

Easy Factor Graph: the flexible and efficient tool for managing undirected graphical models

Generated by Doxygen 1.9.1

1 What is EFG	1
2 Theoretical background on factor graphs	3
2.1 Preliminaries	3
2.2 Message Passing	7
2.2.1 Belief propagation	8
2.2.2 Message Passing	9
2.3 Maximum a posteriori estimation	13
2.4 Gibbs sampling	14
2.5 Sub graphs	15
2.6 Learning	15
2.6.1 Learning of unconditioned model	17
2.6.1.1 Gradient of α	17
2.6.1.2 Gradient of β	18
2.6.2 Learning of conditioned model	19
2.6.2.1 Gradient of β	20
2.6.3 Learning of modular structure	21
2.6.3.1 Gradient of α	21
2.6.3.2 Gradient of β	21
3 Import models from xml files	23
4 Samples	27
4.1 Sample 01: Potential handling	27
4.1.1 Variables creation	27
4.1.1.1 part 01	27
4.1.1.2 part 02	27
4.1.2 Factors creation	27
4.1.2.1 part 01	27
4.1.2.2 part 02	29
4.2 Sample 02: Belief propagation, part A	30
4.2.1 part 01	30
4.2.2 part 02	33
4.2.3 part 03	33
4.3 Sample 03: Belief propagation, part B	34
4.3.1 part 01	34
4.3.2 part 02	34
4.3.3 part 03	36
4.3.4 part 04	36
4.4 Sample 04: Hidden Markov model like structure	37
4.5 Sample 05: Matricial structure	37
4.6 Sample 06: Learning, part A	39
4.6.1 part 01	39

4.6.2 part 02	39
4.6.3 part 03	40
4.6.4 part 04	40
4.7 Sample 07: Learning, part B	40
4.8 Sample 08: Sub-graphing	41
4.8.1 part 01	41
4.8.2 part 02	41
5 Namespace Index	45
5.1 Namespace List	45
6 Hierarchical Index	47
6.1 Class Hierarchy	47
7 Class Index	51
7.1 Class List	51
8 Namespace Documentation	55
8.1 EFG Namespace Reference	55
8.1.1 Detailed Description	56
8.1.2 Typedef Documentation	56
8.1.2.1 SmartMap	56
8.1.2.2 SmartSet	56
8.2 EFG::categoric Namespace Reference	57
8.2.1 Detailed Description	57
8.2.2 Function Documentation	57
8.2.2.1 get_complementary()	58
8.3 EFG::factor Namespace Reference	58
8.3.1 Detailed Description	58
8.4 EFG::io Namespace Reference	59
8.4.1 Detailed Description	59
8.4.2 Function Documentation	59
8.4.2.1 import_train_set()	59
8.5 EFG::io::json Namespace Reference	60
8.5.1 Detailed Description	60
8.6 EFG::io::xml Namespace Reference	60
8.6.1 Detailed Description	60
8.7 EFG::model Namespace Reference	60
8.7.1 Detailed Description	60
8.8 EFG::strct Namespace Reference	61
8.8.1 Detailed Description	62
8.9 EFG::train Namespace Reference	62
8.9.1 Detailed Description	62
8.9.2 Function Documentation	62

8.9.2.1 set_ones()	62
8.9.2.2 train_model()	63
9 Class Documentation	65
9.1 EFG::strct::BaselineLoopyPropagator Class Reference	65
9.2 EFG::train::BaseTuner Class Reference	65
9.3 EFG::strct::BeliefAware Class Reference	66
9.3.1 Detailed Description	67
9.4 EFG::train::BinaryTuner Class Reference	67
9.5 EFG::Cache< T > Class Template Reference	67
9.6 EFG::factor::Factor::CloneTrasformedImagesTag Struct Reference	68
9.7 EFG::strct::PropagationResult::ClusterInfo Struct Reference	68
9.8 EFG::factor::Function::CombinationHasher Struct Reference	68
9.9 EFG::Comparator< T > Struct Template Reference	69
9.10 EFG::train::CompositeTuner Class Reference	69
9.11 EFG::model::ConditionalRandomField Class Reference	70
9.11.1 Detailed Description	71
9.11.2 Constructor & Destructor Documentation	71
9.11.2.1 ConditionalRandomField()	71
9.11.3 Member Function Documentation	71
9.11.3.1 makeTrainSet()	71
9.11.3.2 setEvidences()	72
9.12 EFG::strct::Node::Connection Struct Reference	72
9.13 EFG::strct::GibbsSampler::SamplerNode::DynamicDependency Struct Reference	73
9.14 EFG::Error Class Reference	73
9.15 EFG::factor::Evidence Class Reference	73
9.16 EFG::strct::EvidenceRemover Class Reference	74
9.16.1 Member Function Documentation	74
9.16.1.1 removeEvidence()	74
9.16.1.2 removeEvidences()	75
9.17 EFG::strct::EvidenceSetter Class Reference	75
9.17.1 Member Function Documentation	76
9.17.1.1 setEvidence()	76
9.18 EFG::io::json::Exporter Class Reference	76
9.18.1 Member Function Documentation	77
9.18.1.1 exportToFile()	77
9.18.1.2 exportToJson()	77
9.19 EFG::io::xml::Exporter Class Reference	77
9.19.1 Member Function Documentation	78
9.19.1.1 exportToFile()	78
9.19.1.2 exportToString()	78
9.20 EFG::io::xml::ExportInfo Struct Reference	78

9.21 EFG::factor::Factor Class Reference	79
9.21.1 Constructor & Destructor Documentation	80
9.21.1.1 Factor() [1/2]	80
9.21.1.2 Factor() [2/2]	80
9.21.2 Member Function Documentation	81
9.21.2.1 cloneWithPermutedGroup()	81
9.22 EFG::factor::FactorExponential Class Reference	82
9.22.1 Detailed Description	83
9.22.2 Constructor & Destructor Documentation	83
9.22.2.1 FactorExponential()	83
9.22.3 Member Function Documentation	83
9.22.3.1 getWeight()	83
9.23 EFG::strct::FactorsAware Class Reference	84
9.23.1 Member Function Documentation	84
9.23.1.1 getAllFactors()	84
9.24 EFG::strct::FactorsConstGetter Class Reference	84
9.24.1 Member Function Documentation	85
9.24.1.1 getConstFactors()	85
9.25 EFG::strct::FactorsConstInserter Class Reference	85
9.26 EFG::train::FactorsTunableGetter Class Reference	86
9.26.1 Member Function Documentation	87
9.26.1.1 getWeights()	87
9.26.1.2 getWeightsGradient()	87
9.26.1.3 setWeights()	87
9.27 EFG::train::FactorsTunableInserter Class Reference	88
9.27.1 Member Function Documentation	89
9.27.1.1 addTunableFactor()	89
9.27.1.2 copyTunableFactor()	89
9.28 EFG::factor::Function Class Reference	89
9.28.1 Member Function Documentation	90
9.28.1.1 transform()	90
9.29 EFG::strct::GibbsSampler Class Reference	91
9.29.1 Detailed Description	91
9.29.2 Member Function Documentation	91
9.29.2.1 makeSamples()	91
9.30 EFG::model::Graph Class Reference	92
9.30.1 Detailed Description	92
9.30.2 Member Function Documentation	92
9.30.2.1 absorb()	92
9.31 EFG::strct::StateAware::GraphState Struct Reference	93
9.32 EFG::categoric::Group Class Reference	93
9.32.1 Detailed Description	94

9.32.2 Constructor & Destructor Documentation	94
9.32.2.1 Group() [1/3]	94
9.32.2.2 Group() [2/3]	94
9.32.2.3 Group() [3/3]	94
9.32.3 Member Function Documentation	95
9.32.3.1 add()	95
9.32.3.2 getVariables()	95
9.32.3.3 getVariablesSet()	96
9.32.3.4 replaceVariables()	96
9.32.3.5 size()	96
9.33 EFG::categoric::GroupRange Class Reference	96
9.33.1 Detailed Description	97
9.33.2 Constructor & Destructor Documentation	97
9.33.2.1 GroupRange()	97
9.33.3 Member Function Documentation	98
9.33.3.1 operator++()	98
9.34 std::hash< EFG::categoric::Variable > Struct Reference	98
9.35 EFG::Hasher< T > Struct Template Reference	98
9.36 EFG::strct::HiddenCluster Struct Reference	98
9.36.1 Detailed Description	99
9.37 EFG::factor::ImageFinder Class Reference	99
9.37.1 Detailed Description	99
9.37.2 Member Function Documentation	99
9.37.2.1 findTransformed()	99
9.38 EFG::factor::Immutable Class Reference	100
9.38.1 Member Function Documentation	100
9.38.1.1 getProbabilities()	101
9.39 EFG::io::json::Importer Class Reference	101
9.39.1 Member Function Documentation	101
9.39.1.1 importFromFile()	101
9.39.1.2 importFromJson()	102
9.40 EFG::io::xml::Importer Class Reference	102
9.40.1 Member Function Documentation	102
9.40.1.1 importFromFile()	102
9.41 EFG::factor::Indicator Class Reference	103
9.42 EFG::factor::Function::Info Struct Reference	103
9.43 EFG::train::TrainSet::Iterator Class Reference	103
9.43.1 Detailed Description	104
9.43.2 Constructor & Destructor Documentation	104
9.43.2.1 Iterator()	104
9.43.3 Member Function Documentation	104
9.43.3.1 size()	104

9.44 EFG::strct::LoopyBeliefPropagationStrategy Class Reference	105
9.45 EFG::factor::MergedUnaries Class Reference	105
9.46 EFG::detail::Merger< Separator > Struct Template Reference	106
9.47 EFG::factor::MessageMAP Class Reference	106
9.48 EFG::factor::MessageSUM Class Reference	107
9.49 EFG::factor::Mutable Class Reference	107
9.49.1 Member Function Documentation	108
9.49.1.1 set()	108
9.50 EFG::detail::Node< T > Struct Template Reference	108
9.51 EFG::strct::Node Struct Reference	108
9.52 EFG::detail::NodeConst< T > Struct Template Reference	109
9.53 EFG::strct::StateAware::NodeLocation Struct Reference	109
9.54 EFG::strct::Pool Class Reference	110
9.55 EFG::strct::PoolAware Class Reference	110
9.56 EFG::strct::PropagationContext Struct Reference	110
9.57 EFG::strct::PropagationResult Struct Reference	111
9.57.1 Detailed Description	111
9.58 EFG::strct::QueryManager Class Reference	111
9.58.1 Member Function Documentation	112
9.58.1.1 getHiddenSetMAP()	112
9.58.1.2 getJointMarginalDistribution() [1/2]	112
9.58.1.3 getJointMarginalDistribution() [2/2]	112
9.58.1.4 getMAP()	113
9.58.1.5 getMarginalDistribution()	113
9.59 EFG::model::RandomField Class Reference	114
9.59.1 Detailed Description	114
9.59.2 Member Function Documentation	114
9.59.2.1 absorb()	114
9.60 EFG::strct::GibbsSampler::SamplerNode Struct Reference	115
9.61 EFG::strct::GibbsSampler::SamplesGenerationContext Struct Reference	115
9.61.1 Member Data Documentation	116
9.61.1.1 transient	116
9.62 EFG::strct::PoolAware::ScopedPoolActivator Class Reference	116
9.63 EFG::factor::Factor::SimplyAntiCorrelatedTag Struct Reference	116
9.64 EFG::factor::Factor::SimplyCorrelatedTag Struct Reference	116
9.65 EFG::strct::StateAware Class Reference	117
9.65.1 Member Function Documentation	117
9.65.1.1 findVariable()	117
9.65.1.2 getAllVariables()	118
9.65.1.3 getEvidences()	118
9.65.1.4 getHiddenVariables()	118
9.65.1.5 getObservedVariables()	118

9.66 EFG::strct::HiddenCluster::TopologyInfo Struct Reference	119
9.67 EFG::train::TrainInfo Struct Reference	119
9.68 EFG::train::TrainSet Class Reference	119
9.68.1 Constructor & Destructor Documentation	119
9.68.1.1 TrainSet()	119
9.69 EFG::train::Tuner Class Reference	120
9.70 EFG::factor::UnaryFactor Class Reference	120
9.71 EFG::train::UnaryTuner Class Reference	121
9.72 EFG::strct::UniformSampler Class Reference	122
9.73 EFG::categoric::Variable Class Reference	122
9.73.1 Detailed Description	122
9.73.2 Constructor & Destructor Documentation	122
9.73.2.1 Variable()	122
9.74 EFG::Visitor< Ts > Struct Template Reference	123
9.75 EFG::VisitorConst< Ts > Struct Template Reference	124
Index	125

Chapter 1

What is EFG

Easy Factor Graph (EFG), is a simple and efficient C++ library for managing undirected graphical models.

EFG allows you to build step by step a graphical model made of unary or binary potentials, i.e. factors involving one or two variables. It contains several tools for exporting and importing graphs from textual file. EFG allows you to perform all the probabilistic queries described in Chapter 2, from marginal probabilities computation to learning the tunable parameters of a graph. All the theoretical computations described in the initial Sections of this guide are already implemented inside the library. However, you are strongly encouraged to read this guide to understand how to use such functionalities.

The rest of this guide is structured as follows. Chapter 2 will introduce the main theoretical concepts about factor graphs, with the aim of explaining the capabilities of EFG. Chapter 3 will explain the format of the xml files adopted to represent factor graphs, exploited when importing or exporting the models to or from textual files. Chapter 4 will present the examples adopted for showing how EFG works. All the remaining Chapters, will describe the structure of the classes constituting EFG ¹.

¹A similar guide, but in a html format, is also available at http://www.andreacasalino.altervista.org/__EFG_doxy_guide/index.html.

Chapter 2

Theoretical background on factor graphs

This Section will provide the basic concepts about probabilistic models. Moreover, a precise notation will be introduced and used for the rest of this document.

2.1 Preliminaries

This library is intended for managing network of categorical variables. Formally, the generic categorical variable V has a discrete domain Dom :

$$Dom(V) = \{v_0, \dots, v_n\} \quad (2.1)$$

Essentially, $Dom(V)$ contains all the possible realizations of V . The above notation will be adopted for the rest of the guide: capital letters will refer to variable names, while non capital refer to their realizations. Group of categorical variables can be considered categorical variables too, having a domain that is the Cartesian product of the domains of the variables constituting the group. Suppose X is obtained as the union of variables $V_{1,2,3,4}$, i.e. $X = \bigcup_{i=1}^4 V_i$, then:

$$Dom(X) = Dom(V_1) \times Dom(V_2) \times Dom(V_3) \times Dom(V_4) \quad (2.2)$$

The generic realization x of X is a set of realizations of the variables $V_{1,2,3,4}$, i.e. $x = \{v_1, v_2, v_3, v_4\}$. Suppose $V_{1,2,3}$ have the domains reported in the tables 2.1. The union $X = \bigcup_{i=1}^3 V_i$ is a categoric variable whose domain is made by the combinations reported in table 2.2.

The entire population of variables contained in a model is a set denoted as $\mathcal{V} = \{V_1, \dots, V_m\}$. As will be exposed in the following, the probability of $\bigcup_{V_i \in \mathcal{V}} V_i$ ¹ is computed as the product of a certain number of components called factors.

Knowing the joint probability of $V_{1,\dots,m}$, the probability distribution of a subset $S \subset \{V_1, \dots, V_m\}$ can be in general (not only for graphical models) obtained through marginalization. Assume C is the complement of S :

$$C \cup S = \bigcup_{i=1}^m V_i \quad (2.3)$$

¹ Which is the joint probability distribution of all the variables in a model

$Dom(V_1)$	$Dom(V_2)$	$Dom(V_3)$
v_{10}	v_{20}	v_{30}
v_{11}	v_{21}	v_{31}
	v_{22}	

Table 2.1 Example of domains for the group of variables $V_{1,2,3}$.

$Dom(X) = Dom(V_1 \cup V_2 \cup V_3)$
$x_0 = \{v_{10}, v_{20}, v_{30}\}$
$x_1 = \{v_{10}, v_{20}, v_{31}\}$
$x_2 = \{v_{11}, v_{20}, v_{30}\}$
$x_3 = \{v_{11}, v_{20}, v_{31}\}$
$x_4 = \{v_{10}, v_{21}, v_{30}\}$
$x_5 = \{v_{10}, v_{21}, v_{31}\}$
$x_6 = \{v_{11}, v_{21}, v_{30}\}$
$x_7 = \{v_{11}, v_{21}, v_{31}\}$
$x_8 = \{v_{10}, v_{22}, v_{30}\}$
$x_9 = \{v_{10}, v_{22}, v_{31}\}$
$x_{10} = \{v_{11}, v_{22}, v_{30}\}$
$x_{11} = \{v_{11}, v_{22}, v_{31}\}$

Table 2.2 Example of domains for the group of variables $V_{1,2,3}$.

with $C \cap S = \emptyset$, then:

$$\mathbb{P}(S = s) = \sum_{\forall \hat{c} \in Dom(C)} \mathbb{P}(S = s, C = \hat{c}) \quad (2.4)$$

In the above computation, variables in C were marginalized. Indeed they were in a certain sense eliminated, since the probability of the sub set S was of interest, no matter the realizations of all the variables in C .

A factor, sometimes also called a potential, is a positive real function describing the correlation existing among a subset of variables $D^i \subset \mathcal{V}$. Suppose factor Φ_i involves $\{X, Y, Z\}$, i.e. $D^i = \{X, Y, Z\}$. Then, $\Phi_i(X, Y, Z)$ is a function defined over $Dom(D^i)$. More formally:

$$\Phi_i(D^i) = \Phi_i(X, Y, Z) : \text{DOMAIN}(X) \times \text{DOMAIN}(Y) \times \text{DOMAIN}(Z) \longrightarrow \mathbb{R}^+ \quad (2.5)$$

The aim of Φ_i is to assume 'high' values for those combinations $d^i = \{x, y, z\}$ that are probable to appear together and low values (at least a zero) for those being improbable. The entire population of factors $\{\Phi_1, \dots, \Phi_p\}$ correlating the variables of a model, is considered for computing $\mathbb{P}(V_{1,\dots,m})$, i.e. the joint probability distribution of all the variables in the model. The energy function E of a graph is defined as the product of the factors:

$$E(V_{1,\dots,m}) = \Phi_1(D^1) \cdot \dots \cdot \Phi_p(D^p) = \prod_{i=1}^p \Phi_i(D^i) \quad (2.6)$$

E is addressed for computing the joint probability distribution of the variables in \mathcal{V} :

$$\mathbb{P}(V_{1,\dots,m}) = \frac{E(V_{1,\dots,m})}{\mathcal{Z}} \quad (2.7)$$

where \mathcal{Z} is a normalization coefficient defined as follows:

$$\mathcal{Z} = \sum_{\forall \tilde{V}_{1,\dots,m} \in Dom(\bigcup_{i=1,\dots,m} V_i)} E(\tilde{V}_{1,\dots,m}) \quad (2.8)$$

Although the general theory behind graphical models supports the existence of generic multivaried factors, this library will address only two possible types:

- Binary potentials: they involve a pair of variables.
- Unary potentials: they involve a single variable.

We can store the values in the image of a Binary potential in a two dimensional table. For instance, suppose Φ_b involves variables A and B , whose domains contains 3 and 5 possible values respectively:

$$\begin{aligned} \text{DOM}(A) &= \{a_1, a_2, a_3\} \\ \text{DOM}(B) &= \{b_1, b_2, b_3, b_4, b_5\} \end{aligned} \quad (2.9)$$

	b_0	b_1	b_2	b_3	b_4
a_0	1	4	0	0	0
a_1	0	1	0	0	0
a_2	0	0	5	0	1

Table 2.3 The values in the image of $\Phi_b(A, B)$.

a_0	a_1	a_2
0	2	0.5

Table 2.4 The values in the image of $\Phi_u(A)$.

The values assumed by $\Phi_b(A, B)$ are described by table 2.3. Essentially, $\Phi_b(A, B)$ tells us that the combinations $\{a_0, b_1\}$, $\{a_2, b_2\}$ are highly probable; while $\{a_0, b_0\}$, $\{a_1, b_1\}$ and $\{a_2, b_4\}$ are moderately probable. Let be $\Phi_u(A)$ a Unary potential involving variable A . The values characterizing Φ_u can be stored in a simple vector, see table 2.4. If $\Phi_b(A, B)$ would be the only potential in the model, the joint probability density of A and B will assume the following values ²:

$$\mathbb{P}(a_0, b_1) = \frac{\Phi_b(a_0, b_1)}{\mathcal{Z}} = \frac{4}{\mathcal{Z}} = 0.3333 \quad (2.10)$$

$$\mathbb{P}(a_2, b_2) = \frac{\Phi_b(a_2, b_2)}{\mathcal{Z}} = \frac{5}{\mathcal{Z}} = 0.4167 \quad (2.11)$$

$$\mathbb{P}(a_0, b_0) = \frac{\Phi_b(a_0, b_0)}{\mathcal{Z}} = \mathbb{P}(a_1, b_1) = \mathbb{P}(a_2, b_4) = \frac{1}{\mathcal{Z}} = 0.0833 \quad (2.12)$$

since \mathcal{Z} is equal to:

$$\mathcal{Z} = \sum_{\forall i=\{0,1,2\}, \forall j=\{0,1,2,3,4\}} \Phi_b(A = a_i, B = b_j) = 12 \quad (2.13)$$

Both Unary and Binary potentials, can be of two possible classes:

- **Factors.** The potential is simply described by a set of values characterizing the image of the factor. $\Phi_b(A, B)$ and $\Phi_u(A)$ of the previous example are both Simple shapes. Classes `EFG::distribution::factor::cnst::Factor` and `EFG::distribution::factor::modif::Factor` handles this kind of potentials.
- **Exponential Factors.** They are indicated with Ψ_i and their image set is defined as follows:

$$\Psi_i(X) = \exp(w \cdot \Phi_i(X)) \quad (2.14)$$

where Φ_i is a Simple shape. Classes `EFG::distribution::factor::cnst::FactorExponential` and `EFG::distribution::factor::modif::FactorExponential` handles this kind of potentials. The weight w , can be tunable or not. In the first case, w is a free parameter whose value is decided after training the model (see Section 2.6), otherwise is a constant. Exponential factors with fixed weights will be denoted with $\bar{\Psi}_i$.

Figure 2.1 resumes all the possible categories of factors that can be present in the models handled by this library.

Figure 2.2 reports an example of undirected graph. Set \mathcal{V} is made of 4 variables: A, B, C, D . There are 5 Binary potentials and 2 Unary ones. The graphical notation adopted for Fig. 2.2 will be adopted for the rest of this guide. Weights α, β, γ and δ are assumed for respectively $\Psi_{AC}, \Psi_{AB}, \Psi_{CD}, \Psi_B$. For the sake of clarity, the joint probability of the variables in Fig. 2.2 is computable as follows:

$$\begin{aligned} \mathbb{P}(A, B, C, D) &= \frac{E(A, B, C, D)}{\mathcal{Z}(\alpha, \beta, \gamma, \delta)} = \frac{E(A, B, CD)}{\sum_{\tilde{A}, \tilde{B}, \tilde{C}, \tilde{D}} E(\tilde{A}, \tilde{B}, \tilde{C}, \tilde{D})} \\ E(A, B, C, D) &= \Phi_A(A) \cdot \exp(\alpha \Phi_{AC}(A, C)) \cdot \exp(\beta \Phi_{AB}(A, B)) \cdot \dots \\ &\dots \Phi_{BC}(B, C) \cdot \exp(\gamma \Phi_{CD}(C, D)) \cdot \Phi_{BD}(B, D) \cdot \exp(\delta \Phi_B(B)) \end{aligned} \quad (2.15)$$

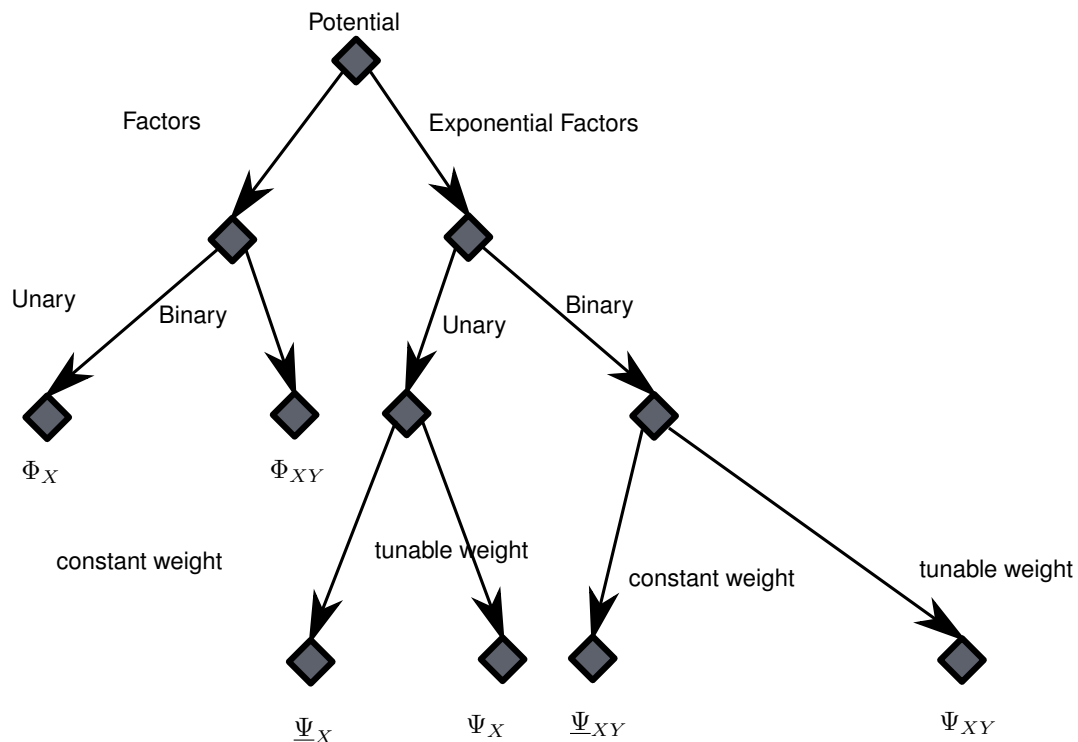


Figure 2.1 All the possible categories of factors, with the corresponding notation.

