian mixture model for speaker identification. *Speech Communications*, 3(1), 1999.

BIBLIOGRAPHY

- [455] Dong Zhang, Daniel Gatica-Perez, Samy Bengio, and Deb Roy. Learning influence among interacting Markov chains. IDIAP-RR 48, IDIAP, Martigny, Switzerland, 2005. Published in NIPS, Dec, 2005.
- [456] H. Zhang. Sato: An efficient propositional prover. In *Proc. of International Conference on Automated Deduction (CADE)*, 1997.
- [457] L. Zhang, C. Madigan, M. Moskewicz, and S. Malik. Efficient conflict driven learning in a boolean satisfiability solver. In *Proceedings of ICCAD*, San Jose, CA, November 2001.
- [458] L. Zhang and S. Malik. Conflict driven learning in a quantified boolean satisfiability solver. In *Proceedings of International Conference on Computer Aided Design (ICCAD)*, San Jose, CA, November 2002.
- [459] L. Zhang and S. Malik. The quest for efficient boolean satisfiability solvers. In *Proceedings of 8th International Conference on Computer Aided Deduction (CADE)*, Copenhagen, Denmark, July 2002.
- [460] Y. Zhang, Q. Diao, S. Huang, W. Hu, C. Bartels, and J. Bilmes. DBN based multi-stream models for speech. In *Proc. IEEE Intl. Conf. on Acoustics, Speech, and Signal Processing*, Hong Kong, China, April 2003.
- [461] Y. Zhang, Q. Diao, S. Huang, W. Hu, C. Bartels, and J. Bilmes. DBN based multi-stream models for speech. In *Proc. IEEE Intl. Conf. on Acoustics, Speech, and Signal Processing*, Hong Kong, China, April 2003.
- [462] G. Zweig. *Speech Recognition with Dynamic Bayesian Networks*. PhD thesis, U.C. Berkeley, 1998.
- [463] G. Zweig. Bayesian network structures and inference techniques for automatic speech recognition. *Computer Speech and Language*, 17:173—193, April—July 2003.
- [464] G. Zweig, J. Bilmes, T. Richardson, K. Filali, K. Livescu, P. Xu, K. Jackson, Y. Brandman, E. Sandness, E. Holtz, J. Torres, and B. Byrne. Structurally discriminative graphical models for automatic speech recognition — results from the 2001 Johns Hopkins summer workshop. *Proc. IEEE Intl. Conf. on Acoustics, Speech, and Signal Processing*, 2002.
- [465] G. Zweig and M. Padmanabhan. Dependency modeling with bayesian networks in a voicemail transcription system. In *European Conf. on Speech Communication and Technology (Eurospeech)*, 6th, 1999.
- [466] G. Zweig and M. Padmanabhan. Exact alpha-beta computation in logarithmic space with application to map word graph construction. *Int. Conf. on Spoken Lanugage Processing*, 2000.
- [467] G. Zweig and S. Russell. Probabilistic modeling with Bayesian networks for automatic speech recognition. In *Int. Conf. on Spoken Language Processing*, 1998.
- [468] G. Zweig and S. Russell. Speech recognition with dynamic Bayesian networks. *AAAI-98*, 1998.
- [469] G. Zweig and S. Russell. Probabilistic modeling with Bayesian networks for automatic speech recognition. *Australian Journal of Intelligent Information Processing*, 5(4):253–260, 1999.
- [470] Geoff Zweig, Jeff Bilmes, Thomas Richardson, Karim Filali, Karen Livescu, Peng Xu, Kirk Jackson, Yigal Brandman, Eric Sandness, Eva Holtz, Jerry Torres, and Bill Byrne. Structurally discriminative graphical models for automatic speech recognition — results from the 2001 Johns Hopkins summer workshop. *Proc. IEEE Intl. Conf. on Acoustics, Speech, and Signal Processing*, 2002.

Part IX

Index