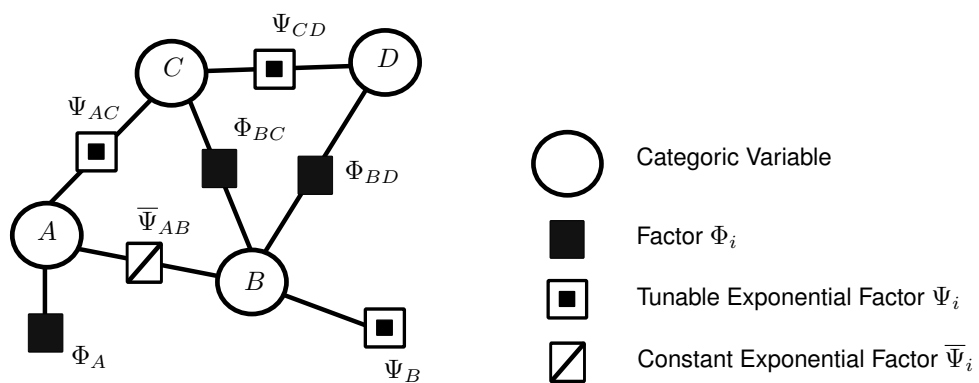


Figure 2.2 Example of graph: the legend of the right applies.

Graphical models are mainly used for performing belief propagation. Subset $\mathcal{O} = \{O_1, \dots, O_f\} \subset \mathcal{V}$ is adopted for denoting the set of evidences: those variables in the net whose value become known. \mathcal{O} can be dynamical or not. The hidden variables are contained in the complementary set $\mathcal{H} = \{H_1, \dots, H_t\}$. Clearly $\mathcal{O} \cup \mathcal{H} = \mathcal{V}$ and $\mathcal{O} \cap \mathcal{H} = \emptyset$. H will be used for referring to the union of all the variables in the hidden set:

$$H = \bigcup_{i=1}^t H_i \quad (2.16)$$

while O is used for indicating the evidences:

$$O = \bigcup_{i=1}^f O_i \quad (2.17)$$

Knowing the joint probability distribution of variables in \mathcal{V} (equation (2.7)) the conditional distribution of H w.r.t. O can be determined as follows:

$$\begin{aligned} \mathbb{P}(H = h | O = o) &= \frac{\mathbb{P}(H = h, O = o)}{\sum_{\hat{h} \in \text{Dom}(H)} \mathbb{P}(H = \hat{h}, O = o)} \\ &= \frac{E(h, o)}{\sum_{\hat{h} \in \text{Dom}(H)} E(\hat{h}, o)} = \frac{E(h, o)}{\mathcal{Z}(o)} \end{aligned} \quad (2.18)$$

The above computations are not actually done, since the number of combinations in the domain of \mathcal{H} is huge also when considering a low-medium size graph. On the opposite, the marginal probability $\mathbb{P}(H_i = h_i | O = 0)$ of a single variable in $H_i \in \mathcal{H}$ is computationally tractable. Formally $\mathbb{P}(H_i = h_i | O = 0)$ is defined as follows:

$$\mathbb{P}(H_i = h_i | O = o) = \sum_{\tilde{h} \in \{\mathcal{H} \setminus H_i\}} \mathbb{P}(H_i = h_i, \tilde{h} | O = o) \quad (2.19)$$

The above marginal distribution is essentially the conditional distribution of H_i w.r.t. O , no matter the other variables in \mathcal{H} .

A generic Random Field is a graphical model for which set \mathcal{O} (and consequently \mathcal{H}) is dynamical: the set of observations as well as the values assumed by the evidences may change during time. Random field are handled by class `RandomField`. Conditional Random Field are Random Field for which set \mathcal{O} must be decided once and cannot change after. Only the values of the evidences during time may change. Class `ConditionalRandomField` is in charge of handling Conditional Random Field. Both Random Fields and Conditional Random Fields can be learnt knowing a training set, see Section 2.6. On the opposite, class `Graph` handles constant graphs: they are conceptually similar to Random Fields but learning is not possible. Indeed, all the Exponential Shape involved must be constant.

The rest of this Chapter is structured as follows. Section 2.2.2 will introduce the message passing algorithm, which is the pillar for performing belief propagation. Section 2.3 will expose the concept of maximum a posteriori estimation, useful when querying a graph, while Section 2.4 will address Gibbs sampling for producing a training set of a known model. Section 2.5 will present the concept of subgraph which is a useful way for computing the marginal probabilities of a sub group of variables in \mathcal{H} . Finally, 2.6 will discuss how the learning of a graphical model is done, with the aim of computing the weights of the Exponential shapes that are tunable.

2.2 Message Passing

Message passing is a powerful but conceptually simple algorithm adopted for propagating the belief across a net. Such a propagation is the starting point for performing many important operations, like computing the marginal distributions of single variables or obtaining sub graphs. Before detailing the steps involved in the message passing algorithm, let's start from an example of belief propagation. Without loss of generality we assume all the factors as simple Factors.

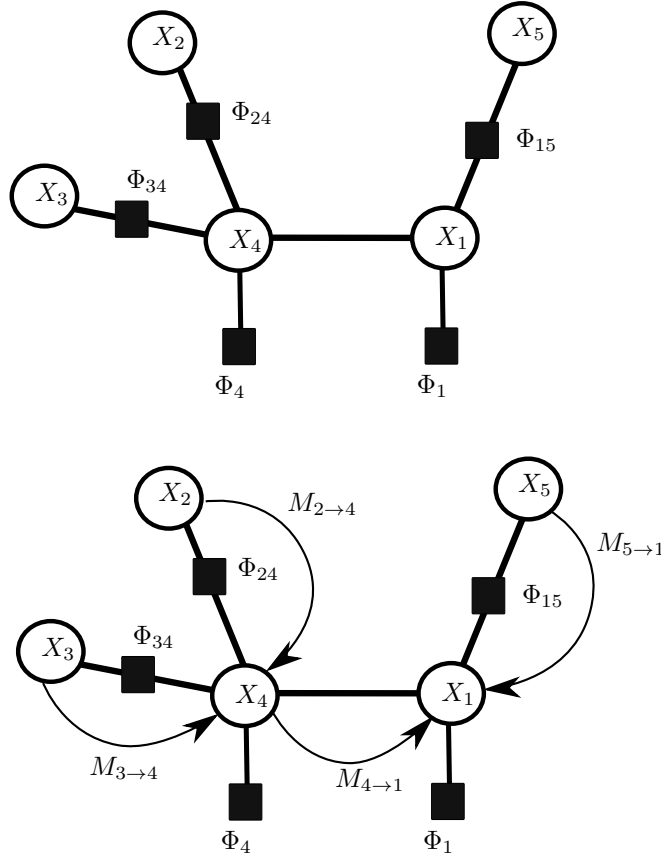


Figure 2.3 Example of graph adopted for explaining the message passing algorithm. Below are reported the messages to compute for obtaining the marginal probability of variable x_1

2.2.1 Belief propagation

Consider the graph reported in Figure 2.3. Supposing for the sake of simplicity that no evidences are available (i.e. $\mathcal{O} = \emptyset$). We are interested in computing $\mathbb{P}(X_1)$, i.e. the marginal probability of X_1 . Recalling the definition introduced in the previous Section, the marginal probability is obtained by the following computation:

$$\mathbb{P}(x_1) = \sum_{\forall \tilde{x}_{2,3,4,5} \in \cup_{i=2}^5 X_i} \mathbb{P}(x_1, \tilde{x}_{2,3,4,5}) \quad (2.20)$$

Simplifying the notation and getting rid of the normalization coefficient \mathcal{Z} we can state the following:

$$\mathbb{P}(x_1) \propto \sum_{\tilde{x}_{2,3,4,5}} E(x_1, \tilde{x}_{2,3,4,5}) \quad (2.21)$$

Adopting the algebraic properties of the sums-products we can distribute the computations as follows:

$$\mathbb{P}(x_1) \propto \Phi_1(x_1) \sum_{\tilde{x}_5} \Phi_{15}(x_1, \tilde{x}_5) \sum_{\tilde{x}_4} \Phi_{14}(x_1, \tilde{x}_4) \Phi_4(\tilde{x}_4) \sum_{\tilde{x}_2} \Phi_{24}(\tilde{x}_2, \tilde{x}_4) \sum_{\tilde{x}_3} \Phi_{34}(\tilde{x}_3, \tilde{x}_4) \quad (2.22)$$

The first variable to marginalize can be \tilde{x}_2 or \tilde{x}_3 , since they are involved in the last terms of the sums products. The 'messages' $M_{2 \rightarrow 4}$, $M_{3 \rightarrow 4}$ are defined as follows:

$$\begin{aligned} M_{2 \rightarrow 4}(\tilde{x}_4) &= \sum_{\tilde{x}_2} \Phi_{24}(\tilde{x}_2, \tilde{x}_4) \\ M_{3 \rightarrow 4}(\tilde{x}_4) &= \sum_{\tilde{x}_3} \Phi_{34}(\tilde{x}_3, \tilde{x}_4) \end{aligned} \quad (2.23)$$

²combinations having a null probability were omitted

Inserting $M_{2 \rightarrow 4}$ and $M_{3 \rightarrow 4}$ into equation (2.22) leads to:

$$\mathbb{P}(x_1) \propto \Phi_1(x_1) \sum_{\tilde{x}_5} \Phi_{15}(x_1, \tilde{x}_5) \sum_{\tilde{x}_4} \Phi_{14}(x_1, \tilde{x}_4) \Phi_4(\tilde{x}_4) M_{2 \rightarrow 4}(\tilde{x}_4) M_{3 \rightarrow 4}(\tilde{x}_4) \quad (2.24)$$

At this point the messages $M_{4 \rightarrow 1}$ and $M_{5 \rightarrow 1}$ can be computed in the following way:

$$\begin{aligned} M_{4 \rightarrow 1}(x_1) &= \sum_{\tilde{x}_4} \Phi_{14}(x_1, \tilde{x}_4) \Phi_4(\tilde{x}_4) M_{2 \rightarrow 4}(\tilde{x}_4) M_{3 \rightarrow 4}(\tilde{x}_4) \\ M_{5 \rightarrow 1}(x_1) &= \sum_{\tilde{x}_5} \Phi_{15}(x_1, \tilde{x}_5) \end{aligned} \quad (2.25)$$

After inserting $M_{4 \rightarrow 1}$ and $M_{5 \rightarrow 1}$ into equation (2.24) we obtain:

$$\begin{aligned} \mathbb{P}(x_1) &\propto \Phi_1(x_1) M_{4 \rightarrow 1}(x_1) M_{5 \rightarrow 1}(x_1) \\ \mathbb{P}(x_1) &= \frac{\Phi_1(x_1) M_{4 \rightarrow 1}(x_1) M_{5 \rightarrow 1}(x_1)}{\sum_{\tilde{x}_1} \Phi_1(\tilde{x}_1) M_{4 \rightarrow 1}(\tilde{x}_1) M_{5 \rightarrow 1}(\tilde{x}_1)} \end{aligned} \quad (2.26)$$

which ends the computations. Messages are, in a certain sense, able to simplify the graph sending some information from an area of the graph to another one. Indeed, variables can be replaced by messages, which can be treated as additional factors. Figure 2.3 resumes the computations exposed. Notice that the computation of $M_{4 \rightarrow 1}$ must be done after computing the messages $M_{2 \rightarrow 4}$ and $M_{3 \rightarrow 4}$, while $M_{5 \rightarrow 1}$ can be computed independently from all the others.

2.2.2 Message Passing

The aforementioned considerations can be extended to a general structured graph. Look at Figure 2.4: the computation of Message $M_{B \rightarrow A}$ can be performed only after having computed all the messages $M_{V_1, \dots, m \rightarrow B}$, i.e. the messages incoming from all the neighbours of B a part from A . Clearly $M_{B \rightarrow A}$ is computed as follows:

$$\begin{aligned} M_{B \rightarrow A}(a) &= \sum_{\tilde{b}} \Phi_{AB}(a, \tilde{b}) M_{V_1 \rightarrow B}(\tilde{b}) \cdots M_{V_m \rightarrow B}(\tilde{b}) \\ &= \sum_{\tilde{b}} \Phi_{AB}(a, \tilde{b}) \prod_{i=1}^m M_{V_i \rightarrow B}(\tilde{b}) \end{aligned} \quad (2.27)$$

Essentially, it's like having simplified the graph: we can append to A the message $M_{B \rightarrow A}(a)$ as it's a Factor, deleting factor Φ_{AB} and all the other portions of the graph, see Figure 2.4. In turn, $M_{B \rightarrow A}(a)$ will be adopted for computing the message outgoing from A .

The above elimination is not actually done: all messages incoming to all nodes of a graph are computed and stored for using them in subsequent queries. This is partially not true when considering the evidences. Indeed, when the values of the evidences are retrieved, variables in \mathcal{O} are temporary deleted and replaced with messages, see Figure 2.5. Suppose variable C is connected to a variable A through a binary potential $\Phi_{AC}(A, C)$ and to variable B through Φ_{BC} . Suppose also that variable C is an evidence assuming a value equal to \hat{c} , then the messages sent to A and B can be computed independently as follows:

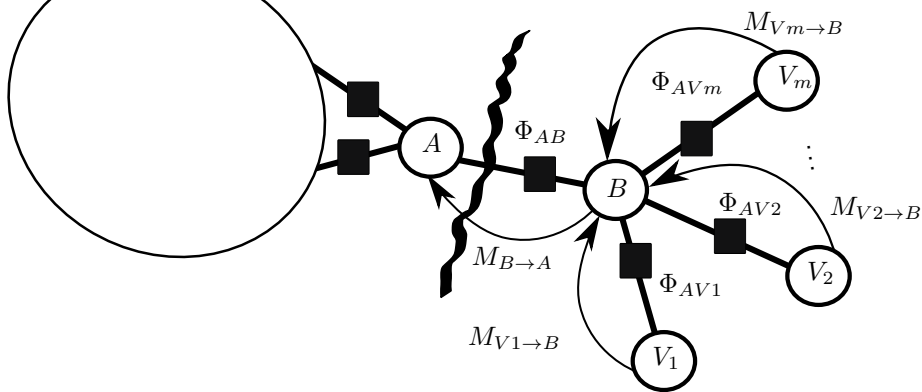
$$\begin{aligned} M_{C \rightarrow A}(a) &= \Phi_{AC}(a, \hat{c}) \\ M_{C \rightarrow B}(b) &= \Phi_{BC}(b, \hat{c}) \end{aligned} \quad (2.28)$$

Therefore all the variables that become evidences can be considered as leaves of the graph, sending messages to all the neighbouring nodes, possibly splitting an initial compact graph into many subgraphs, refer to Figure 2.5. Such computations are automatically handled by the library.

All the above considerations are valid when considering politree, i.e. graph without loops. Indeed, for these kind of graphs the message passing algorithm is able in a finite number of iterations to compute all the messages, see Figure 2.6. The same is not true when having loopy graphs (see Figure 2.7), since deadlocking situations arise: no further messages can be computed since for every nodes some incoming ones are missing. In such cases a variant of the message passing called loopy belief propagation can be adopted. Loopy belief propagation initializes all the messages to basic shapes having the values of the image all equal to 1 and then recomputes all the messages of all the variables till convergence.

You don't have to handle the latter aspect: the belief propagation mechanism is automatically handled by the library, according to the connectivity of the model for which a query is asked.

Remaining structure of the graph



Remaining structure of the graph

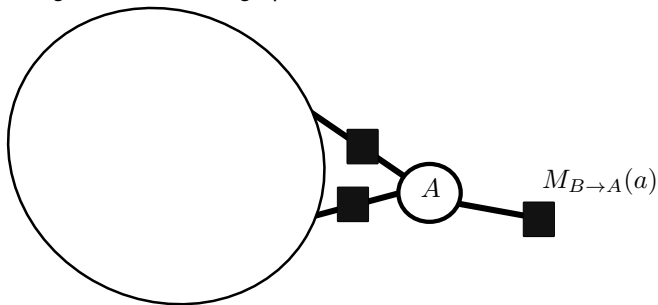


Figure 2.4 On the top the general mechanism involved in the message computation; on the bottom the simplification of the graph considering the computed message.

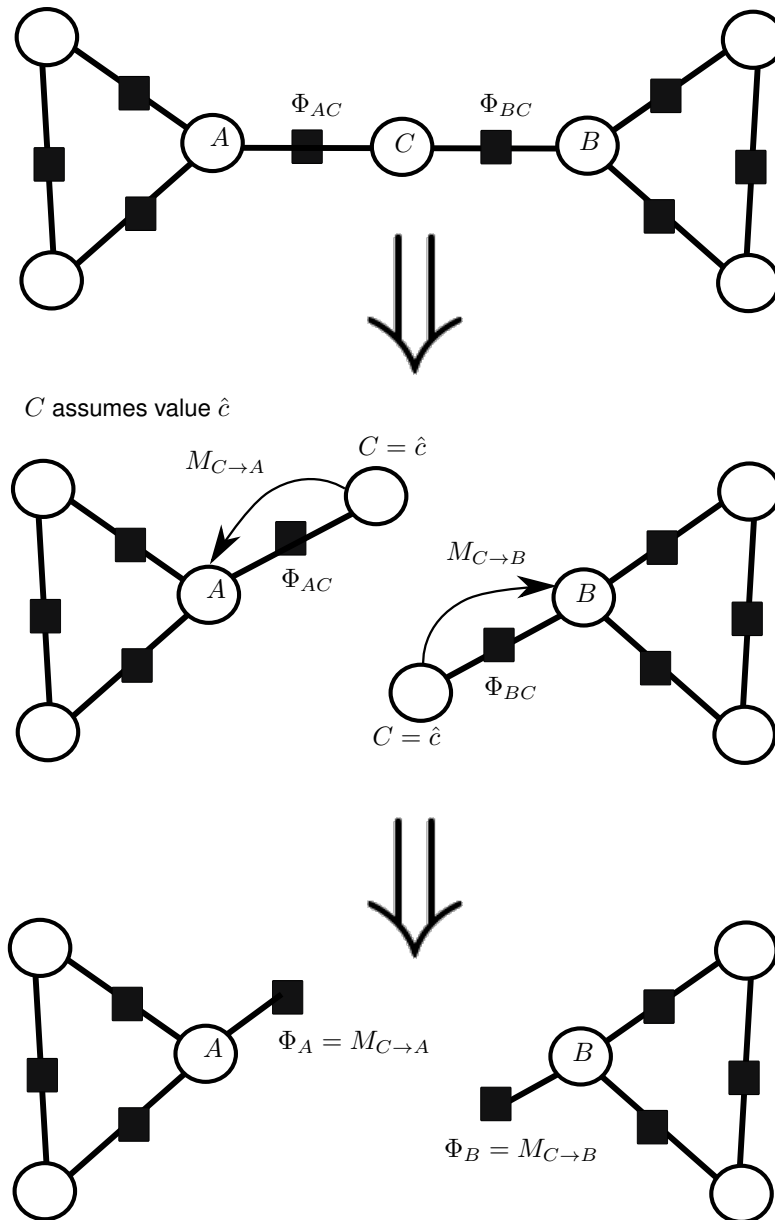


Figure 2.5 When variable C becomes an evidence, it is temporary deleted from the graph, replaced by messages.

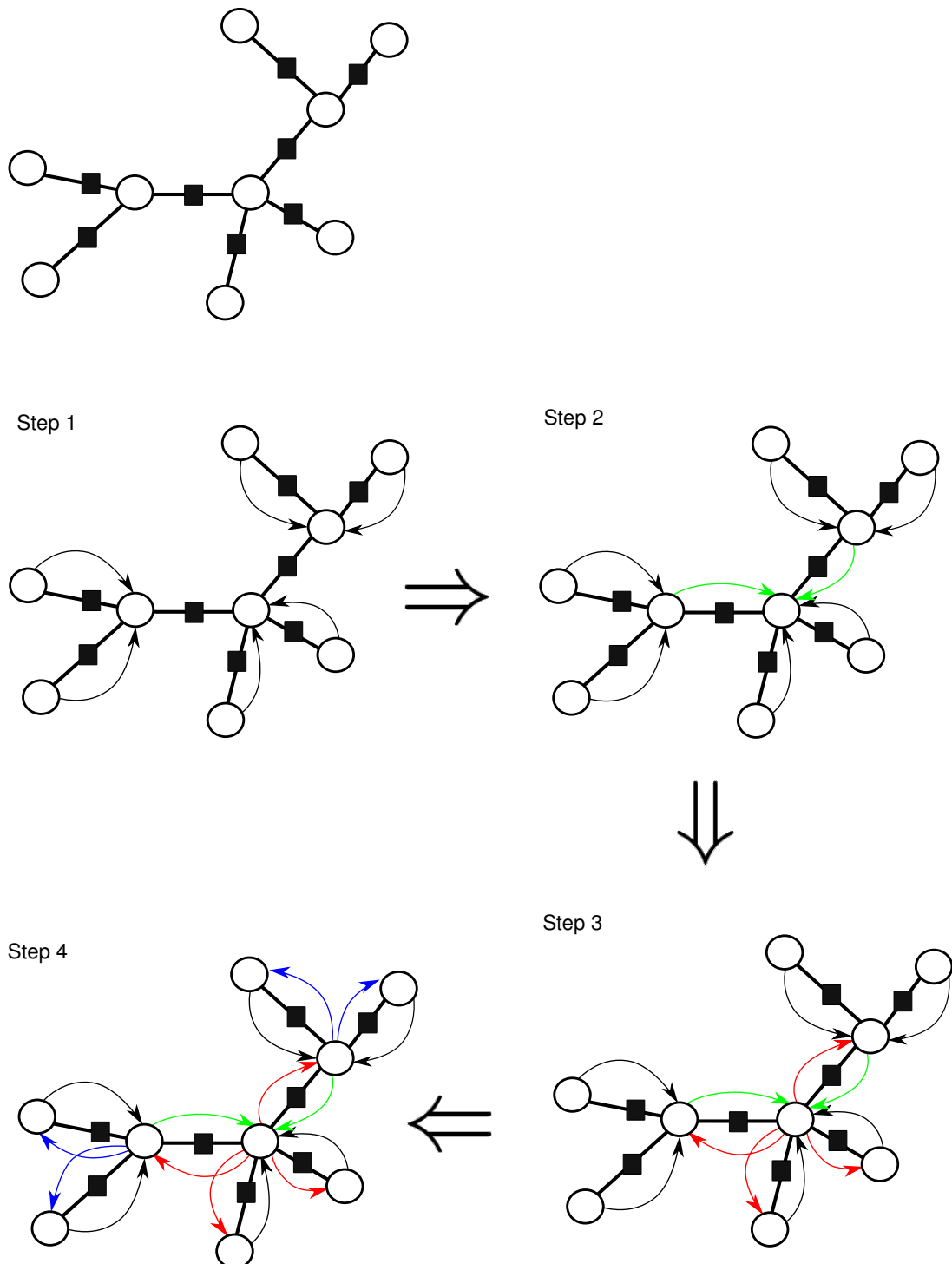


Figure 2.6 Steps involved for computing the messages of the politree represented at the top. The leaves are the first nodes for which the outgoing messages can be computed.

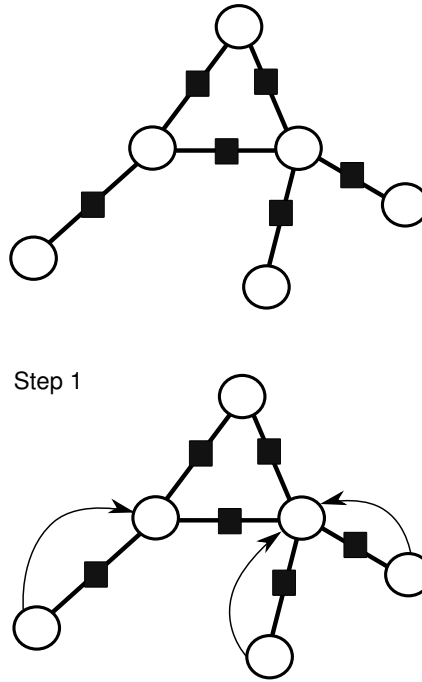


Figure 2.7 Steps involved for computing the messages on a loopy graph: after computing the messages outgoing from the leaves, a deadlock is reached since no further messages are computable.

2.3 Maximum a posteriori estimation

Suppose we are not interested in determining the marginal probability of a specific variable, but rather we want the combination in the hidden set \mathcal{H} that maximises the probability $\mathbb{P}(H_1, \dots, H_n | O)$. Clearly, we could try to compute the entire distribution $\mathbb{P}(H_1, \dots, H_n | O)$ and then take the value of H maximising that distribution. However, this is not computationally possible since even for low medium size graphs the size of $Dom(\cup_{H_i \in \mathcal{H}} H_i)$ can be huge.

Maximum a posteriori estimations solve this problem: the value maximising $\mathbb{P}(H_1, \dots, H_n | O)$ is computed, without explicitly building the entire distribution $\mathbb{P}(H_1, \dots, H_n | O)$. This is achieved by performing belief propagation with a slightly different version of the message passing algorithm presented in Section 2.2.2. Referring to Figure 2.4, the message to A is computed as follows when performing a maximum a posteriori estimation:

$$M_{B \rightarrow A}(a) = \max_b \{ \Phi_{AB}(a, \tilde{b}) \prod_{i=1}^m M_{V_i \rightarrow B}(\tilde{b}) \} \quad (2.29)$$

Essentially, the summation in equation (2.27) is replaced with the max operator. After all messages are computed, the estimation $h_{MAP} = \{h_{1MAP}, h_{2MAP}, \dots\}$ is obtained by considering for every variable in \mathcal{H} the value maximising:

$$h_{iMAP} = \operatorname{argmax} \{ \Phi_{H_i}(h_{iMAP}) \prod_{k=1}^L M_k(h_{iMAP}) \} \quad (2.30)$$

where M_1, \dots, M_L refer to all the messages incoming to H_i . To be precise, this procedure is not guaranteed to return the value actually maximising $\mathbb{P}(H_1, \dots, H_n | O)$, but at least a strong local maximum is obtained.

At this point it is worthy to clarify that the combination $h_{MAP} = \{h_{1MAP}, h_{2MAP}, \dots\}$ could not be obtained by simply assuming for every H_i the realization maximising the marginal distribution:

$$h_{MAP} \neq \{ \operatorname{argmax}(\mathbb{P}(h_1)), \dots, \operatorname{argmax}(\mathbb{P}(h_n)) \} \quad (2.31)$$

This is due to the fact that $\mathbb{P}(H_1, \dots, H_n | O)$ is a joint probability distribution, while the marginals $\mathbb{P}(H_i)$ are not. For better understanding this aspect consider the graph reported in Figure 2.8, with the potentials Φ_{XA} , Φ_{AB} and Φ_{YB} having the images defined in table 2.5. Suppose discovering that $X = 0$ and $Y = 1$. Then, performing the standard message passing algorithm explained in the previous Section we obtain the messages reported in Figure

	b_0	b_1		x_0	x_1		y_0	y_1
a_0	2	0	a_0	1	0.1	b_0	1	0.1
a_1	0	2	a_1	0.1	1	b_1	0.1	1

Table 2.5 Factors involved in the graph of Figure 2.8.

A	B	$E(A, B, X = 0, Y = 1)$
0	0	0.2
0	1	0
1	0	0
1	1	0.2

Table 2.6 Factors involved in the graph of Figure 2.8.

2.8. Clearly individual marginals for A and B would be equal to:

$$\begin{aligned}\mathbb{P}(A) &= \begin{pmatrix} \mathbb{P}(A = 0) \\ \mathbb{P}(A = 1) \end{pmatrix} = \begin{pmatrix} 0.5 \\ 0.5 \end{pmatrix} \\ \mathbb{P}(B) &= \begin{pmatrix} \mathbb{P}(B = 0) \\ \mathbb{P}(B = 1) \end{pmatrix} = \begin{pmatrix} 0.5 \\ 0.5 \end{pmatrix}\end{aligned}\quad (2.32)$$

Therefore, all the combinations $\{A = 0, B = 0\}$, $\{A = 0, B = 1\}$, $\{A = 1, B = 0\}$, $\{A = 1, B = 1\}$ maximise $\mathbb{P}(A, B|O)$. However, it easy to prove that $E(A, B, X, Y)$ assumes the values reported in table 2.6. Therefore, the combinations actually maximising the joint distribution $\mathbb{P}(A, B|O)$ are $\{A = 0, B = 0\}$ and $\{A = 1, B = 1\}$, leading to a different result.

2.4 Gibbs sampling

Gibbs sampling is a Monte Carlo method for obtaining samples from a joint distribution of variables X_1, \dots, m , without explicitly compute that distribution. Indeed, Gibbs sampling is an iterative method which requires every time to determine the conditional distribution of a single variable X_i w.r.t to all the others in the group.

More formally the algorithm starts with an initial combination of values $\{x_1, \dots, m\}$ for the variable $\cup_{i=\{1, \dots, m\}} X_i$. At every iteration, all the values of that combination are recomputed. At the j^{th} iteration the value of x_k^{j+1} for the subsequent iteration is obtaining by sampling from the following marginal distribution:

$$x_k^{j+1} \sim \mathbb{P}(x_k | x_{\{1, \dots, m\} \setminus k}^j) \quad (2.33)$$

After an initial transient, the samples cumulated during the iterations can be considered as drawn from the joint distribution involving group X_1, \dots, m .

This algorithm can be easily applied to graphical model. Indeed the methodologies exposed in Section 2.2.2 can be applied for determining the conditional distribution of a single variable $H_i \in \mathcal{H}$ w.r.t all the others (as well the evidences in \mathcal{O}), assuming all variables in $\mathcal{H} \setminus H_i$ as additional observations and computing the marginal probability of H_i .

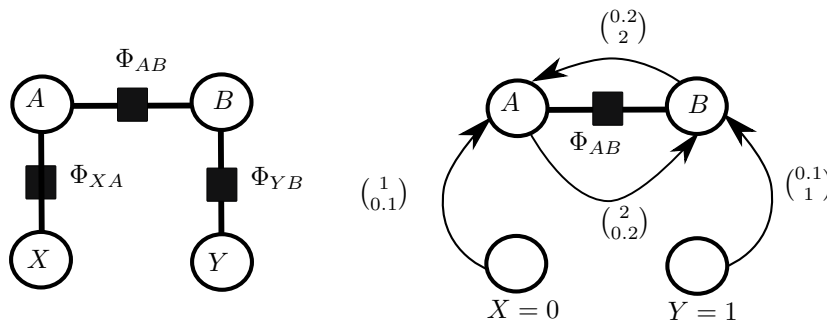


Figure 2.8 Example of graph adopted. When the evidences are retrieved, the messages computed by making use of the message passing algorithm are reported below.

2.5 Sub graphs

As explained in Section 2.2.2, the marginal probability of a variable $H_i \in \mathcal{H}$ can be efficiently computed by considering the messages produced by the message passing algorithm. The same messages can be also used for performing graph reduction, with the aim to model the joint probability distribution of a subset of variables $\{H_1, H_2, H_3\} \subset \mathcal{H}$, i.e. $\mathbb{P}(H_{1,2,3}|O)$. The latter quantity is the marginal probability of the subset of variables of interest.

The aim of message passing is essentially to simplify the graph, condensing all the belief information into the messages. Such property is exploited for computing sub graphs. Without loss of generality assume from now on $\mathcal{O} = \emptyset$. Consider the graph in Figure 2.9 and suppose we are interested in modelling $\mathbb{P}(A, B, C)$, no matter the values of the other variables. After computing all the messages exploiting message passing, the sub graph reported in Figure 2.9 is the one modelling $\mathbb{P}(A, B, C)$. Actually, that sub graph is a graphical model itself, for which all the properties exposed so far hold. For example the energy function E is computable as follows:

$$E(A = a, B = b, C = c) = \Phi_{AB}(a, b)\Phi_{BC}(b, c)\Phi_{AC}(a, c)M_{X \rightarrow A}(a)M_{Y \rightarrow B}(b) \quad (2.34)$$

while the joint probability of A, B and C can be computed in this way:

$$\mathbb{P}(A = a, B = b, C = c) = \frac{E(a, b, c)}{\sum_{\forall \tilde{a}, \tilde{b}, \tilde{c}} E(\tilde{a}, \tilde{b}, \tilde{c})} \quad (2.35)$$

Notice that in this case the graph is significantly smaller than the originating one, implying that the above computations can be performed in an acceptable time.

Also Gibbs sampling can be applied to a reduced graph, producing samples drawn from the marginal probability $\mathbb{P}(A, B, C)$.

The reduction described so far is always possible when considering a subset of variables forming a connected sub-portion of the original graph, i.e. after reduction there must be a unique sub structure. For instance, variables X and Y of the graph in Figure 2.10 do not respect the latter specification, meaning that it is not possible to build a sub graph involving X and Y .

2.6 Learning

The aim of learning is to determine the optimal values for the w (equation (2.14)) of all the tunable potentials (see Section 2.1) Ψ . To this aim two cases must be distinguished:

- Learning must be performed for a RandomField: see Section 2.6.1
- Learning must be performed for a ConditionalRandomField: see Section 2.6.2

No matter the case, the population of tunable weights, i.e. the weights of the tunable Exponential Factors of the model, will be indicated with W :

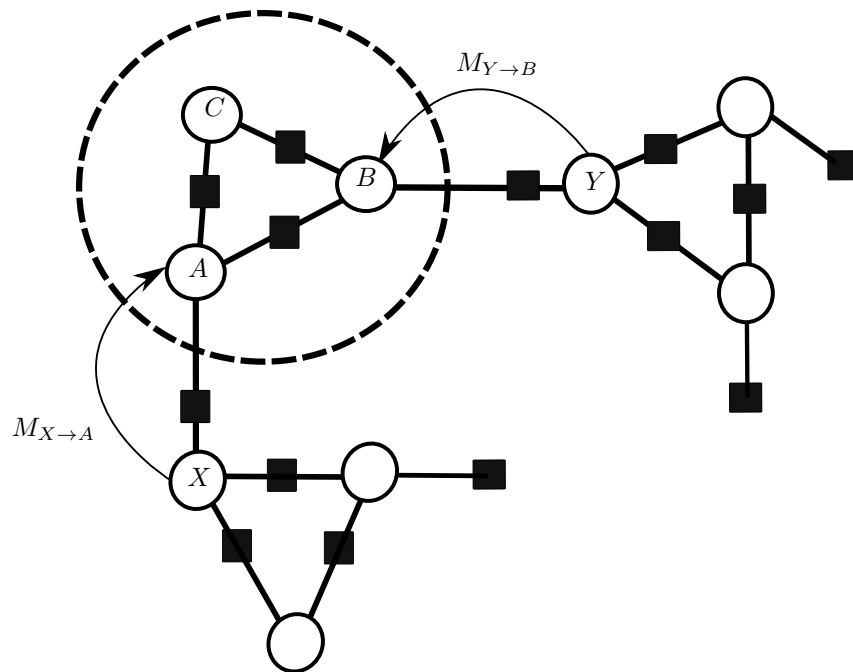
$$W = \{w_1, \dots, w_D\} \quad (2.36)$$

w_i will refer to the i^{th} free parameter of the model. For the purpose of learning, we assume $\mathcal{O} = \emptyset$. Learning considers a training set $T = \{t_1, \dots, t_N\}$ made of realizations of the joint distribution correlating all the variables in \mathcal{V} , no matter the fact that they are involved in tunable or non tunable potentials. As exposed in Section 2.1, if W is known, the probability of a combination t_j can be evaluated as follows:

$$\mathbb{P}(t_j) = \frac{E(t_j, W)}{\mathcal{Z}(W)} \quad (2.37)$$

At this point we can observe that the energy function is the product of two main factors: one depending from t_j and W and the other depending only upon t_j representing the contribution of all the non tunable potentials (Factors and Exponential Factors, see Section 2.1):

$$\begin{aligned} E(t_j, W) &= \exp(w_1 \Phi_1(t_j)) \cdots \exp(w_D \Phi_D(t_j)) \cdot E_0(t_j) \\ &= \exp\left(\sum_{i=1}^D w_i \Phi_i(t_j)\right) \cdot E_0(t_j) \end{aligned} \quad (2.38)$$



Sub graph involving A, B, C

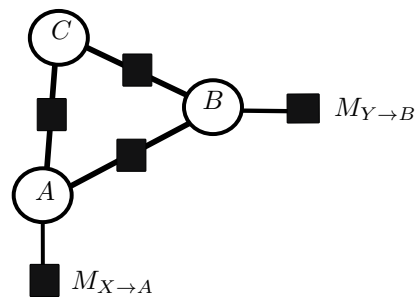


Figure 2.9 Example of graph reduction.

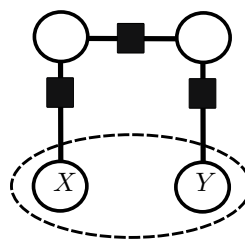


Figure 2.10 Example of a subset of variables for which the graph reduction is not possible.

The likelihood function L can be defined as follows:

$$L = \prod_{t_j \in T} \mathbb{P}(t_j) \quad (2.39)$$

passing to the logarithmic likelihood and dividing by the training set size N we obtain:

$$\begin{aligned} J = \frac{\log(L)}{N} &= \sum_{t_j \in T} \frac{\log(\mathbb{P}(t_j))}{N} \\ &= \sum_{t_j \in T} \frac{\log(E(t_j, W)) - \log(\mathcal{Z}(W))}{N} \\ &= \frac{1}{N} \sum_{t_j \in T} \log(E(t_j, W)) - \log(\mathcal{Z}(W)) \\ &= \frac{1}{N} \sum_{t_j \in T} \left(\sum_{i=1}^D w_i \Phi_i(t_j) \right) - \log(\mathcal{Z}(W)) + \dots \\ &+ \frac{1}{N} \sum_{t_j \in T} \log(E_0(t_j)) \end{aligned} \quad (2.40)$$

The aim of learning become essentially to find the value of W maximising J . This is typically done iteratively, by searching at every iteration the optimum along a direction similar to the one given by the gradient $\frac{\partial J}{\partial W}$. The way the gradient is exploited to update the searching direction characterize the kind of aproach.

The basic gradient descend approach assumes the searching direction equal to the gradient, while Quasi Newton methods (<https://www.jstor.org/stable/2006193>) use an estimation of the hessian (computed using only the gradient variation) in order to correct the direction given by the gradient. Non linear conjugate approach (<https://www.caam.rice.edu/~yzhang/caam554/pdf/cgsurvey.pdf>) implements memory-based approaches that compute the new searching direction by considering both the actual value of the gradient as well as the old used direction.

The above methods are already implemented inside this library.

2.6.1 Learning of unconditioned model

The computations to perform for evaluating the gradient $\frac{\partial J}{\partial W}$ in case of unconditioned model will be exposed in this Section. Notice that in equation (2.40), term $\sum_{t_j \in T} \log(E_0(t_j))$ is constant and consequently will be not considered for computing the gradient of J . Equation (2.40) can be rewritten as follows:

$$\begin{aligned} J &= \alpha(T, W) - \beta(W) \\ \alpha &= \frac{1}{N} \sum_{t_j \in T} \left(\sum_{i=1}^D w_i \Phi_i(t_j) \right) \end{aligned} \quad (2.41)$$

$$\beta = \log(\mathcal{Z}(W)) \quad (2.42)$$

α is influenced by T , while the same is not valid for β .

2.6.1.1 Gradient of α

By the analysis of the equation (2.41) it is clear that:

$$\frac{\partial \alpha}{\partial w_i} = \frac{1}{N} \sum_{t_j \in T} \Phi_i(t_j) \quad (2.43)$$

2.6.1.2 Gradient of β

The computation of $\frac{\partial \beta}{\partial w_i}$ requires to manipulate a little bit equation (2.42). Firstly the derivative of the logarithm must be computed:

$$\frac{\partial \beta}{\partial w_i} = \frac{1}{\mathcal{Z}} \frac{\partial \mathcal{Z}}{\partial w_i} \quad (2.44)$$

The normalizing coefficient \mathcal{Z} is made of the following terms (see also equation (2.7)):

$$\mathcal{Z}(W) = \sum_{\tilde{V} \in \bigcup_{i=1}^p V_i} \left(\exp\left(\sum_{i=1}^D w_i \Phi_i(\tilde{V})\right) \cdot E_0(\tilde{V}) \right) \quad (2.45)$$

Introducing equation (2.45) into (2.44) leads to:

$$\begin{aligned} \frac{\partial \beta}{\partial w_i} &= \frac{1}{\mathcal{Z}} \frac{\partial}{\partial w_i} \left(\sum_{\tilde{V}} \exp\left(\sum_{i=1}^D w_i \Phi_i(\tilde{V})\right) E_0(\tilde{V}) \right) \\ &= \frac{1}{\mathcal{Z}} \sum_{\tilde{V}} \frac{\partial}{\partial w_i} \left(\exp\left(\sum_{i=1}^D w_i \Phi_i(\tilde{V})\right) \right) E_0(\tilde{V}) \\ &= \frac{1}{\mathcal{Z}} \sum_{\tilde{V}} \exp\left(\sum_{i=1}^D w_i \Phi_i(\tilde{V})\right) E_0(\tilde{V}) \Phi_i(\tilde{V}) \\ &= \sum_{\tilde{V}} \frac{\exp\left(\sum_{i=1}^D w_i \Phi_i(\tilde{V})\right) E_0(\tilde{V})}{\mathcal{Z}} \Phi_i(\tilde{V}) \\ &= \sum_{\tilde{V}} \frac{E(\tilde{V})}{\mathcal{Z}} \Phi_i(\tilde{V}) \\ &= \sum_{\tilde{V}} \mathbb{P}(\tilde{V}) \Phi_i(\tilde{V}) \end{aligned} \quad (2.46)$$

Last term in the above equations can be further elaborated. Assume that the variables involved in potential Φ_j are $V_{1,2}$, then:

$$\begin{aligned} \frac{\partial \beta}{\partial w_i} &= \sum_{\tilde{V}} \mathbb{P}(\tilde{V}) \Phi_i(\tilde{V}) \\ &= \sum_{\tilde{V}_{1,2}} \Phi_i(\tilde{V}_{1,2}) \sum_{\tilde{V}_{3,4,\dots}} \mathbb{P}(\tilde{V}_{1,2,3,4,\dots}) \\ &= \sum_{\tilde{V}_{1,2}} \Phi_i(\tilde{V}_{1,2}) \mathbb{P}(\tilde{V}_{1,2}) \end{aligned} \quad (2.47)$$

where $\mathbb{P}(\tilde{V}_{1,2})$ is the marginal probability (see the initial part of Section 2.1) of the variables involved in the potential Φ_i , which can be easily computable by considering the sub graph containing only V_1 and V_2 as variables (see Section 2.5). Notice that in case Φ_i is a unary potential the same holds, considering the marginal distribution of the single variable involved by Φ_i :

$$\frac{\partial \beta}{\partial w_i} = \sum_{\forall \tilde{V}_1} \Phi_i(\tilde{V}_1) \mathbb{P}(\tilde{V}_1) \quad (2.48)$$

which can be easily obtained through the message passing algorithm (Section 2.2.2).

After all the manipulations performed, the gradient $\frac{\partial J}{\partial w_i}$ has the following compact expression:

$$\frac{\partial J}{\partial w_i} = \frac{1}{N} \sum_{j=1}^N \Phi_i(D_j^i) - \sum_{\tilde{D}^i} \mathbb{P}(\tilde{D}^i) \Phi_i(\tilde{D}^i) \quad (2.49)$$

2.6.2 Learning of conditioned model

For such models learning is more demanding as will be exposed. Recalling the definition provided in the final part of Section 2.1, Conditional Random Fields are graphs for which the set of observations \mathcal{O} is fixed. The training set T is made of realizations of both \mathcal{H} and \mathcal{O} :

$$\begin{aligned} T &= \{t_1, \dots, t_N\} \\ &= \{\{h_1, o_1\}, \dots, \{h_N, o_N\}\} \end{aligned} \quad (2.50)$$

We recall, equation (2.18), that the conditional probability of the hidden variables w.r.t. the observed ones is defined as follows:

$$\begin{aligned} \mathbb{P}(h_j, o_j) &= \frac{E(h_j, o_j, W)}{\mathcal{Z}(o_j, W)} \\ E(h_j, o_j, W) &= \exp\left(\sum_{i=1}^D w_i \Phi_i(h_j, o_j)\right) E_0(h_j, o_j) \\ \mathcal{Z}(o_j, W) &= \sum_{\tilde{h}} E(\tilde{h}, o_j, W) \end{aligned} \quad (2.51)$$

The aim of learning is to maximise a likelihood uncton L defined in this case as follows:

$$L = \prod_{h_j \in T} \mathbb{P}(h_j | o_j) \quad (2.52)$$

Passing to the logarithms and dividing by the training set size we obtain the following objective function J :

$$\begin{aligned} J &= \frac{\log(L)}{N} \\ &= \frac{1}{N} \sum_{h_j, o_j \in T} \log(E(h_j, o_j, W)) - \frac{1}{N} \sum_{h_j, o_j \in T} \log(\mathcal{Z}(o_j, W)) \\ &= \frac{1}{N} \sum_{h_j, o_j \in T} \left(\sum_{i=1}^D w_i \Phi_i(h_j, o_j) \right) - \frac{1}{N} \sum_{h_j, o_j \in T} \log(\mathcal{Z}(o_j, W)) \\ &\quad + \frac{1}{N} \sum_{h_j, o_j \in T} \log(E_0(h_j, o_j)) \end{aligned} \quad (2.53)$$

Neglecting E_0 which not depends upon W , equation (2.53) can be rewritten as follows:

$$\begin{aligned} J &= \alpha(T, W) - \beta(T, W) \\ \alpha(T, W) &= \frac{1}{N} \sum_{h_j, o_j} \left(\sum_{i=1}^D w_i \Phi_i(h_j, o_j) \right) \\ \beta(T, W) &= \frac{1}{N} \sum_{o_j} \log(\mathcal{Z}(o_j, W)) \end{aligned} \quad (2.54)$$

At this point, an important remark must be done: differently from the β defined in equation (2.42), $\beta(T, W)$ of conditioned model is a function of the training set. The latter observation has an important consequence: when performing learning of unconditioned model, belief propagation (i.e. the computation of the messages through message passing with the aim of computing the marginal probabilities of the groups of variables involved in the factor of the model) must be performed once for every iteration of the gradient descend; on the opposite when considering conditioned model, belief propagation must be performed at every iteration for every element of the training set, see equation (2.58). This makes the learning of conditioned models much more computationally demanding. This price is paid in order to not model the correlation among the observations³, which can be interesting for many applications. The computation of $\frac{\partial \alpha}{\partial w_i}$ is analogous to the one of non conditioned model, equation (2.43).

³that can be highly correlated

2.6.2.1 Gradient of β

Following the same approach in Section 2.6.1.2, the gradient of β can be computed as follows:

$$\begin{aligned}
\frac{\partial \beta}{\partial w_i} &= \frac{1}{N} \sum_{j=1}^N \frac{\partial \log(\mathcal{Z}(o_j, W))}{\partial w_i} \\
&= \frac{1}{N} \sum_{j=1}^N \frac{1}{\mathcal{Z}(o_j)} \frac{\partial \mathcal{Z}(o_j, W)}{\partial w_i} \\
&= \frac{1}{N} \sum_{j=1}^N \frac{\partial}{\partial w_i} \left(\sum_{\tilde{h}} \exp\left(\sum_{i=1}^D w_i \Phi_i(\tilde{h}, o_j)\right) E_0(\tilde{h}, o_j) \right) \\
&= \frac{1}{N} \sum_{j=1}^N \sum_{\tilde{h}} \left(\exp\left(\sum_{i=1}^D w_i \Phi_i(\tilde{h}, o_j)\right) E_0(\tilde{h}, o_j) \Phi_i(\tilde{h}, o_j) \right) \\
&= \frac{1}{N} \sum_{j=1}^N \sum_{\tilde{h}} \frac{E(\tilde{h}, o_j, W)}{\mathcal{Z}(o_j)} \Phi_i(\tilde{h}, o_j) \\
&= \frac{1}{N} \sum_{j=1}^N \sum_{\tilde{h}} \mathbb{P}(\tilde{h}|o_j) \Phi_i(\tilde{h}, o_j)
\end{aligned} \tag{2.55}$$

Suppose the variables involved in the factor Φ_j are $\tilde{h}_{1,2}$, then:

$$\begin{aligned}
\frac{\partial \beta}{\partial w_i} &= \frac{1}{N} \sum_{j=1}^N \sum_{\tilde{h}} \mathbb{P}(\tilde{h}|o_j) \Phi_i(\tilde{h}, o_j) \\
&= \frac{1}{N} \sum_{j=1}^N \sum_{\tilde{h}_{1,2}} \Phi_i(\tilde{h}_{1,2}) \sum_{\tilde{h}_{3,4,\dots}} \mathbb{P}(\tilde{h}_{1,2,3,4,\dots}|o_j) \\
&= \frac{1}{N} \sum_{j=1}^N \sum_{\tilde{h}_{1,2}} \Phi_i(\tilde{h}_{1,2}) \mathbb{P}(\tilde{h}_{1,2}|o_j)
\end{aligned} \tag{2.56}$$

where $\mathbb{P}(\tilde{h}_{1,2}|o_j)$ is the conditioned marginal probability of group $\tilde{h}_{1,2}$ w.r.t. the observations o_j .

Grouping all the simplifications we obtain:

$$\frac{\partial J}{\partial w_i} = \frac{1}{N} \sum_{j=1}^N \Phi_i(h_j, o_j) - \frac{1}{N} \sum_{j=1}^N \left(\sum_{\tilde{h}_{1,2}} \mathbb{P}(\tilde{h}_{1,2}|o_j) \Phi_i(\tilde{h}_{1,2}) \right) \tag{2.57}$$

Generalizing:

$$\frac{\partial J}{\partial w_i} = \frac{1}{N} \sum_{j=1}^N \Phi_i(D_j^i, o_j) - \frac{1}{N} \sum_{j=1}^N \left(\sum_{\tilde{D}^i} \mathbb{P}(\tilde{D}^i|o_j) \Phi_i(\tilde{D}^i, o_j) \right) \tag{2.58}$$

2.6.3 Learning of modular structure

Suppose to have a modular structure made of repeating units as for example the graph in Figure 2.11. Every single unit has the same population of potentials and we would like to enforce this fact when performing learning. In particular we'll have some sets of Exponential shape sharing the same weight w_1 (see Figure 2.11). Motivated by this example, we included in the library the possibility to specify that a potential must share its weight with another one. Then, learning is done consistently with the aforementioned specification.

2.6.3.1 Gradient of α

Considering the model in Figure 2.11, the α part of J (equation (2.41)) can be computed as follows:

$$\alpha = \frac{1}{N} \sum_{t_j} (w_1 \Phi_1(a_{1j}, b_{1j}) + w_1 \Phi_2(a_{2j}, b_{2j}) + w_1 \Phi_3(a_{3j}, b_{3j}) + \dots + \dots + \sum_{i=2}^D w_i \Phi_i(t_j)) \quad (2.59)$$

which leads to:

$$\frac{\partial \alpha}{\partial w_1} = \frac{1}{N} \sum_{t_j} (\Phi_1(a_{1j}, b_{1j}) + \Phi_2(a_{2j}, b_{2j}) + \Phi_3(a_{3j}, b_{3j})) \quad (2.60)$$

2.6.3.2 Gradient of β

Regarding the β part of J we can write what follows:

$$\begin{aligned} \frac{\partial \beta}{\partial w_1} &= \frac{1}{Z} \frac{\partial Z}{\partial w_1} \\ &= \frac{1}{Z} \frac{\partial}{\partial w_1} \left(\sum_{\tilde{V}} \left(\exp(w_1(\Psi_1(a_{1j}, b_{1j}) + \dots \right. \right. \\ &\quad \left. \left. \dots + \Psi_2(a_{2j}, b_{2j}) + \Psi_3(a_{3j}, b_{3j})) + \sum_{i=2}^D w_i \Phi_i(\tilde{V})) E_0(\tilde{V})) \right) \right) \\ &= \sum_{\tilde{V}} \mathbb{P}(\tilde{V}) (\Phi_1(\tilde{a}_1, \tilde{b}_1) + \Phi_2(\tilde{a}_2, \tilde{b}_2) + \Phi_3(\tilde{a}_3, \tilde{b}_3)) \\ &= \sum_{\tilde{V}} \mathbb{P}(\tilde{V}) \Phi_1(\tilde{a}_1, \tilde{b}_1) + \sum_{\tilde{V}} \mathbb{P}(\tilde{V}) \Phi_2(\tilde{a}_2, \tilde{b}_2) + \sum_{\tilde{V}} \mathbb{P}(\tilde{V}) \Phi_3(\tilde{a}_3, \tilde{b}_3) \\ &= \sum_{\tilde{A}_1, \tilde{B}_1} \mathbb{P}(\tilde{A}_1, \tilde{B}_1) \Phi_1(\tilde{A}_1, \tilde{B}_1) + \sum_{\tilde{A}_2, \tilde{B}_2} \mathbb{P}(\tilde{A}_2, \tilde{B}_2) \Phi_2(\tilde{A}_2, \tilde{B}_2) + \dots \\ &\quad \dots + \sum_{\tilde{A}_3, \tilde{B}_3} \mathbb{P}(\tilde{A}_3, \tilde{B}_3) \Phi_3(\tilde{A}_3, \tilde{B}_3) \end{aligned} \quad (2.61)$$

Notice that the gradient $\frac{\partial J}{\partial w_1}$ is the summation of three terms: the ones that would have been obtained considering separately the three potentials in which w_1 is involved (equation (2.49)):

$$\begin{aligned} \frac{\partial J}{\partial w_1} &= \frac{1}{N} \sum_{j=1}^N \Phi_1(a_i^1, b_i^1) - \sum_{\tilde{a}^1, \tilde{b}^1} \mathbb{P}(\tilde{a}^1, \tilde{b}^1) \Phi_1(\tilde{a}^1, \tilde{b}^1) + \dots \\ &\quad + \frac{1}{N} \sum_{j=1}^N \Phi_2(a_i^2, b_i^2) - \sum_{\tilde{a}^2, \tilde{b}^2} \mathbb{P}(\tilde{a}^2, \tilde{b}^2) \Phi_2(\tilde{a}^2, \tilde{b}^2) + \dots \\ &\quad + \frac{1}{N} \sum_{j=1}^N \Phi_3(a_i^3, b_i^3) - \sum_{\tilde{a}^3, \tilde{b}^3} \mathbb{P}(\tilde{a}^3, \tilde{b}^3) \Phi_3(\tilde{a}^3, \tilde{b}^3) + \dots \end{aligned} \quad (2.62)$$

The above result has a general validity, also considering conditioned graphs.

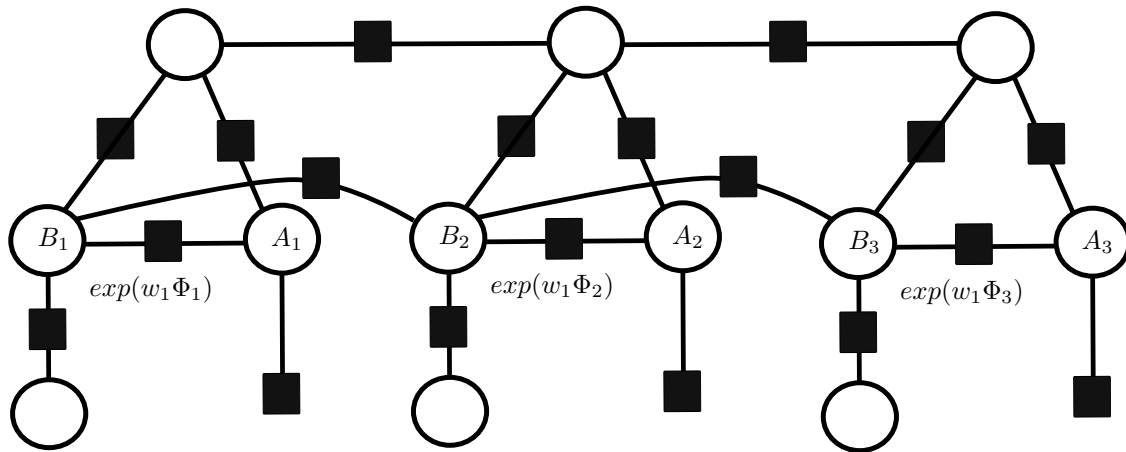


Figure 2.11 Example of modular structure: weight w_1 is simultaneously involved into potentials Φ_1, Φ_2 and Φ_3 .

Chapter 3

Import models from xml files

The aim of this Section is to expose how to build graphical models from XML files describing their structures. In particular, the syntax of such an XML will be clarified. Figure 3.1 visually explains the structure of a valid XML. Essentially two kind of tags must be incorporated:

- **Variable:** describes the information related to a variable present in the graph. There must a tag of this kind for every variable constituting the model. Fields description:
 - **name:** is a string indicating the name of this variable.
 - **Size:** is the size of the variable, i.e. the size of Dom , see Section 2.1.
 - **flag[optional] :** is a flag that can assume two possible values, 'O' or 'H' according to the fact that this variable is in set \mathcal{O} (Section 2.1) or not respectively. When non specifying this flag 'H' is assumed.
 - **Potential:** describes the information related to a unary or a binary potential present in the graph (see Section 2.1). Fields description:
 - **var:** the name of the first variable involved.
 - **var[optional]:** the name of the second variable involved. Is omitted when considering unary potentials, while is mandatory when a binary potentials is described by this tag.
 - **weight[optional]:** when specifying an Exponential Factors (Section 2.1) it must be present for indicating the value of the weight w (equation (2.14)). When omitting, the potential is assumed to be a simple Factor.
 - **tunability[optional]:** it is a flag for specifying whether the weight of this Exponential Factor is tunable or not (see Section 2.1). Is ignored in case weight is omitted. It can assumes two possible values, 'Y' or 'N' according to the fact that the weight involved is tunable or not respectively. When weight is specified and tunability is omitted, a value equal to 'Y' is assumed.
 - **Share[optional]:** you must specify this sub tag when the containing Exponential Factor shares its weight with another potential in the model. Sub fields var are exploited for specifying the variables involved by the potential whose weight is to share. If weight is omitted in the containing Potential tag, this sub tag is ignored, even though the value assigned to weight is ignored since it is shared with another potential. The potential sharing its weight must be clearly an Exponential shape, otherwise the sharing directive is ignored.
- The following components are exclusive: only one of them can be specified in a Potential tag and at the same time at least one must be present.
- **Correlation:** it can assume two possible values, 'T' or 'F'. When 'T' is passed, this potential is assumed to be a correlating potential (see sample 4.1.2.2), otherwise when passing 'F' a simple anti correlating factor is assumed. It is invalid in case this Potential is a unary one. In case weight was specified, an Exponential Factor is built, passing as input the correlating or anti-correlating Factor described by this tag.

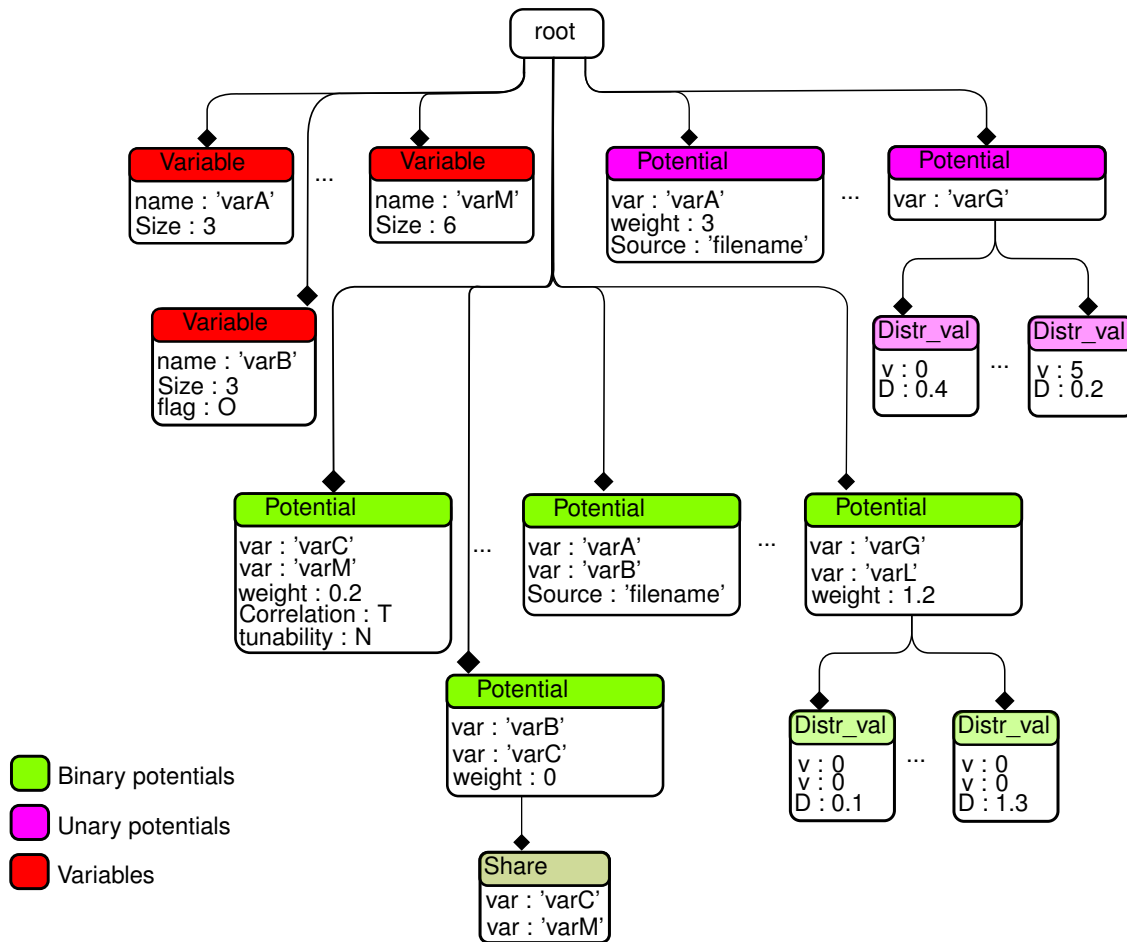


Figure 3.1 The structure of the XML describing a graphical model.

- Source: it is the location of a textual file describing the values of the distribution characterizing this potential. Rows of this file contain the values characterizing the image of the potential. Combinations not specified are assumed to have an image value equal to 0. Clearly the number of values characterizing the distribution must be consistent with the number of specified var fields. In case weight was specified, an Exponential Factor is built, starting from the Factor described by the values specified in the aforementioned file. For instance, the potential Φ_b of Section 2.1 would have been described by a file containing the following rows:

```

0 0 1
0 1 4
1 1 1
2 2 5
2 4 1

```

(3.1)

- Set of sub tags Distr_val: is a set of nested tags describing the distribution of the this potential. Similarly to Source, every element use fields v for describing the combination, while D is used for specifying the value assumed by the distribution. For example the potential Φ_b of Section 2.1 would have been described by the syntax reported in Figure 3.2. In case weight was specified, an Exponential Factor is built, starting from the Factor whose distribution is specified by the aforementioned sub tags.

```
<Potential . . . >
  <Distr_val "v"=0 "v"=0 "D"=1>
  <Distr_val "v"=0 "v"=1 "D"=4>
  <Distr_val "v"=1 "v"=1 "D"=1>
  <Distr_val "v"=2 "v"=2 "D"=5>
  <Distr_val "v"=2 "v"=4 "D"=1>
</Potential>
```

Figure 3.2 Syntax to adopt for describing the potential Φ_b of Section 2.1, using a population of `Distr_val` sub tags.

Chapter 4

Samples

4.1 Sample 01: Potential handling

The aim of this series of examples is mainly to show how to handle the creation of variables and factors.

4.1.1 Variables creation

4.1.1.1 part 01

This example creates a tow initial variables A and B with a domain size equal to 2 and places them into a group storing them. Later, it adds variables C and D , with the same domain size. Finally, tries to add again C , showing the this further addition is correctly refused.

4.1.1.2 part 02

This example considers 3 variables A , B and C with a domain sizes equal to, respectively, 2,4 and 3, i.e:

$$\begin{aligned} Dom(A) &= \{a_1 = 0, a_2 = 1\} \\ Dom(B) &= \{b_1 = 0, b_2 = 1, b_3 = 2, b_4 = 3\} \\ Dom(C) &= \{c_1 = 0, c_2 = 1, c_3 = 2\} \end{aligned} \tag{4.1}$$

Then, evaluates the joint domain of: $A \cup B$, $A \cup C$ and $A \cup B \cup C$, which should results in the following combinations reported in, respectively, tables [4.1](#), [4.2](#) and [4.3](#).

4.1.2 Factors creation

4.1.2.1 part 01

Part 01 creates a shape factor Φ_{AB} , involving the pair of variables A and B . Both that variables have a domain size equal to 4, i.e. $Dom(A) = \{a_0 = 0, a_1 = 1, a_2 = 2, a_3 = 3\}$ and $Dom(B) = \{b_0 = 0, b_1 = 1, b_2 = 2, b_3 = 3\}$. The generic value in the image Φ_{AB} is equal to:

$$\Phi_{AB}(A = a, B = b) = a + 2 \cdot b \tag{4.2}$$

Table [4.4](#) reports the entire image of Φ_{AB} .

$Dom(AB) = Dom(A \cup B)$
$\{a_1, b_1\}$
$\{a_1, b_2\}$
$\{a_1, b_3\}$
$\{a_1, b_4\}$
$\{a_2, b_1\}$
$\{a_2, b_2\}$
$\{a_2, b_3\}$
$\{a_2, b_4\}$

Table 4.1 Domain of AB .

$Dom(AC) = Dom(A \cup C)$
$\{a_1, c_1\}$
$\{a_1, c_2\}$
$\{a_1, c_3\}$
$\{a_2, c_1\}$
$\{a_2, c_2\}$
$\{a_2, c_3\}$

Table 4.2 Domain of AC .

$Dom(ABC) = Dom(A \cup B \cup C)$
$\{a_1, b_1, c_1\}$
$\{a_1, b_1, c_2\}$
$\{a_1, b_1, c_3\}$
$\{a_1, b_2, c_1\}$
$\{a_1, b_2, c_2\}$
$\{a_1, b_2, c_3\}$
$\{a_1, b_3, c_1\}$
$\{a_1, b_3, c_2\}$
$\{a_1, b_3, c_3\}$
$\{a_1, b_4, c_1\}$
$\{a_1, b_4, c_2\}$
$\{a_1, b_4, c_3\}$
$\{a_2, b_1, c_1\}$
$\{a_2, b_1, c_2\}$
$\{a_2, b_1, c_3\}$
$\{a_2, b_2, c_1\}$
$\{a_2, b_2, c_2\}$
$\{a_2, b_2, c_3\}$
$\{a_2, b_3, c_1\}$
$\{a_2, b_3, c_2\}$
$\{a_2, b_3, c_3\}$
$\{a_2, b_4, c_1\}$
$\{a_2, b_4, c_2\}$
$\{a_2, b_4, c_3\}$

Table 4.3 Domain of ABC .

	$b_0 = 0$	$b_1 = 1$	$b_2 = 2$	$b_3 = 3$
$a_0 = 0$	0	2	4	6
$a_1 = 1$	1	3	5	7
$a_2 = 2$	2	4	6	8
$a_3 = 3$	3	5	7	9

Table 4.4 The values in the image of Φ_{AB} .

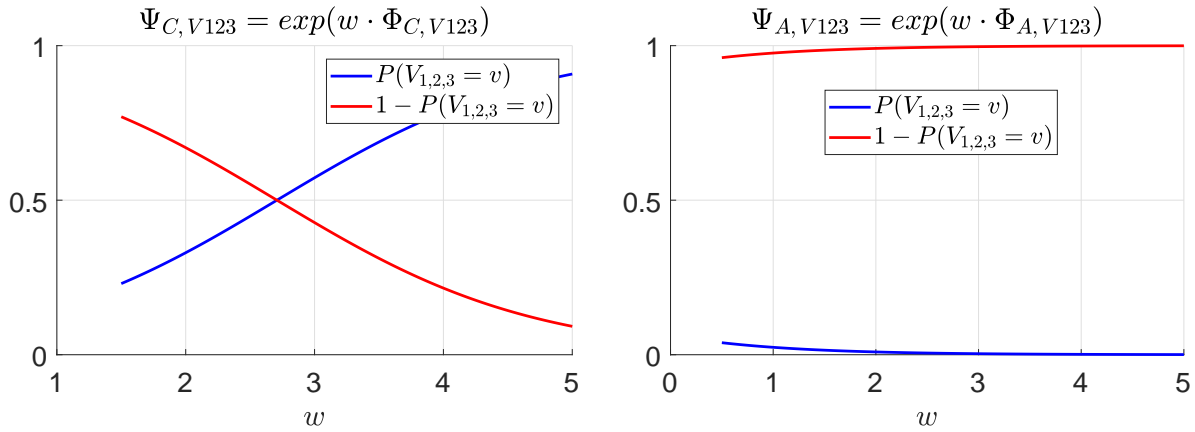


Figure 4.1 The probability $\mathbb{P}(V_1 = v, V_2 = v, V_3 = v|w)$ and its complement, when considering a ternary correlating factor, on the left, and an anti-correlating one, on the right.

4.1.2.2 part 02

Part 02 considers a ternary correlating factor $\Phi_{C, V_{123}}$, involving variables V_1 , V_2 and V_3 , each having a domain size equal to 3. Ternary factors cannot be part of a graph, but it is anyway possible to build them as distribution object. The values in the image of $\Phi_{C, V_{123}}$ are all 0, except for those combination for which V_1 , V_2 and V_3 assume the same value (0, 1 or 2) and in such cases, the image is equal to 1.

The same example builds at a second stage a ternary anti-correlating factor $\Phi_{A, V_{123}}$. The values in the image of $\Phi_{A, V_{123}}$ are all 1, except for those combination for which V_1 , V_2 and V_3 assume the same value (0, 1 or 2) and in such cases the image is equal to 0.

When considering a graph having only $\Psi_{C, V_{123}} = \exp(\Phi_{C, V_{123}} \cdot w)$ as a factor, the ripartition function Z is equal to:

$$Z = (4^3 - 4) + 4 \cdot \exp(w) \quad (4.3)$$

The probability to have as a realization a combination with the same values is equal to:

$$\mathbb{P}(V_1 = v, V_2 = v, V_3 = v) = \sum_{i=0}^3 \mathbb{P}(V_1 = i, V_2 = i, V_3 = i) \quad (4.4)$$

$$= \sum_{i=0}^3 \frac{\Psi_{C, V_{123}}(i, i, i)}{Z} \quad (4.5)$$

$$= 4 \cdot \frac{\Psi_{C, V_{123}}(0, 0, 0)}{Z} \quad (4.6)$$

$$= \frac{4 \cdot \exp(w)}{(4^3 - 4) + 4 \cdot \exp(w)} \quad (4.7)$$

The value assumed by $\mathbb{P}(V_1 = v, V_2 = v, V_3 = v|w)$ is reported in Figure 4.1, together with the complementary probability $1 - \mathbb{P}(V_1 = v, V_2 = v, V_3 = v|w)$.

When considering a graph having only $\Psi_{A, V_{123}} = \exp(\Phi_{A, V_{123}} \cdot w)$ as factor, the ripartition function Z is equal to:

$$Z = (4^3 - 4) \cdot \exp(w) + 4 \quad (4.8)$$

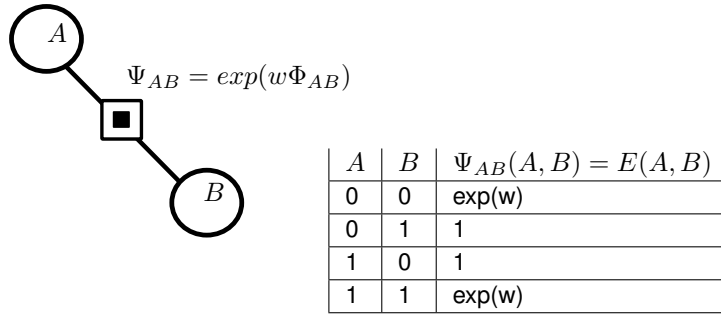


Figure 4.2 On the left the graph considered in this example, while on the right the image of factor Ψ_{AB} . Since that potential is the only one present in the graph, the values in the image of Ψ_{AB} are also the ones assume by the energy function E .

The probability to have as a realization a combination with the same values is equal to:

$$\mathbb{P}(V_1 = v, V_2 = v, V_3 = v) = \sum_{i=0}^3 \mathbb{P}(V_1 = i, V_2 = i, V_3 = i) \quad (4.9)$$

$$= \sum_{i=0}^3 \frac{\Psi_{A \ V123}(i, i, i)}{Z} \quad (4.10)$$

$$= 4 \cdot \frac{\Psi_{A \ V123}(0, 0, 0)}{Z} \quad (4.11)$$

$$= \frac{4}{(4^3 - 4) \cdot \exp(w) + 4} \quad (4.12)$$

The value assumed by $\mathbb{P}(V_1 = v, V_2 = v, V_3 = v|w)$ is reported in Figure 4.1, together with its complement $1 - \mathbb{P}(V_1 = v, V_2 = v, V_3 = v|w)$. Indeed, when the variables are correlated, i.e. they share $\Psi_{C \ V123}$, the probability $\mathbb{P}(V_1 = v, V_2 = v, V_3 = v|w)$ is big. Moreover, the more w is high (i.e. the more the variables are correlated), the more the latter probability is big. On the opposite, when the variables are anti-correlated, the opposite situation arises.

4.2 Sample 02: Belief propagation, part A

The aim of this series of examples is to show how to perform probabilistic queries on factor graphs.

4.2.1 part 01

This example creates a graph having a single binary exponential shape Ψ_{AB} , see Figure 4.2, with a A and B having a Dom size equal to 2. $\Psi_{AB} = \exp(w\Phi_{AB})$, where Φ_{AB} is a simple correlating factor. The image of Ψ_{AB} is reported in the right part of Figure 4.2.

Variable B is considered as an evidence, whose value is equal, for the first part of the example, to 0, while a value of 1 is assumed in the second part. The probability of A conditioned to B , is equal to (see equation (2.18)):

$$\mathbb{P}(A = a|B = 0) = \frac{E(A = a, B = 0)}{E(A = 0, B = 0) + E(A = 1, B = 0)} \Rightarrow \begin{bmatrix} \mathbb{P}(A = 0|B = 0) = \frac{\exp(w)}{1 + \exp(w)} \\ \mathbb{P}(A = 1|B = 0) = \frac{1}{1 + \exp(w)} \end{bmatrix} \quad (4.13)$$

$$\mathbb{P}(A = a|B = 1) = \frac{E(A = a, B = 1)}{E(A = 0, B = 1) + E(A = 1, B = 1)} \Rightarrow \begin{bmatrix} \mathbb{P}(A = 0|B = 1) = \frac{1}{1 + \exp(w)} \\ \mathbb{P}(A = 1|B = 1) = \frac{\exp(w)}{1 + \exp(w)} \end{bmatrix} \quad (4.14)$$

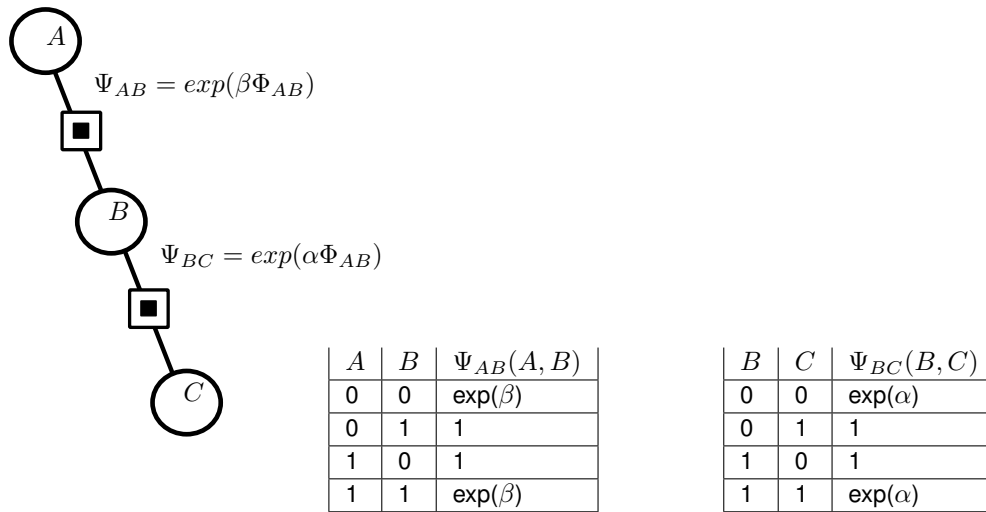


Figure 4.3 On the left the graph considered in this example, while on the right the images of factor Ψ_{AB} and Ψ_{BC} having, respectively, a weight equal to β and α .

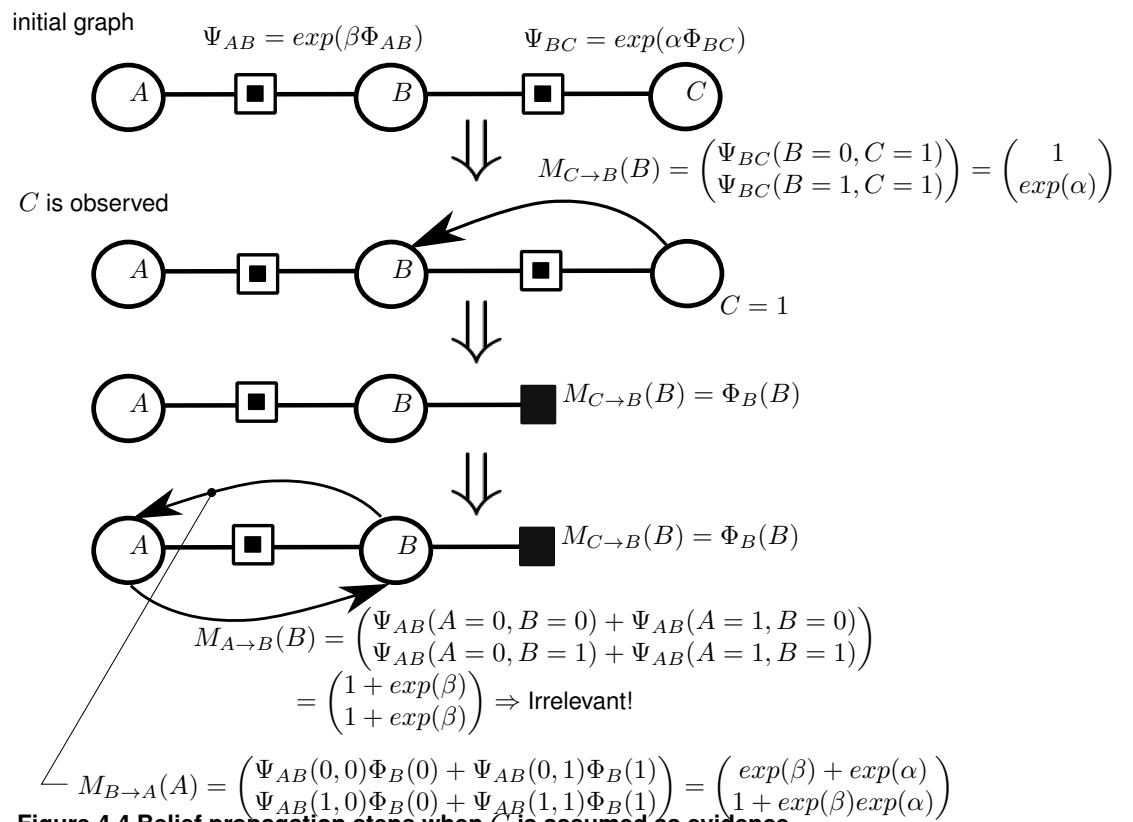


Figure 4.4 Belief propagation steps when C is assumed as evidence.

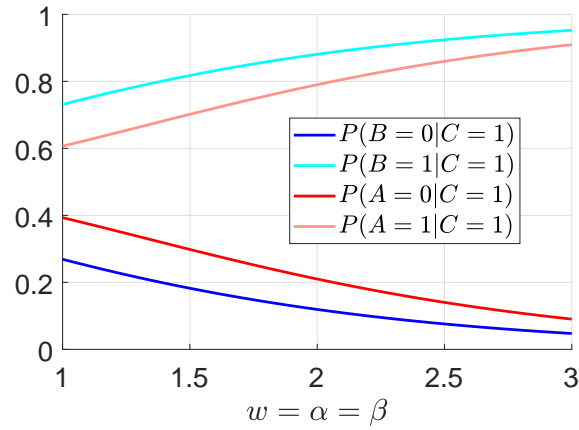


Figure 4.5 The marginals of variables B and A , when having a $C = 1$ as evidence of the graph reported in Figure 4.4.

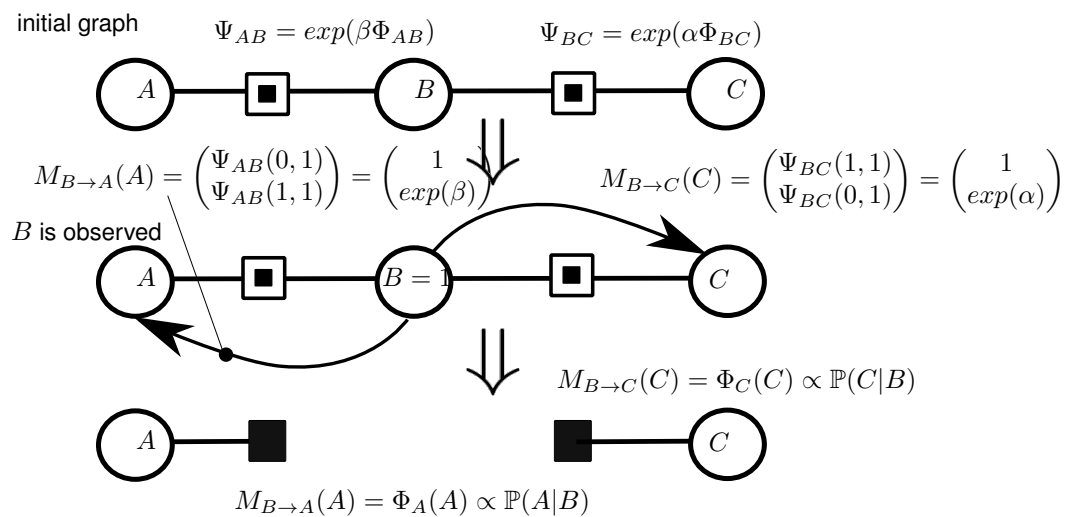
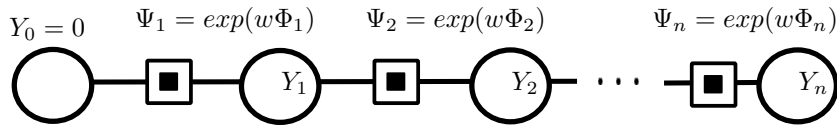


Figure 4.6 Belief propagation steps when B is assumed as evidence.



$\Psi_i(Y_{i-1}, Y_i)$	$Y_i = 0$	$Y_i = 1$	\dots	$Y_i = m$
$Y_{i-1} = 0$	$\exp(w)$	1	\dots	1
$Y_{i-1} = 1$	1	$\exp(w)$	\dots	1
\vdots	\vdots	\vdots	\ddots	\vdots
$Y_{i-1} = m$	1	1	\dots	$\exp(w)$

Figure 4.7 On the top the chain considered in this example, while on the bottom the image of the generic factor $\Psi_i(Y_{i-1}, Y_i)$.

4.2.2 part 02

A slightly more complex graph, made of two exponential correlating factors Ψ_{BC} and Ψ_{AB} , is built in this sample. The considered graph is reported in Figure 4.3. The two involved factors have two different weights, α and β : the resulting image sets are reported in the right part of Figure 4.3.

In the first part, $C = 1$ is assumed as evidence and the marginal probabilities of A and B conditioned to C are computed. They are compared with the theoretical results, obtained by applying the message passing algorithm (Section 2.2.2), whose steps are here detailed ¹. The message passing steps are summarized in Figure 4.4. After having computed all the messages, it is clear that the marginal probabilities are equal to:

$$\mathbb{P}(A|C=1) = \frac{1}{Z} M_{B \rightarrow A}(A) = \frac{1}{Z} \left[\frac{\exp(\alpha) + \exp(\beta)}{1 + \exp(\alpha) \cdot \exp(\beta)} \right] \quad (4.15)$$

$$\mathbb{P}(B|C=1) = \frac{1}{Z} \Phi_B(B) \cdot M_{A \rightarrow B}(B) = \frac{1}{Z} \Phi_B(B) = \frac{1}{Z} \left[\frac{1}{\exp(\alpha)} \right] \quad (4.16)$$

Figure 4.5 shows the values assumed by the marginals when varying the coefficients α and β . As can be seen, the more A , B and C are correlated (i.e. the more α and β are big) the more $\mathbb{P}(B=1|C=1)$ and $\mathbb{P}(A=1|C=1)$ are big. Notice also that when assuming $\alpha = \beta$, $\mathbb{P}(B=1|C=1)$ is always bigger than $\mathbb{P}(A=1|C=1)$. This is intuitively explained by the fact that C is directly connected to B , while A is indirectly connected to C , through B .

In the second part, $B = 1$ is assumed as evidence and the marginal probabilities of A and C conditioned to B are computed. The theoretical results can be computed again considering the message passing, whose steps are reported in Figure 4.6. The marginal probabilities are in this case equal to:

$$\mathbb{P}(A|B=1) = \frac{1}{Z} \Phi_A(A) = \frac{1}{Z} \left[\frac{1}{\exp(\beta)} \right] \quad (4.17)$$

$$\mathbb{P}(C|B=1) = \frac{1}{Z} \Phi_C(C) = \frac{1}{Z} \left[\frac{1}{\exp(\alpha)} \right] \quad (4.18)$$

4.2.3 part 03

In this sample, a linear chain of variables $Y_{0,1,2,\dots,n}$ is considered. All variables in the chain have the same *Dom* size and all the factors $\Psi_{1,\dots,n}$, Figure 4.7, are simple exponential correlating factors. The image of the generic factor Ψ_i is reported in the right part of Figure 4.7.

The evidence $Y_0 = 0$ is assumed and the marginals of the last variable in the chain Y_n , i.e. the one furthest to Y_0 , are computed. Figure 4.8 reports the probability $\mathbb{P}(Y_n = 0|Y_0 = 0)$, when varying the chain size, as well the domain size of the variables. As can be seen, the more the chain is longer, the lower is the aforementioned probability, as Y_n is more and more indirectly correlated to Y_0 . Similar considerations hold for the domain size.

¹ The same steps are internally execute by Node_factory.

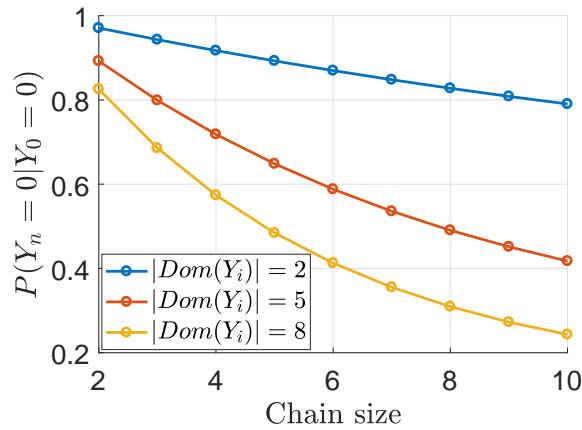


Figure 4.8 Marginal probability of Y_n when varying the chain size of the structure presented in Figure 4.7. w is assumed equal to 3.5.

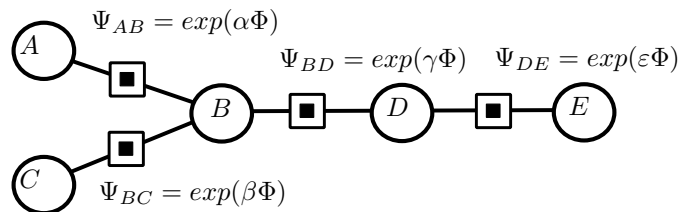


Figure 4.9 The factor graph considered by part 01.

4.3 Sample 03: Belief propagation, part B

The aim of this series of examples is to show how to perform probabilistic queries on articulated complex graphs.

4.3.1 part 01

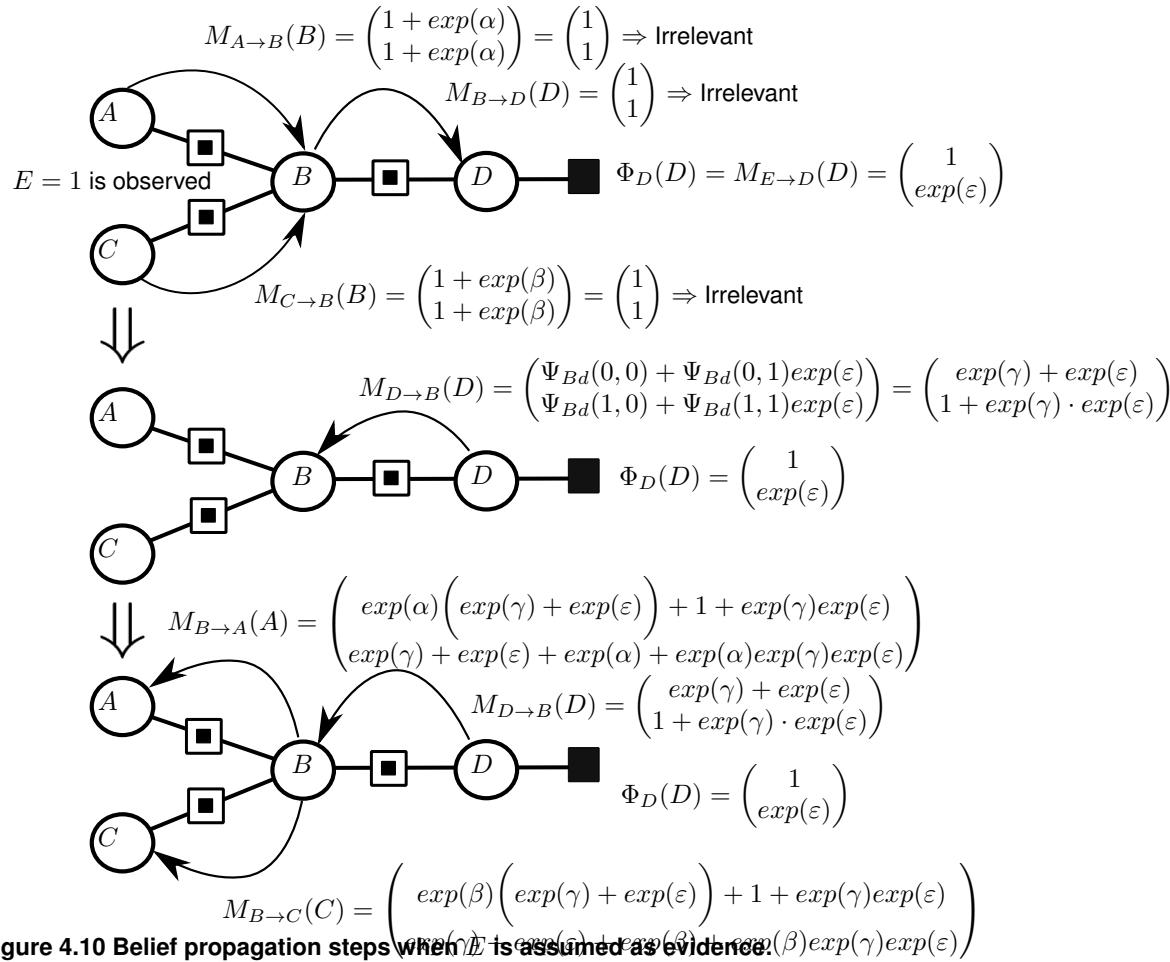
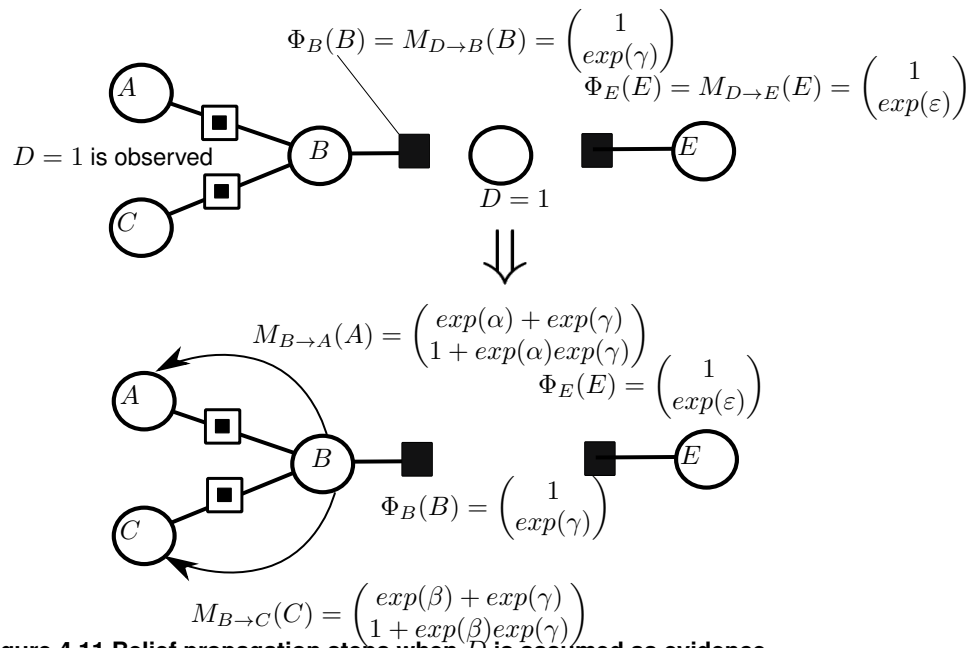
Part 01 considers a graph made of 5 variables with a Dom size equal to 2 and some simple exponential correlating factors, having different weights. The graph is reported in Figure 4.9, together with the weights of factor $\alpha, \beta, \gamma, \epsilon$. At first stage, the evidence $E = 1$ is assumed and the marginal probabilities of the other variables are computed with the message passing, whose steps are summarized in Figure 4.10. After the convergence of the message passing, the marginals of the variables are computed as similarly done for the previous examples. In a second phase, the evidence $E = 0$ is assumed. The computation of the marginals is omitted since it is specular to the previous case.

Finally, $D = 1$ is assumed and a new belief propagation is done, whose computations are reported in Figure 4.11.

4.3.2 part 02

Part 02 considers the graph reported in Figure 4.12. All the variables in Figure 4.12 have a domain size equal to 2, and all the factors are simply correlating exponential shape, having a unitary weight. Variables v_1, v_2 and v_3 are treated as evidences and the belief is propagated across the other ones, leading to the computation of the individual marginal probabilities. Since, the addressed structure is a politree (refer to Figure 2.6), the message passing algorithm converges within a finite number of steps.

In principle, the same approach followed in the previous examples can be followed to compute some theoretical results, with the aim of performing the comparisons. Anyway, for this kind of graph such an approach would be too complex. For this reason, results are compared with a Gibbs sampling approach: a series of samples $\mathcal{T} = \{T_1, \dots, T_N\}$ are taken from the joint conditioned distribution $\mathbb{P}(T = v_{4,5,6,7,8,9,10,11,12,13} | v_{1,2,3})$. Then, to

Figure 4.10 Belief propagation steps when $E = 1$ is assumed as evidence.Figure 4.11 Belief propagation steps when $D = 1$ is assumed as evidence.

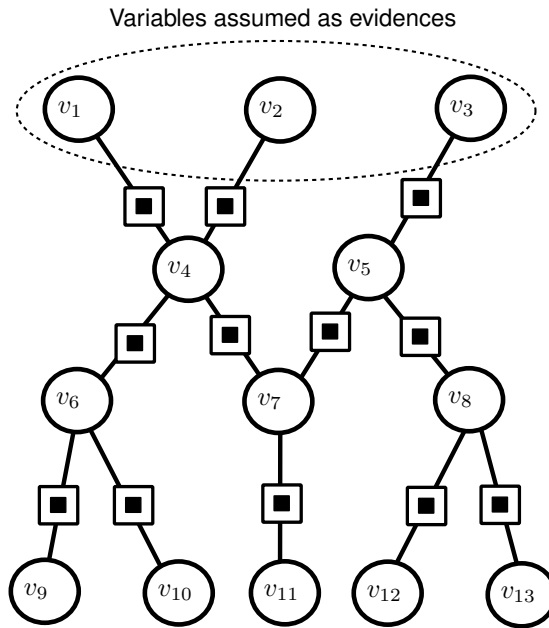


Figure 4.12 The factor graph considered by part 02.

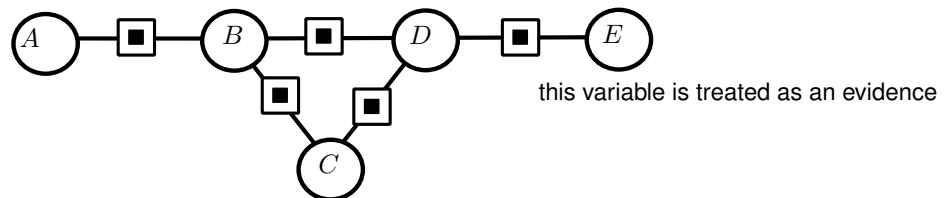


Figure 4.13 The factor graph considered by part 03.

evaluate the marginal probability $\mathbb{P}(v_i | v_{1,2,3})$ of a generic hidden variable v_i , the following empirical frequency is computed:

$$\mathbb{P}(v_i = v | v_{1,2,3}) = \frac{\sum_{T_j \in \mathcal{T}} L_{Ti}(T_j, v)}{N} \quad (4.19)$$

where $L_{Ti}(T_j, v)$ is an indicator function equal to 1 only for those samples for which v_i assumed a value equal to v .

4.3.3 part 03

Part 03 considers the graph reported in Figure 4.13. As for the example in the previous part, all variables are binary, and the potentials are simply exponential correlating with a unitary weight. However, this structure is loopy. E is treated as an evidence and the belief propagation is performed considering the loopy version of the message passing discussed in Section 2.2.2.

4.3.4 part 04

The last example in this series, considers a complex loopy graph, represented in Figure 4.14. As for other examples, all the variables are binary and the factors are exponential simply correlating with unitary weights. v_1 is assumed as evidence and the belief is propagated with the loopy version of message passing. Results are compared to the empirical frequencies obtained with a Gibbs sampler, as similarly done for the example of part 02.

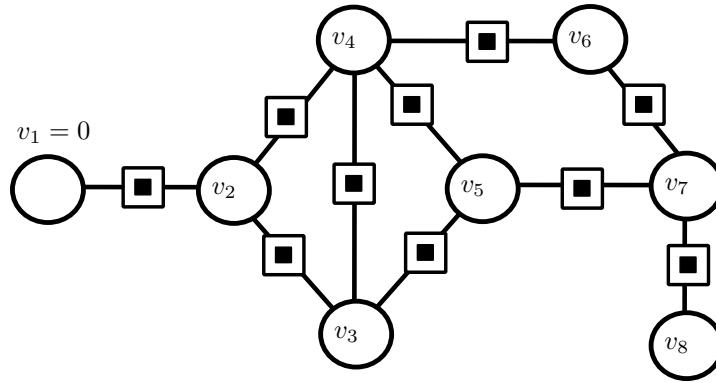


Figure 4.14 The factor graph considered by part 04.

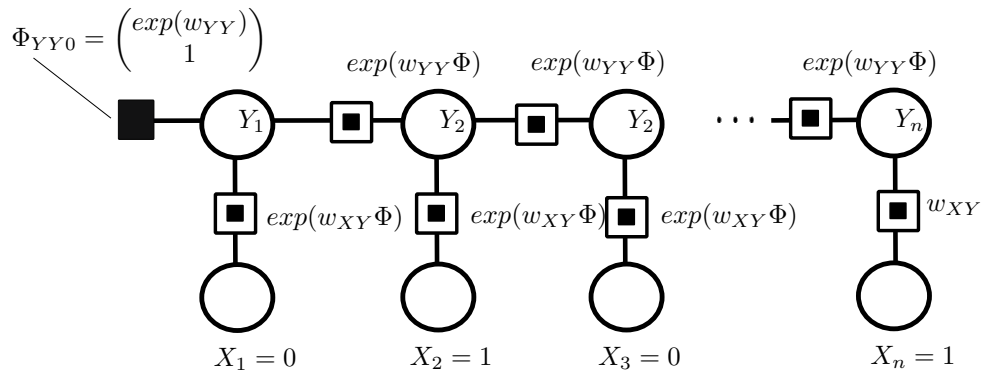


Figure 4.15 The chain structure considered by Sample 04.

4.4 Sample 04: Hidden Markov model like structure

The structure reported in Figure 4.15 is considered in this example. The reported chain is similar to those considered in Hidden Markov models, for which the chain of hidden variables $Y_{1,2,\dots}$ are connected to the chain of evidences $X_{1,2,\dots}$. The potential Φ_{YY0} , represents an a-priori knowledge about variable Y_1 . All the variables are binary and the potentials are simply correlating exponential potentials. In particular, the ones connecting the hidden variables have a weight equal to w_{YY} , while the ones connecting the evidences to the hidden set share a weight equal to w_{XY} . The evidences are set as indicated in Figure 4.15, i.e. 0 and 1 are alternated in the chain represented by $X_{1,2,\dots}$. The MAP estimation² of the hidden set (Section 2.3) is computed into two different situations:

- case a): $w_{XY} \gg w_{YY}$
- case b): $w_{XY} \ll w_{YY}$

Here the point is that when considering case a), the information about the evidences and the correlations between $Y_{1,2,\dots}$ and $X_{1,2,\dots}$ is predominant. On the opposite, when dealing with case b), the correlations among the hidden variables as well as the prior knowledge about Y_0 is predominant. For this reason, for case a) the MAP estimation of the hidden variables is equal to $h_{MAP}^a = \{0, 1, 0, 1, \dots\}$, while for case b) the MAP estimation is equal to $h_{MAP}^b = \{0, 0, 0, 0, \dots\}$.

4.5 Sample 05: Matricial structure

The matrix-like structure reported in Figure 4.16 is considered in this example. The variables in the matrix have all the same domain size and they are correlated by the potentials populating the matrix, which are all simple exponential

²The Node_factory class is in charge of invoking the proper version of the message passing algorithm that leads to the MAP estimation computation.

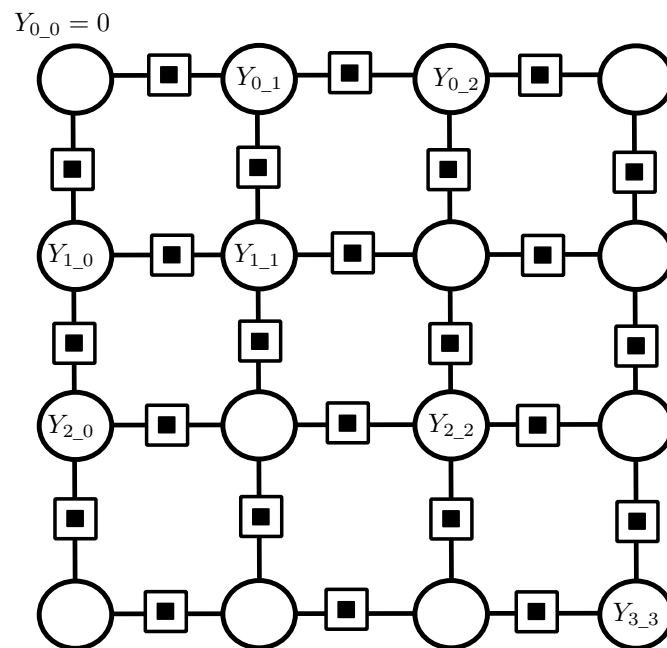


Figure 4.16 The matricial structure considered by Sample 05.

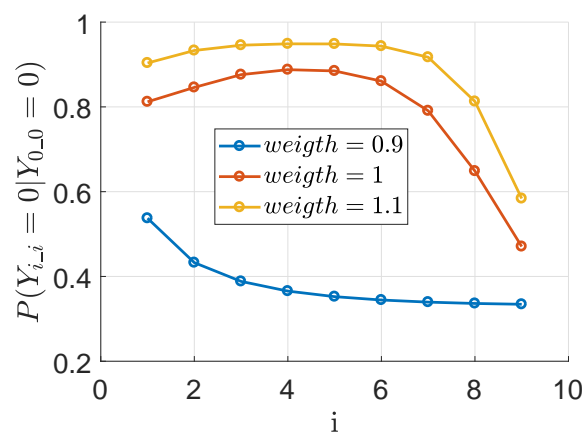


Figure 4.17 The marginals of variables $Y_{i,i}$, conditioned to $Y_{0,0} = 0$ as evidence of the graph reported in Figure 4.16, when varying the weight of the correlating exponential factors involved in the structure.

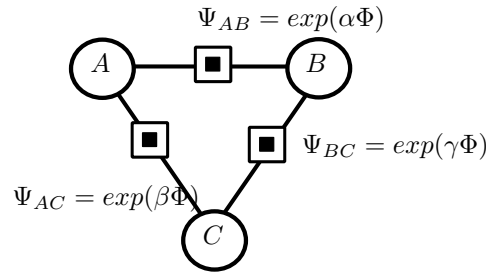


Figure 4.18 The graph considered by part 01.

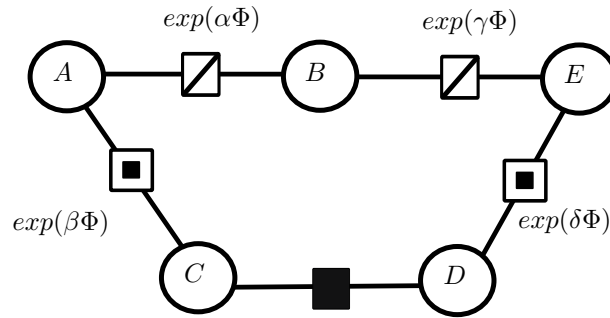


Figure 4.19 The graph considered by part 02.

correlating factors sharing the same weight. The example builds the matrix and then assumes $Y_{0_0} = 0$ as an evidence. Then, the marginals of the variables along the diagonal of the matrix, i.e. Y_{i_i} , are evaluated. As can be seen, the marginal probability $\mathbb{P}(Y_{i_i} = 0 | Y_{0_0})$ decreases descending the diagonal. Figure 4.17 reports the results obtained when varying the weight of the correlating factors, on a matrix made of 10×10 variables having a *Dom* size equal to 3.

4.6 Sample 06: Learning, part A

The aim of this series of examples is to show how to perform the learning of factor graphs. In all the examples contained in this Section, learning is done with the following methodology. A Gibbs sampler is used to take samples from the joint distribution correlating all the variables in a model A. Model A is actually the model to learn. The training set obtained from model A, is used to train a model B. Model B has the same variables and factors of model A, but with different values for the free parameters $w_{1,2,\dots}$ (Section 2.6). In this way, after having performed the learning, the value of the free parameters in model B will assume similar values to the ones in model A, showing the effectiveness of the functionalities contained in EFG. Clearly, this is not the approach followed in real applications, where the real coefficient of the model are unknown and only a training set of examples are available.

4.6.1 part 01

Part 01 considers the loopy graph reported in Figure 4.18. A , B and C are all binary variables, while Ψ_{AB} , Ψ_{AC} and Ψ_{BC} are simple correlating exponential distributions having as weights, respectively, α , β and γ . Some samples are generated from the model with a Gibbs sampling, in order to use it later as a training set. All the weights in the model are set to 1, and tuned using a quasi newton trainer that consider the previously generated training set. The aim is to show that after training the values of the weights are similar to the initial ones.

4.6.2 part 02

Part 02 considers a structure made of both tunable and non-tunable factors. The considered structure is reported in Figure 4.19. Weights β and δ must be tuned through learning, as similarly described in Section 4.6.1, while α and γ are constant and known (refer also to the formalism described in Figure 2.2).

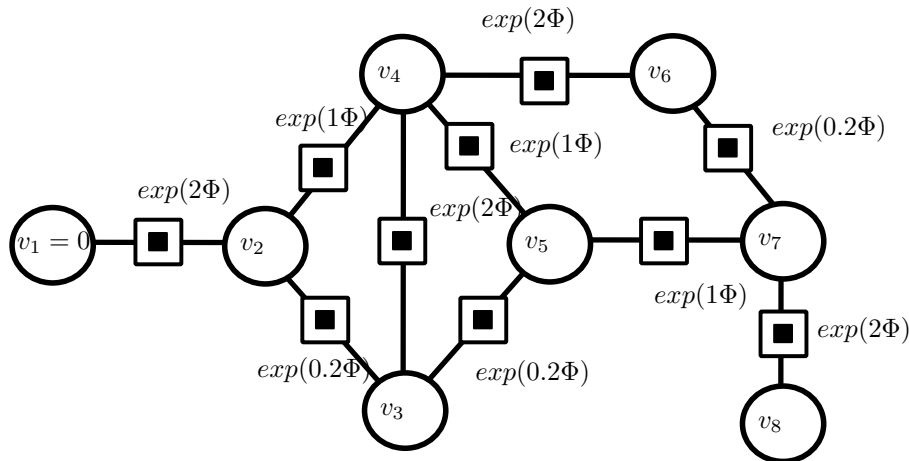


Figure 4.20 The graph considered by part 03.

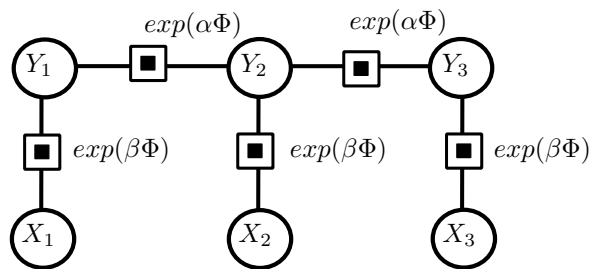


Figure 4.21 The graph considered by part 04.

4.6.3 part 03

Part 03 considers the loopy structure reported in Section 4.3.4. However, here instead of having constant exponential shapes, all the factors are made of tunable exponentials. The value assumed by the weight of model A (see the introduction of this Section) are showed in Figure 4.20. Weights are tuned through learning as similarly described in Section 4.6.1.

4.6.4 part 04

Part 04 considers the structure reported in Figure 4.21, for which all the potentials connecting pairs Y_{i-1}, Y_i share the same weight α , while the factors connecting pairs X_i, Y_i share the weight β . The approach described Weights are tuned through learning as similarly described in Section 4.6.1.

4.7 Sample 07: Learning, part B

The aim of this example is to show how the learning process can be done when dealing with Conditional random fields. In particular, the structure reported in Figure 4.22 is considered (values of the free parameters are not indicated, since the reader may refer to the sources provided).

The approach adopted is similar to the one followed in the previous series of example, considering a couple of model A and B (see the initial part of the previous Section). However, in this case we cannot simply draw samples from the joint distribution correlating the variables in the model, since such a distribution does not exists. Indeed, the conditional random field of Figure 4.22, models the conditional distribution of variables $Y_{1,2,\dots}$ w.r.t the evidences $X_{1,2,\dots}$. For this reason, all the possible combination of evidences are determined, considering all $x \in \{Dom(X_1) \cup Dom(X_2) \cup \dots\}$. For each x , samples from the conditioned distribution $\mathbb{P}(Y_{1,2,\dots}|x)$ are taken with a Gibbs sampler. The entire population of samples determined is actually the training set adopted for training the conditional random field in Figure 4.22.

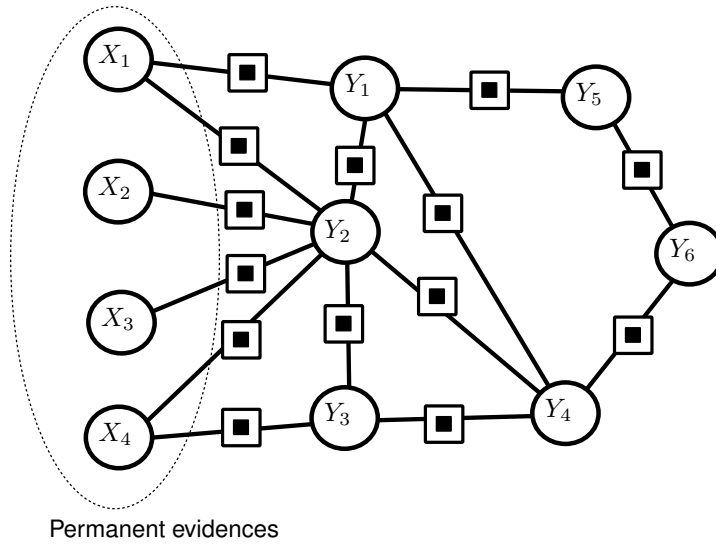


Figure 4.22 The conditional random field considered in Sample 07.

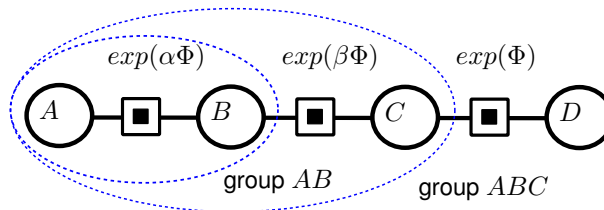


Figure 4.23 The chain considered in the example. All the underlying simple shapes are simple correlating.

4.8 Sample 08: Sub-graphing

4.8.1 part 01

The chain structure described in Figure 4.23 is addressed in this example. The sub-graphs containing variables A, B, C and A, B are built, in order to compute the joint marginal probability distributions of that two groups of variables.

The values are compared to the real ones, obtained by considering the joint distribution³ of all the variables in the chain, which can be obtained by computing the energy function E , equation (2.6) (computations are here omitted for brevity). Knowing the joint distribution of a group of variables, the marginal distribution of a sub-group can be obtained by marginalization, equation (2.4):

$$\begin{aligned}\mathbb{P}(A = a, B = b, C = c) &= \sum_{\tilde{d} \in \text{Dom}(D)} \mathbb{P}(A = a, B = b, C = c, D = \tilde{d}) \\ \mathbb{P}(A = a, B = b) &= \sum_{\tilde{c} \in \text{Dom}(C), \tilde{d} \in \text{Dom}(D)} \mathbb{P}(A = a, B = b, C = \tilde{c}, D = \tilde{d})\end{aligned}\quad (4.20)$$

Applying the above rules to the chain of Figure 4.23 leads to obtain the theoretical marginal distributions indicated in Figure 4.24.

4.8.2 part 02

The aim of this example is to show how sub-graphs (see Section 2.5) can be computed using SubGraph. The example starts building the structure described in Figure 4.25 (refer to the sources for the details regarding the

³Which is significantly time consuming to compute. For this reason, the SubGraph class is able to compute the marginals without explicitly compute the entire joint distribution of the variables in the model. Here we want just to compare the theoretical result with the one obtained by the SubGraph class.

A	B	$\mathbb{P}(A, B)$
0	0	$\frac{\exp(\alpha)}{1+\exp(\alpha)}$
0	1	$\frac{1}{1+\exp(\alpha)}$
1	0	$\frac{1}{1+\exp(\alpha)}$
1	1	$\frac{\exp(\alpha)}{1+\exp(\alpha)}$

A	B	C	$\mathbb{P}(A, B, C)$
0	0	0	$\frac{1}{Z_{ABC}} \cdot \exp(\alpha)\exp(\beta)$
0	1	0	$\frac{1}{Z_{ABC}}$
1	0	0	$\frac{1}{Z_{ABC}} \cdot \exp(\beta)$
1	1	0	$\frac{1}{Z_{ABC}} \cdot \exp(\alpha)$
0	0	1	$\frac{1}{Z_{ABC}} \cdot \exp(\alpha)$
0	1	1	$\frac{1}{Z_{ABC}} \cdot \exp(\beta)$
1	0	1	$\frac{1}{Z_{ABC}}$
1	1	1	$\frac{1}{Z_{ABC}} \cdot \exp(\alpha)\exp(\beta)$

Figure 4.24 Marginal probabilities of the sub-groups $\{A, B, C\}$ and $\{A, B\}$. The normalization coefficient

$$Z_{ABC} \text{ is equal to } Z_{ABC} = 2 \left(1 + \exp(\alpha) + \exp(\beta) + \exp(\alpha)\exp(\beta) \right).$$

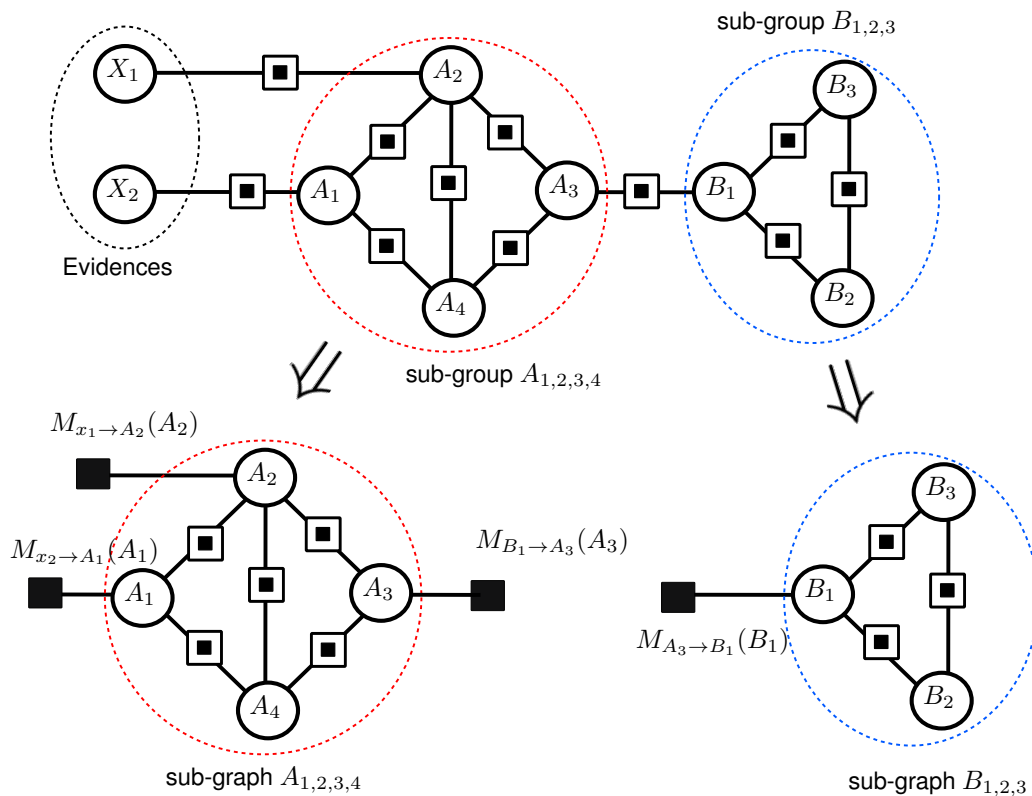


Figure 4.25 On the top, the graph considered by Sample 08, while on the bottom the sub-structures of the two groups $A_{1,2,3,4}$ and $B_{1,2,3}$.

variables and factors involved in the structure) and assumes the following evidences: $X_1 = 0$ and $X_2 = 0$. Then, the two sub-graphs considering the sub-group of variables $A_{1,2,3,4}$ and $B_{1,2,3}$ are computed, refer to Figure 4.25. The marginal probabilities of some combinations for $A_{1,2,3,4}$ conditioned to the evidences $X_{1,2}$ are computed and compared with the empirical frequencies computed considering a samples set produced by a Gibb sampler on the entire graph: samples for $t = A_{1,2,3,4}, B_{1,2,3}$ are drawn and the empirical frequencies of specific combinations of $A_{1,2,3,4}$ are computed as similarly done in 4.3.2. The same thing is done for the sub-graph $B_{1,2,3}$. At a second stage, the evidences $X_{1,2}$ are changed and the sub-graphs, as well as the marginal probabilities, are consequently recomputed.

Chapter 5

Namespace Index

5.1 Namespace List

Here is a list of all documented namespaces with brief descriptions:

EFG	55
EFG::categoric	57
EFG::factor	58
EFG::io	59
EFG::io::json	60
EFG::io::xml	60
EFG::model	60
EFG::strct	61
EFG::train	62

Chapter 6

Hierarchical Index

6.1 Class Hierarchy

This inheritance list is sorted roughly, but not completely, alphabetically:

EFG::factor::Factor::CloneTrasformedImagesTag	68
EFG::strct::PropagationResult::ClusterInfo	68
EFG::factor::Function::CombinationHasher	68
EFG::Comparator< T >	69
EFG::strct::Node::Connection	72
EFG::strct::GibbsSampler::SamplerNode::DynamicDependency	73
EFG::io::json::Exporter	76
EFG::io::xml::Exporter	77
EFG::io::xml::ExportInfo	78
EFG::factor::Function	89
EFG::strct::StateAware::GraphState	93
EFG::categoric::Group	93
EFG::categoric::GroupRange	96
std::hash< EFG::categoric::Variable >	98
EFG::Hasher< T >	98
EFG::strct::HiddenCluster	98
EFG::factor::ImageFinder	99
EFG::factor::Immutable	100
EFG::factor::Factor	79
EFG::factor::UnaryFactor	120
EFG::factor::Evidence	73
EFG::factor::Indicator	103
EFG::factor::MergedUnaries	105
EFG::factor::MessageMAP	106
EFG::factor::MessageSUM	107
EFG::factor::FactorExponential	82
EFG::io::json::Importer	101
EFG::io::xml::Importer	102
EFG::factor::Function::Info	103
EFG::train::TrainSet::Iterator	103
EFG::strct::LoopyBeliefPropagationStrategy	105
EFG::strct::BaselineLoopyPropagator	65
EFG::detail::Merger< Separator >	106
EFG::factor::Mutable	107

EFG::factor::Factor	79
EFG::factor::FactorExponential	82
EFG::detail::Node< T >	108
EFG::strct::Node	108
EFG::detail::Node< Ts >	108
EFG::Visitor< Ts >	123
EFG::detail::NodeConst< T >	109
EFG::detail::NodeConst< Ts >	109
EFG::VisitorConst< Ts >	124
EFG::strct::StateAware::NodeLocation	109
EFG::strct::Pool	110
EFG::strct::PoolAware	110
EFG::strct::BeliefAware	66
EFG::strct::EvidenceRemover	74
EFG::model::ConditionalRandomField	70
EFG::model::Graph	92
EFG::model::RandomField	114
EFG::strct::EvidenceSetter	75
EFG::model::ConditionalRandomField	70
EFG::model::Graph	92
EFG::model::RandomField	114
EFG::strct::FactorsAware	84
EFG::model::ConditionalRandomField	70
EFG::strct::FactorsConstGetter	84
EFG::strct::FactorsConstInserter	85
EFG::model::ConditionalRandomField	70
EFG::model::Graph	92
EFG::model::RandomField	114
EFG::train::FactorsTunableGetter	86
EFG::model::ConditionalRandomField	70
EFG::train::FactorsTunableInserter	88
EFG::model::ConditionalRandomField	70
EFG::model::RandomField	114
EFG::strct::GibbsSampler	91
EFG::model::ConditionalRandomField	70
EFG::model::Graph	92
EFG::model::RandomField	114
EFG::strct::QueryManager	111
EFG::model::ConditionalRandomField	70
EFG::model::Graph	92
EFG::model::RandomField	114
EFG::strct::GibbsSampler	91
EFG::strct::QueryManager	111
EFG::strct::PropagationContext	110
EFG::strct::PropagationResult	111
std::runtime_error	
EFG::Error	73
EFG::strct::GibbsSampler::SamplerNode	115
EFG::strct::GibbsSampler::SamplesGenerationContext	115
EFG::strct::PoolAware::ScopedPoolActivator	116
EFG::factor::Factor::SimplyAntiCorrelatedTag	116
EFG::factor::Factor::SimplyCorrelatedTag	116
EFG::strct::StateAware	117
EFG::strct::BeliefAware	66
EFG::strct::EvidenceRemover	74
EFG::strct::EvidenceSetter	75

EFG::strct::FactorsAware	84
EFG::strct::GibbsSampler	91
EFG::strct::QueryManager	111
EFG::strct::HiddenCluster::TopologyInfo	119
EFG::train::TrainInfo	119
EFG::train::TrainSet	119
EFG::train::Tuner	120
EFG::train::BaseTuner	65
EFG::train::BinaryTuner	67
EFG::train::UnaryTuner	121
EFG::train::CompositeTuner	69
EFG::strct::UniformSampler	122
std::unique_ptr	
EFG::Cache< std::vector< EFG::strct::HiddenCluster::TopologyInfo > >	67
EFG::Cache< const EFG::factor::UnaryFactor >	67
EFG::Cache< T >	67
EFG::categoric::Variable	122

Chapter 7

Class Index

7.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

EFG::struct::BaselineLoopyPropagator	65
EFG::train::BaseTuner	65
EFG::struct::BeliefAware	
The propagation relies on a concrete implementation of a BeliefPropagationStrategy . In case no other is specified, a default one, BaselineBeliefPropagator , is instantiated and used internally.	
You can override this default propagator using setPropagationStrategy(...)	66
EFG::train::BinaryTuner	67
EFG::Cache< T >	67
EFG::factor::Factor::CloneTrasformedImagesTag	68
EFG::struct::PropagationResult::ClusterInfo	68
EFG::factor::Function::CombinationHasher	68
EFG::Comparator< T >	69
EFG::train::CompositeTuner	69
EFG::model::ConditionalRandomField	
Similar to RandomField , with the difference that the model structure is immutable after construction. This applies also to the evidence set, which can't be changed over the time	
EFG::struct::Node::Connection	72
EFG::struct::GibbsSampler::SamplerNode::DynamicDependency	73
EFG::Error	73
EFG::factor::Evidence	73
EFG::struct::EvidenceRemover	74
EFG::struct::EvidenceSetter	75
EFG::io::json::Exporter	76
EFG::io::xml::Exporter	77
EFG::io::xml::ExportInfo	78
EFG::factor::Factor	79
EFG::factor::FactorExponential	
An exponential factor applies an exponential function to map the raw values inside the combination map to the actual images. More precisely, given the weight w characterizing the factor the image value is obtained in this way: $image = \exp(w * raw_value)$	
EFG::struct::FactorsAware	84
EFG::struct::FactorsConstGetter	84
EFG::struct::FactorsConstInsertter	85
EFG::train::FactorsTunableGetter	86
EFG::train::FactorsTunableInsertter	88

EFG::factor::Function	89
EFG::strct::GibbsSampler	
Refer also to https://en.wikipedia.org/wiki/Gibbs_sampling	91
EFG::model::Graph	
A simple graph object, that stores only const factors. Evidences may be changed over the time	92
EFG::strct::StateAware::GraphState	93
EFG::categoric::Group	
An ensemble of categoric variables. Each variable in the ensemble should have its own unique name	93
EFG::categoric::GroupRange	
This object allows to iterate all the elements in the joint domain of a group of variables, without precomputing all the elements in such domain. For example when having a domain made by variables = { A (size = 2), B (size = 3), C (size = 2) }, the elements in the joint domain that will be iterated are: <0,0,0> <0,0,1> <0,1,0> <0,1,1> <0,2,0> <0,2,1> <1,0,0> <1,0,1> <1,1,0> <1,1,1> <1,2,0> <1,2,1> After construction, the Range object starts to point to the first element in the joint domain <0,0,...>. Then, when incrementing the object, the following element is pointed. When calling get() the current pointed element can be accessed	96
std::hash< EFG::categoric::Variable >	98
EFG::Hasher< T >	98
EFG::strct::HiddenCluster	
Clusters of hidden node. Each cluster is a group of connected hidden nodes. Nodes in different clusters are not currently connected, due to the model structure or the kind of evidences currently applied	98
EFG::factor::ImageFinder	
An object used to search for the images associated to sub combinations that are part of a bigger one	99
EFG::factor::Immutable	100
EFG::io::json::Importer	101
EFG::io::xml::Importer	102
EFG::factor::Indicator	103
EFG::factor::Function::Info	103
EFG::train::TrainSet::Iterator	
Object able to iterate all the combinations that are part of a training set or a sub portion of it	103
EFG::strct::LoopyBeliefPropagationStrategy	105
EFG::factor::MergedUnaries	105
EFG::detail::Merger< Separator >	106
EFG::factor::MessageMAP	106
EFG::factor::MessageSUM	107
EFG::factor::Mutable	107
EFG::detail::Node< T >	108
EFG::strct::Node	108
EFG::detail::NodeConst< T >	109
EFG::strct::StateAware::NodeLocation	109
EFG::strct::Pool	110
EFG::strct::PoolAware	110
EFG::strct::PropagationContext	110
EFG::strct::PropagationResult	
Structure that can be exposed after having propagated the belief, providing info on the encountered structure	111
EFG::strct::QueryManager	111
EFG::model::RandomField	
A complete undirected factor graph storing both constant and tunable factors. Evidences may be changed over the time	114
EFG::strct::GibbsSampler::SamplerNode	115
EFG::strct::GibbsSampler::SamplesGenerationContext	115
EFG::strct::PoolAware::ScopedPoolActivator	116
EFG::factor::Factor::SimplyAntiCorrelatedTag	116
EFG::factor::Factor::SimplyCorrelatedTag	116

EFG::strct::StateAware	117
EFG::strct::HiddenCluster::TopologyInfo	119
EFG::train::TrainInfo	119
EFG::train::TrainSet	119
EFG::train::Tuner	120
EFG::factor::UnaryFactor	120
EFG::train::UnaryTuner	121
EFG::strct::UniformSampler	122
EFG::categoric::Variable	
An object representing an immutable categoric variable	122
EFG::Visitor< Ts >	123
EFG::VisitorConst< Ts >	124

Chapter 8

Namespace Documentation

8.1 EFG Namespace Reference

Namespaces

- [categoric](#)
- [factor](#)
- [io](#)
- [model](#)
- [strct](#)
- [train](#)

Classes

- class [Error](#)
- class [Cache](#)
- struct [Hasher](#)
- struct [Comparator](#)
- struct [Visitor](#)
- struct [VisitorConst](#)

Typedefs

- `template<typename K , typename V >`
using [SmartMap](#) = `std::unordered_map< std::shared_ptr< K >, V, Hasher< K >, Comparator< K > >`
An unordered_map that stores as keys shared pointers that can't be null.
- `template<typename T >`
using [SmartSet](#) = `std::unordered_set< std::shared_ptr< T >, Hasher< T >, Comparator< T > >`
An unordered_set that stores shared pointers that can't be null.

Functions

- `template<typename DownCastT , typename T , typename Predicate >`
void **castConstAndUse** (const T &subject, const Predicate &predicate)
- `template<typename DownCastT , typename T , typename Predicate >`
void **castAndUse** (T &subject, const Predicate &predicate)
- `template<typename K , typename V , typename Predicate >`
void **for_each** ([SmartMap](#)< K, V > &subject, const Predicate &pred)
- `template<typename K , typename V , typename Predicate >`
void **for_each** (const [SmartMap](#)< K, V > &subject, const Predicate &pred)
- `template<typename T , typename Predicate >`
void **for_each** (const [SmartSet](#)< T > &subject, const Predicate &pred)
- `template<char Separator, typename... Args>`
void **join** (std::ostream &recipient, Args &&...args)
- `template<char Separator, typename... Args>`
std::string **join** (Args &&...args)

8.1.1 Detailed Description

Author: Andrea Casalino Created: 01.01.2021

report any bug to andreca91@gmail.com.

Author: Andrea Casalino Created: 31.03.2022

report any bug to andreca91@gmail.com.

8.1.2 Typedef Documentation

8.1.2.1 SmartMap

```
template<typename K , typename V >
using EFG::SmartMap = typedef std::unordered_map<std::shared_ptr<K>, V, Hasher<K>, Comparator<K>
>
```

An `unordered_map` that stores as keys shared pointers that can't be null.

Keys are hashed by hashing the elements wrapped inside the shared pointer.

Keys are compared by comparing the elements wrapped inside the shared pointer.

8.1.2.2 SmartSet

```
template<typename T >
using EFG::SmartSet = typedef std::unordered_set<std::shared_ptr<T>, Hasher<T>, Comparator<T>
>
```

An `unordered_set` that stores shared pointers that can't be null.

Elements are hashed by hashing the elements wrapped inside the shared pointer.

Elements are compared by comparing the elements wrapped inside the shared pointer.

8.2 EFG::categoric Namespace Reference

Classes

- class [Group](#)
An ensemble of categoric variables. Each variable in the ensemble should have its own unique name.
- class [GroupRange](#)
This object allows to iterate all the elements in the joint domain of a group of variables, without precomputing all the elements in such domain. For example when having a domain made by variables = { A (size = 2), B (size = 3), C (size = 2) }, the elements in the joint domain that will be iterated are: <0,0,0> <0,0,1> <0,1,0> <0,1,1> <0,2,0> <0,2,1> <1,0,0> <1,0,1> <1,1,0> <1,1,1> <1,2,0> <1,2,1> After construction, the Range object starts to point to the first element in the joint domain <0,0,...>. Then, when incrementing the object, the following element is pointed. When calling get() the current pointed element can be accessed.
- class [Variable](#)
An object representing an immutable categoric variable.

Typedefs

- using **VariablesSoup** = std::vector< VariablePtr >
- using **VariablesSet** = [SmartSet](#)< [Variable](#) >
- using **VariablePtr** = std::shared_ptr< [Variable](#) >

Functions

- VariablesSet **to_vars_set** (const VariablesSoup &soup)
- VariablesSet & **operator=** (VariablesSet &subject, const VariablesSet &to_remove)
removes the second set from the first
- VariablesSet **get_complementary** (const VariablesSet &entire_set, const VariablesSet &subset)
- bool **operator==** (const [GroupRange](#) &a, const [GroupRange](#) &b)
- bool **operator!=** (const [GroupRange](#) &a, const [GroupRange](#) &b)
- template<typename Predicate >
void **for_each_combination** ([GroupRange](#) &range, const Predicate &predicate)
Applies the passed predicate to all the elements that can be ranged using the passed range.
- VariablePtr **make_variable** (std::size_t size, const std::string &name)

8.2.1 Detailed Description

Author: Andrea Casalino Created: 01.01.2021

report any bug to andreca91@gmail.com.

8.2.2 Function Documentation

8.2.2.1 `get_complementary()`

```
VariablesSet EFG::categoric::get_complementary (
    const VariablesSet & entire_set,
    const VariablesSet & subset )
```

Returns

the complementary group of the `entire_set`, i.e. returns `entire_set \ subset`

8.3 EFG::factor Namespace Reference

Classes

- class [UnaryFactor](#)
- class [MergedUnaries](#)
- class [Evidence](#)
- class [Indicator](#)
- class [MessageSUM](#)
- class [MessageMAP](#)
- class [Factor](#)
- class [FactorExponential](#)

*An exponential factor applies an exponential function to map the raw values inside the combination map to the actual images. More precisely, given the weight w characterizing the factor the image value is obtained in this way: $image = exp(w * raw_value)$*

- class [Function](#)
- class [ImageFinder](#)

An object used to search for the images associated to sub combinations that are part of a bigger one.

- class [Immutable](#)
- class [Mutable](#)

Typedefs

- using **FunctionPtr** = `std::shared_ptr< Function >`
- using **ImmutablePtr** = `std::shared_ptr< Immutable >`

Functions

- `std::shared_ptr< const Function::Info > make_info` (const [categoric::Group](#) &vars)

8.3.1 Detailed Description

Author: Andrea Casalino Created: 01.01.2021

report any bug to andreca91@gmail.com.

Author: Andrea Casalino Created: 31.03.2022

report any bug to andreca91@gmail.com.

8.4 EFG::io Namespace Reference

Namespaces

- [json](#)
- [xml](#)

Typedefs

- using **Getters** = std::tuple< const [strct::StateAware](#) *, const [strct::FactorsConstGetter](#) *, const [train::FactorsTunableGetter](#) * >
- using **Inserters** = std::pair< [strct::FactorsConstInserter](#) *, [train::FactorsTunableInserter](#) * >

Functions

- template<typename ModelT >
Getters **castToGetters** (ModelT &model)
- template<typename Model >
Inserters **castToInserters** (Model &model)
- [train::TrainSet](#) **import_train_set** (const std::string &file_name)
Imports the training set from a file. The file should be a matrix of raw values. Each row represent a combination, i.e. a element of the training set to import.

8.4.1 Detailed Description

Author: Andrea Casalino Created: 01.01.2021

report any bug to andreca91@gmail.com.

8.4.2 Function Documentation

8.4.2.1 import_train_set()

```
train::TrainSet EFG::io::import_train_set (
    const std::string & file_name )
```

Imports the training set from a file. The file should be a matrix of raw values. Each row represent a combination, i.e. a element of the training set to import.

Exceptions

<i>in</i>	case the passed file is inexistent
<i>in</i>	case not all the combinations in file have the same size.

8.5 EFG::io::json Namespace Reference

Classes

- class [Exporter](#)
- class [Importer](#)

8.5.1 Detailed Description

Author: Andrea Casalino Created: 01.01.2021

report any bug to andreca91@gmail.com.

8.6 EFG::io::xml Namespace Reference

Classes

- struct [ExportInfo](#)
- class [Exporter](#)
- class [Importer](#)

8.6.1 Detailed Description

Author: Andrea Casalino Created: 01.01.2021

report any bug to andreca91@gmail.com.

8.7 EFG::model Namespace Reference

Classes

- class [ConditionalRandomField](#)
Similar to [RandomField](#), with the difference that the model structure is immutable after construction. This applies also to the evidence set, which can't be changed over the time.
- class [Graph](#)
A simple graph object, that stores only const factors. Evidences may be changed over the time.
- class [RandomField](#)
A complete undirected factor graph storing both constant and tunable factors. Evidences may be changed over the time.

8.7.1 Detailed Description

Author: Andrea Casalino Created: 01.01.2021

report any bug to andreca91@gmail.com.

8.8 EFG::strct Namespace Reference

Classes

- class [BaselineLoopyPropagator](#)
- class [EvidenceSetter](#)
- class [EvidenceRemover](#)
- class [FactorsConstGetter](#)
- class [FactorsConstInserter](#)
- class [UniformSampler](#)
- class [GibbsSampler](#)

Refer also to https://en.wikipedia.org/wiki/Gibbs_sampling.

- class [QueryManager](#)
- struct [Node](#)
- struct [HiddenCluster](#)

Clusters of hidden node. Each cluster is a group of connected hidden nodes. Nodes in different clusters are not currently connected, due to the model structure or the kind of evidences currently applied.

- struct [PropagationContext](#)
- struct [PropagationResult](#)

a structure that can be exposed after having propagated the belief, providing info on the encountered structure.

- class [LoopyBeliefPropagationStrategy](#)
- class [BeliefAware](#)

The propagation relies on a concrete implementation of a BeliefPropagationStrategy. In case no other is specified, a default one, BaselineBeliefPropagator, is instantiated and used internally. You can override this default propagator using setPropagationStrategy(...).

- class [FactorsAware](#)
- class [Pool](#)
- class [PoolAware](#)
- class [StateAware](#)

Typedefs

- using **Nodes** = [SmartMap](#)< [categoric::Variable](#), std::unique_ptr< [Node](#) > >
- using **HiddenClusters** = std::list< [HiddenCluster](#) >
- using **Evidences** = [SmartMap](#)< [categoric::Variable](#), std::size_t >
- using **LoopyBeliefPropagationStrategyPtr** = std::unique_ptr< [LoopyBeliefPropagationStrategy](#) >
- using **Task** = std::function< void(const std::size_t)>
- using **Tasks** = std::vector< Task >

Enumerations

- enum class **PropagationKind** { **SUM** , **MAP** }

Functions

- template<typename MessageT >
std::unique_ptr< MessageT > **make_message** (const [factor::UnaryFactor](#) &merged_unaries, const [factor::Immutable](#) &binary_factor)
- [HiddenClusters](#) **compute_clusters** (const std::unordered_set< [Node](#) * > &nodes)

8.8.1 Detailed Description

Author: Andrea Casalino Created: 01.01.2021

report any bug to andreca91@gmail.com.

Author: Andrea Casalino Created: 28.03.2022

report any bug to andreca91@gmail.com.

8.9 EFG::train Namespace Reference

Classes

- class [FactorsTunableGetter](#)
- class [FactorsTunableInserter](#)
- struct [TrainInfo](#)
- class [TrainSet](#)
- class [BaseTuner](#)
- class [BinaryTuner](#)
- class [CompositeTuner](#)
- class [Tuner](#)
- class [UnaryTuner](#)

Typedefs

- using **TunableClusters** = std::vector< FactorExponentialPtr >
- using **Combinations** = std::vector< std::vector< std::size_t > >
- using **FactorExponentialPtr** = std::shared_ptr< [factor::FactorExponential](#) >
- using **TunerPtr** = std::unique_ptr< [Tuner](#) >
- using **Tuners** = std::vector< TunerPtr >

Functions

- void [set_ones](#) ([FactorsTunableGetter](#) &subject)
- void [train_model](#) ([FactorsTunableGetter](#) &subject, ::train::Trainer &trainer, const [TrainSet](#) &train_set, const [TrainInfo](#) &info=[TrainInfo](#){})
- template<typename BasePred , typename CompositePred >
void **visitTuner** (const [Tuner](#) *tuner, BasePred &&base, CompositePred &&composite)
- template<typename BasePred , typename CompositePred >
void **visitTuner** ([Tuner](#) *tuner, BasePred &&base, CompositePred &&composite)

8.9.1 Detailed Description

Author: Andrea Casalino Created: 01.01.2021

report any bug to andreca91@gmail.com.

8.9.2 Function Documentation

8.9.2.1 set_ones()

```
void EFG::train::set_ones (
    FactorsTunableGetter & subject )
```


Parameters

<i>sets</i>	equal to 1 the weight of all the tunable clusters
-------------	---

8.9.2.2 train_model()

```
void EFG::train::train_model (
    FactorsTunableGetter & subject,
    ::train::Trainer & trainer,
    const TrainSet & train_set,
    const TrainInfo & info = TrainInfo{} )
```

Parameters

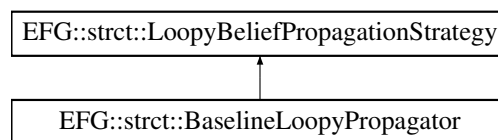
<i>the</i>	model to tune
<i>the</i>	training approach to adopt
<i>the</i>	train set to use

Chapter 9

Class Documentation

9.1 EFG::strct::BaselineLoopyPropagator Class Reference

Inheritance diagram for EFG::strct::BaselineLoopyPropagator:



Public Member Functions

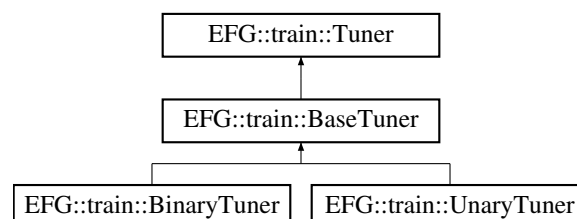
- bool **propagateBelief** ([HiddenCluster](#) &subject, PropagationKind kind, const [PropagationContext](#) &context, [Pool](#) &pool) final

The documentation for this class was generated from the following file:

- /home/andrea/Scrivania/GitProj/EFG-refactoring/src/header/EasyFactorGraph/structure/BaselineLoopyPropagator.h

9.2 EFG::train::BaseTuner Class Reference

Inheritance diagram for EFG::train::BaseTuner:



9.3.1 Detailed Description

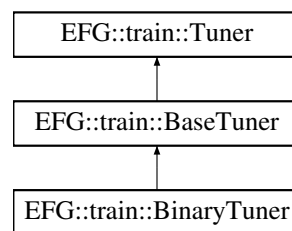
The propagation relies on a concrete implementation of a `BeliefPropagationStrategy`. In case no other is specified, a default one, `BaselineBeliefPropagator`, is instantiated and used internally. You can override this default propagator using `setPropagationStrategy(...)`.

The documentation for this class was generated from the following file:

- `/home/andrea/Scrivania/GitProj/EFG-refactoring/src/header/EasyFactorGraph/structure/bases/BeliefAware.h`

9.4 EFG::train::BinaryTuner Class Reference

Inheritance diagram for `EFG::train::BinaryTuner`:



Public Member Functions

- **BinaryTuner** (`strct::Node` &nodeA, `strct::Node` &nodeB, const `std::shared_ptr`< `factor::FactorExponential` > &factor, const `categoric::VariablesSoup` &variables_in_model)
- `float` **getGradientBeta** () final

Protected Attributes

- `strct::Node` &nodeA
- `strct::Node` &nodeB

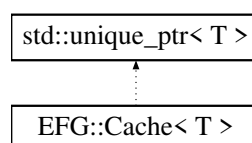
Additional Inherited Members

The documentation for this class was generated from the following file:

- `/home/andrea/Scrivania/GitProj/EFG-refactoring/src/header/EasyFactorGraph/trainable/tuners/BinaryTuner.h`

9.5 EFG::Cache< T > Class Template Reference

Inheritance diagram for `EFG::Cache< T >`:



Public Member Functions

- T & **reset** (std::unique_ptr< T > new_value=nullptr)
- bool **empty** () const
- T * **get** ()
- const T * **get** () const

The documentation for this class was generated from the following file:

- /home/andrea/Scrivania/GitProj/EFG-refactoring/src/header/EasyFactorGraph/misc/Cache.h

9.6 EFG::factor::Factor::CloneTrasformedImagesTag Struct Reference

The documentation for this struct was generated from the following file:

- /home/andrea/Scrivania/GitProj/EFG-refactoring/src/header/EasyFactorGraph/factor/Factor.h

9.7 EFG::strct::PropagationResult::ClusterInfo Struct Reference

Public Attributes

- bool **tree_or_loopy_graph**
- std::size_t **size**
number of nodes that constitutes the sub-graph cluster.

The documentation for this struct was generated from the following file:

- /home/andrea/Scrivania/GitProj/EFG-refactoring/src/header/EasyFactorGraph/structure/Types.h

9.8 EFG::factor::Function::CombinationHasher Struct Reference

Public Member Functions

- std::size_t **operator()** (const std::vector< std::size_t > &comb) const

Public Attributes

- std::shared_ptr< const [Info](#) > **info**

The documentation for this struct was generated from the following file:

- /home/andrea/Scrivania/GitProj/EFG-refactoring/src/header/EasyFactorGraph/factor/Function.h

9.9 EFG::Comparator< T > Struct Template Reference

Public Member Functions

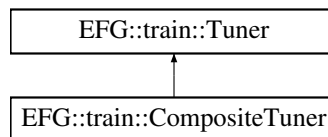
- bool **operator()** (const std::shared_ptr< T > &a, const std::shared_ptr< T > &b) const

The documentation for this struct was generated from the following file:

- /home/andrea/Scrivania/GitProj/EFG-refactoring/src/header/EasyFactorGraph/misc/SmartPointerUtils.h

9.10 EFG::train::CompositeTuner Class Reference

Inheritance diagram for EFG::train::CompositeTuner:



Public Member Functions

- Tuners & **getElements** ()
- const Tuners & **getElements** () const
- **CompositeTuner** (TunerPtr elementA, TunerPtr elementB)
- float **getGradientAlpha** (const [TrainSet::Iterator](#) &iter) final
- float **getGradientBeta** () final
- void **setWeight** (float w) final
- float **getWeight** () const final
- void **addElement** (TunerPtr element)

The documentation for this class was generated from the following file:

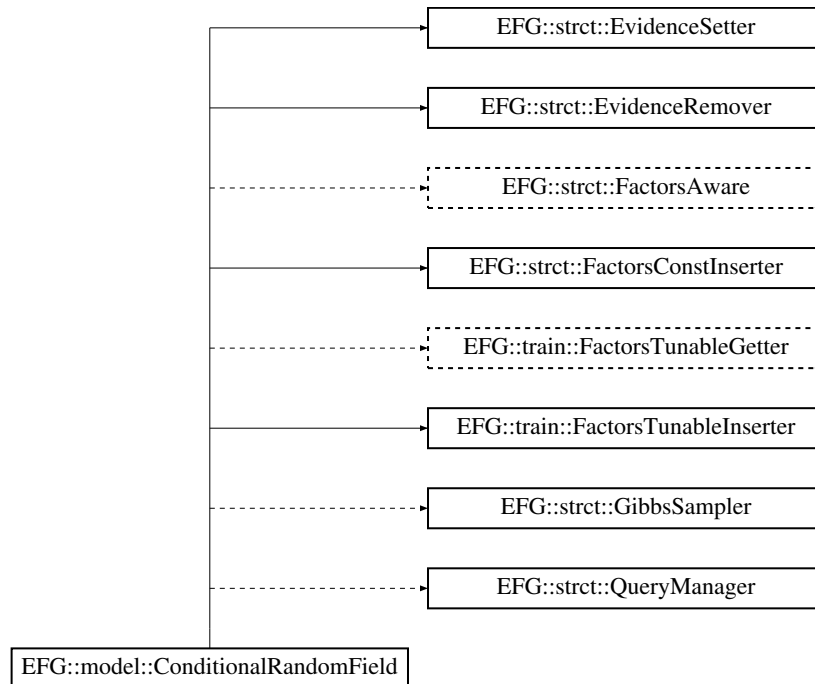
- /home/andrea/Scrivania/GitProj/EFG-refactoring/src/header/EasyFactorGraph/trainable/tuners/CompositeTuner.h

9.11 EFG::model::ConditionalRandomField Class Reference

Similar to [RandomField](#), with the difference that the model structure is immutable after construction. This applies also to the evidence set, which can't be changed over the time.

```
#include <ConditionalRandomField.h>
```

Inheritance diagram for EFG::model::ConditionalRandomField:



Public Member Functions

- **ConditionalRandomField** (const [ConditionalRandomField](#) &o)
- [ConditionalRandomField](#) & **operator=** (const [ConditionalRandomField](#) &)=delete
- [ConditionalRandomField](#) (const [RandomField](#) &source, bool copy)

All the factors of the passed source are inserted/copied. The evidence set is deduced by the passed source.
- void [setEvidences](#) (const std::vector< std::size_t > &values)

Sets the new set of evidences.
- std::vector< std::vector< std::size_t > > [makeTrainSet](#) (const GibbsSampler::SamplesGenerationContext &context, float range_percentage=1.f, std::size_t threads=1)

Builds a training set for the conditioned model. Instead of using Gibbs sampler for a single combination of evidence, it tries to span all the possible combination of evidences and generate some samples conditioned to each of this evidences value. Then, gather results to build the training set. Actually, not ALL possible evidence are spanned if that would be too much computationally demanding. In such cases, simply pass a number lower than 1 as range_percentage.

Protected Member Functions

- std::vector< float > [getWeightsGradient_](#) (const [train::TrainSet::iterator](#) &train_set_combinations) final

Additional Inherited Members

9.11.1 Detailed Description

Similar to [RandomField](#), with the difference that the model structure is immutable after construction. This applies also to the evidence set, which can't be changed over the time.

9.11.2 Constructor & Destructor Documentation

9.11.2.1 ConditionalRandomField()

```
EFG::model::ConditionalRandomField::ConditionalRandomField (
    const RandomField & source,
    bool copy )
```

All the factors of the passed source are inserted/copied. The evidence set is deduced by the passed source.

Parameters

<i>the</i>	model to emulate for building the structure of this one.
<i>then</i>	passing true the factors are deep copied, while in the contrary case the smart pointers storing the factors of the source are copied and inserted.

Exceptions

<i>in</i>	case the passed source has no evidences
-----------	---

9.11.3 Member Function Documentation

9.11.3.1 makeTrainSet()

```
std::vector<std::vector<std::size_t> > EFG::model::ConditionalRandomField::makeTrainSet (
    const GibbsSampler::SamplesGenerationContext & context,
    float range_percentage = 1.f,
    std::size_t threads = 1 )
```

Builds a training set for the conditioned model. Instead of using Gibbs sampler for a single combination of evidence, it tries to span all the possible combination of evidences and generate some samples conditioned to each of this evidences value. Then, gather results to build the training set. Actually, not ALL possible evidence are spwan if that would be too much computationally demanding. In such cases, simply pass a number lower than 1 as range_↵ percentage.

Parameters

<i>information</i>	used for samples generation
<i>parameter</i>	handling how many evidence values are accounted for the samples generation
<i>the</i>	number of threads to use for speeding up the process

9.11.3.2 setEvidences()

```
void EFG::model::ConditionalRandomField::setEvidences (
    const std::vector< std::size_t > & values )
```

Sets the new set of evidences.

Parameters

<i>the</i>	new set of evidence values. The variables order is the same of the set obtained using getObservedVariables() .
------------	--

Exceptions

<i>the</i>	number of passed values does not match the number of evidences.
<i>in</i>	case some evidence values are inconsistent

The documentation for this class was generated from the following file:

- `/home/andrea/Scrivania/GitProj/EFG-refactoring/src/header/EasyFactorGraph/model/ConditionalRandomField.h`

9.12 EFG::strct::Node::Connection Struct Reference

Public Attributes

- `factor::ImmutablePtr factor`
- `std::unique_ptr< const factor::UnaryFactor > message`

The documentation for this struct was generated from the following file:

- `/home/andrea/Scrivania/GitProj/EFG-refactoring/src/header/EasyFactorGraph/structure/Types.h`

9.13 EFG::strct::GibbsSampler::SamplerNode::DynamicDependency Struct Reference

Public Attributes

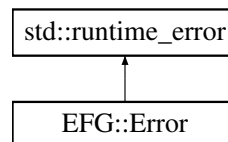
- `categoric::VariablePtr` **sender**
- `const std::size_t *` **sender_value_in_combination**
- `factor::ImmutablePtr` **factor**

The documentation for this struct was generated from the following file:

- `/home/andrea/Scrivania/GitProj/EFG-refactoring/src/header/EasyFactorGraph/structure/GibbsSampler.h`

9.14 EFG::Error Class Reference

Inheritance diagram for EFG::Error:



Public Member Functions

- **Error** (const std::string &what)

Static Public Member Functions

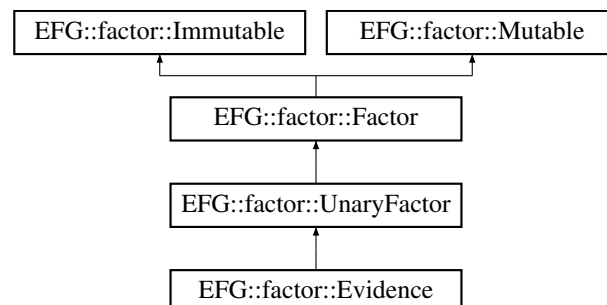
- `template<typename... Args>`
static **Error** **make** (Args &&...args)

The documentation for this class was generated from the following file:

- `/home/andrea/Scrivania/GitProj/EFG-refactoring/src/header/EasyFactorGraph/Error.h`

9.15 EFG::factor::Evidence Class Reference

Inheritance diagram for EFG::factor::Evidence:



Public Member Functions

- **Evidence** (const [Immutable](#) &binary_factor, const categoric::VariablePtr &evidence_var, std::size_t evidence)

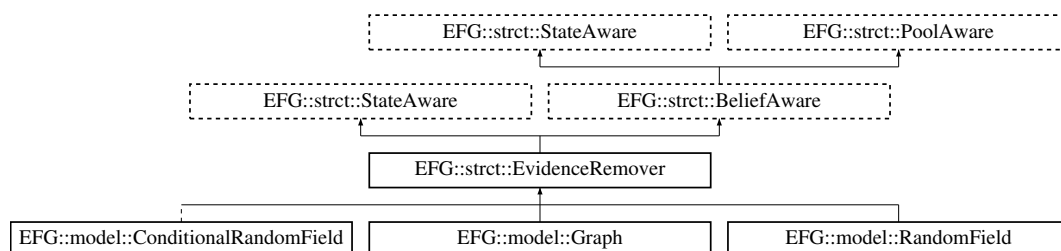
Additional Inherited Members

The documentation for this class was generated from the following file:

- /home/andrea/Scrivania/GitProj/EFG-refactoring/src/header/EasyFactorGraph/structure/SpecialFactors.h

9.16 EFG::strct::EvidenceRemover Class Reference

Inheritance diagram for EFG::strct::EvidenceRemover:



Public Member Functions

- void [removeEvidence](#) (const categoric::VariablePtr &variable)
update the evidence set by removing the specified variable.
- void [removeEvidence](#) (const std::string &variable)
similar to [removeEvidence\(const categoric::VariablePtr &\)](#), but passing the variable name, which is internally searched.
- void [removeEvidences](#) (const categoric::VariablesSet &variables)
update the evidence set by removing all the specified variables.
- void [removeEvidences](#) (const std::unordered_set< std::string > &variables)
similar to [removeEvidences\(const categoric::VariablesSet &\)](#), but passing the variable names, which are internally searched.
- void [removeAllEvidences](#) ()
removes all the evidences currently set for this model.

Additional Inherited Members

9.16.1 Member Function Documentation

9.16.1.1 removeEvidence()

```
void EFG::strct::EvidenceRemover::removeEvidence (
    const categoric::VariablePtr & variable )
```

update the evidence set by removing the specified variable.

Parameters

<i>the</i>	involved variable
------------	-------------------

Exceptions

<i>in</i>	case the passed variable is not part of the model.
<i>in</i>	case the passed variable is not part of the current evidence set.

9.16.1.2 removeEvidences()

```
void EFG::strct::EvidenceRemover::removeEvidences (
    const categoric::VariablesSet & variables )
```

update the evidence set by removing all the specified variables.

Parameters

<i>the</i>	involved variables
------------	--------------------

Exceptions

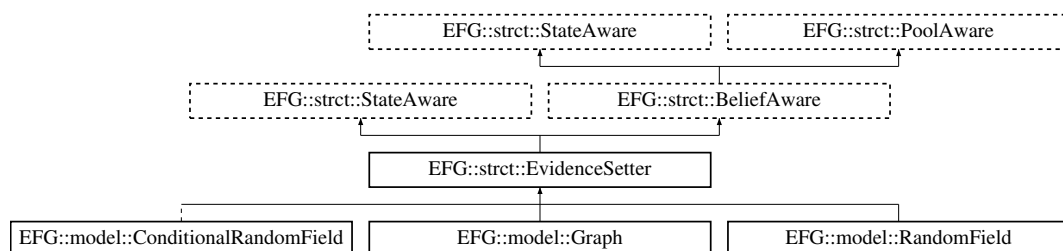
<i>in</i>	case one of the passed variable is not part of the model.
<i>in</i>	case one of the passed variable is not part of the current evidence set.

The documentation for this class was generated from the following file:

- /home/andrea/Scrivania/GitProj/EFG-refactoring/src/header/EasyFactorGraph/structure/EvidenceManager.↔
h

9.17 EFG::strct::EvidenceSetter Class Reference

Inheritance diagram for EFG::strct::EvidenceSetter:



Public Member Functions

- void [setEvidence](#) (const categoric::VariablePtr &variable, std::size_t value)
update the evidence set with the specified new evidence. In case the involved variable was already part of the evidence set, the evidence value is simply updated. On the contrary case, the involved variable is moved into the evidence set, with the specified value.
- void [setEvidence](#) (const std::string &variable, std::size_t value)
Similar to [setEvidence\(const categoric::VariablePtr &, const std::size_t\)](#) , but passing the variable name, which is internally searched.

Additional Inherited Members

9.17.1 Member Function Documentation

9.17.1.1 [setEvidence\(\)](#)

```
void EFG::strct::EvidenceSetter::setEvidence (
    const categoric::VariablePtr & variable,
    std::size_t value )
```

update the evidence set with the specified new evidence. In case the involved variable was already part of the evidence set, the evidence value is simply updated. On the contrary case, the involved variable is moved into the evidence set, with the specified value.

Parameters

<i>the</i>	involved variable
<i>the</i>	evidence value

Exceptions

<i>in</i>	case the passed variable is not part of the model.
-----------	--

The documentation for this class was generated from the following file:

- [/home/andrea/Scrivania/GitProj/EFG-refactoring/src/header/EasyFactorGraph/structure/EvidenceManager.↔h](#)

9.18 EFG::io::json::Exporter Class Reference

Static Public Member Functions

- `template<typename Model >`
static nlohmann::json [exportToJson](#) (const Model &model)
exports the model (variables and factors) into a json.
- `template<typename Model >`
static void [exportToFile](#) (const Model &model, const std::filesystem::path &file_path)
exports the model (variables and factors) into an json file

9.18.1 Member Function Documentation

9.18.1.1 exportToFile()

```
template<typename Model >
static void EFG::io::json::Exporter::exportToFile (
    const Model & model,
    const std::filesystem::path & file_path ) [inline], [static]
```

exports the model (variables and factors) into an json file

Parameters

<i>the</i>	model to export
<i>the</i>	file to generate storing the exported json

9.18.1.2 exportToJson()

```
template<typename Model >
static nlohmann::json EFG::io::json::Exporter::exportToJson (
    const Model & model ) [inline], [static]
```

exports the model (variables and factors) into a json.

Parameters

<i>the</i>	model to export
------------	-----------------

The documentation for this class was generated from the following file:

- /home/andrea/Scrivania/GitProj/EFG-refactoring/src/header/EasyFactorGraph/io/json/Exporter.h

9.19 EFG::io::xml::Exporter Class Reference

Static Public Member Functions

- template<typename Model >
static std::string [exportToString](#) (const Model &model, const std::string &model_name)
exports the model (variables and factors) into a string, describing an xml.
- template<typename Model >
static void [exportToFile](#) (const Model &model, const [ExportInfo](#) &info)
exports the model (variables and factors) into an xml file

9.19.1 Member Function Documentation

9.19.1.1 exportToFile()

```
template<typename Model >
static void EFG::io::xml::Exporter::exportToFile (
    const Model & model,
    const ExportInfo & info ) [inline], [static]
```

exports the model (variables and factors) into an xml file

Parameters

<i>the</i>	model to export
<i>info</i>	describing the xml to generate.

9.19.1.2 exportToString()

```
template<typename Model >
static std::string EFG::io::xml::Exporter::exportToString (
    const Model & model,
    const std::string & model_name ) [inline], [static]
```

exports the model (variables and factors) into a string, describing an xml.

Parameters

<i>the</i>	model to export
<i>the</i>	model name to report in the xml

The documentation for this class was generated from the following file:

- /home/andrea/Scrivania/GitProj/EFG-refactoring/src/header/EasyFactorGraph/io/xml/Exporter.h

9.20 EFG::io::xml::ExportInfo Struct Reference

Public Attributes

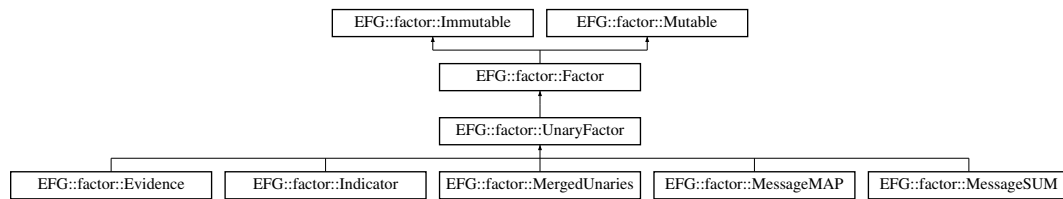
- std::string **file_path**
- std::string **model_name** = "Model"

The documentation for this struct was generated from the following file:

- /home/andrea/Scrivania/GitProj/EFG-refactoring/src/header/EasyFactorGraph/io/xml/Exporter.h

9.21 EFG::factor::Factor Class Reference

Inheritance diagram for EFG::factor::Factor:



Classes

- struct [CloneTrasformedImagesTag](#)
- struct [SimplyAntiCorrelatedTag](#)
- struct [SimplyCorrelatedTag](#)

Public Member Functions

- **Factor** (const [Factor](#) &o)
- **Factor** (const [categoric::Group](#) &vars)

The variables set representing this factor is assumed equal to the passed one. No combinations is instanciated, implicitly assuming all the images equal to 0.
- **Factor** (const [Immutable](#) &to_clone, [CloneTrasformedImagesTag](#))

The variables set representing this factor is copied from the passed one. All the combinations instanciated in the passed factor, are copied in the combinations map of the factor to build, assigning the images values obtained by evaluating the passed factor.
- **Factor** (const [categoric::Group](#) &vars, [SimplyCorrelatedTag](#))

A simply correlating factor is built. All the variables in the passed group should have the same size. The images pertaining to the combinations having all values equal, are assumed equal to 1, all the others to 0. For instance assume to pass a variable set equal to: {<A: size 3>, <B: size 3>, <C: size 3>}. Then, the following combinations map is built: <0,0,0> -> 1 <0,0,1> -> 0 <0,0,2> -> 0.
- **Factor** (const [categoric::Group](#) &vars, [SimplyAntiCorrelatedTag](#))

Similar to [Factor](#)(const [categoric::Group](#) &, const [UseSimpleCorrelation](#) &), but considering a simple anti-correlation. Therefore, to all combinations having all equal values, an image equal to 0 is assigned. All the other ones, are assigned a value equal to 1. For instance assume to pass a variable set equal to: {<A: size 2>, <B: size 2>}. Then, the following combinations map is built: <0,0,0> -> 0 <0,0,1> -> 1 <0,0,2> -> 1.
- **Factor** (const std::vector< const [Immutable](#) * > &factors)
- **Factor** [cloneWithPermutedGroup](#) (const [categoric::Group](#) &new_order) const

Generates a [Factor](#) similar to this one, permuting the group of variables.

Static Public Member Functions

- template<typename... Immutables>
static [Factor](#) [merge](#) (const [Immutable](#) &first, const [Immutable](#) &second, const Immutables &...others)

Builds the factor by merging all the passed factors. The variables set representing this factor is obtained as the union of the all the variables sets of the passed distribution.

Protected Member Functions

- **Factor** (FunctionPtr data)

9.21.1 Constructor & Destructor Documentation

9.21.1.1 Factor() [1/2]

```
EFG::factor::Factor::Factor (
    const categorical::Group & vars,
    SimplyCorrelatedTag )
```

A simply correlating factor is built. All the variables in the passed group should have the same size. The images pertaining to the combinations having all values equal, are assumed equal to 1, all the others to 0. For instance assume to pass a variable set equal to: {<A: size 3>, <B: size 3>, <C: size 3>}. Then, the following combinations map is built: <0,0,0> -> 1 <0,0,1> -> 0 <0,0,2> -> 0.

<0,1,0> -> 0 <0,1,1> -> 0 <0,1,2> -> 0

<0,2,0> -> 0 <0,2,1> -> 0 <0,2,2> -> 0

<1,0,0> -> 0 <1,0,1> -> 0 <1,0,2> -> 0

<1,1,0> -> 0 <1,1,1> -> 1 <1,1,2> -> 0

<1,2,0> -> 0 <1,2,1> -> 0 <1,2,2> -> 0

<2,0,0> -> 0 <2,0,1> -> 0 <2,0,2> -> 0

<2,1,0> -> 0 <2,1,1> -> 0 <2,1,2> -> 0

<2,2,0> -> 0 <2,2,1> -> 0 <2,2,2> -> 1

9.21.1.2 Factor() [2/2]

```
EFG::factor::Factor::Factor (
    const categorical::Group & vars,
    SimplyAntiCorrelatedTag )
```

Similar to [Factor](#)(const [categorical::Group](#) &, const UseSimpleCorrelation &), but considering a simple anti-correlation. Therefore, to all combinations having all equal values, an image equal to 0 is assigned. All the other ones, are assigned a value equal to 1. For instance assume to pass a variable set equal to: {<A: size 2>, <B: size 2>}. Then, the following combinations map is built: <0,0,0> -> 0 <0,0,1> -> 1 <0,0,2> -> 1.

<0,1,0> -> 1 <0,1,1> -> 1 <0,1,2> -> 1

<0,2,0> -> 1 <0,2,1> -> 1 <0,2,2> -> 1

<1,0,0> -> 1 <1,0,1> -> 1 <1,0,2> -> 1

<1,1,0> -> 1 <1,1,1> -> 0 <1,1,2> -> 1

<1,2,0> -> 1 <1,2,1> -> 1 <1,2,2> -> 1

<2,0,0> -> 1 <2,0,1> -> 1 <2,0,2> -> 1

<2,1,0> -> 1 <2,1,1> -> 1 <2,1,2> -> 1

<2,2,0> -> 1 <2,2,1> -> 1 <2,2,2> -> 0

9.21.2 Member Function Documentation

9.21.2.1 cloneWithPermutedGroup()

```
Factor EFG::factor::Factor::cloneWithPermutedGroup (
    const categorical::Group & new_order ) const
```

Generates a [Factor](#) similar to this one, permuting the group of variables.

Parameters

<i>the</i>	new variables group order to assume
------------	-------------------------------------

Returns

the permuted variables factor

Exceptions

<i>in</i>	case new_order.getVariablesSet() != this->getVariables().getVariablesSet()
-----------	--

The documentation for this class was generated from the following file:

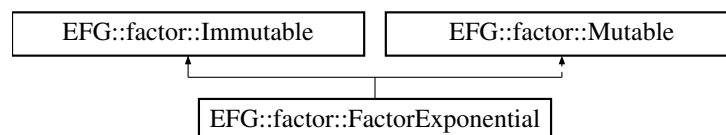
- /home/andrea/Scrivania/GitProj/EFG-refactoring/src/header/EasyFactorGraph/factor/Factor.h

9.22 EFG::factor::FactorExponential Class Reference

An exponential factor applies an exponential function to map the raw values inside the combination map to the actual images. More precisely, given the weight w characterizing the factor the image value is obtained in this way: $\text{image} = \exp(w * \text{raw_value})$

```
#include <FactorExponential.h>
```

Inheritance diagram for EFG::factor::FactorExponential:



Public Member Functions

- **FactorExponential** (const **Factor** &factor, float weighth)

The same variables describing the passed factor are assumed for the object to build. The map of combinations is built by iterating all the possible ones of the group of variables describing the passed factor. The raw values are computed by evaluating the passed factor over each possible combination.
- **FactorExponential** (const **Factor** &factor)

*Same as **FactorExponential**(const **Factor** &, float), assuming weighth = 1.*
- **FactorExponential** (const **FactorExponential** &o)
- void **setWeight** (float w)

sets the weight used by the exponential evaluator.
- float **getWeight** () const
- void **replaceVariables** (const categoric::VariablesSoup &new_variables)

Replaces the variables this distribution should refer to.

Protected Member Functions

- **FactorExponential** (FunctionPtr data)

9.22.1 Detailed Description

An exponential factor applies an exponential function to map the raw values inside the combination map to the actual images. More precisely, given the weight w characterizing the factor the image value is obtained in this way:
 $\text{image} = \exp(w * \text{raw_value})$

All the combinations are instanciated in the combinations map when building this object.

9.22.2 Constructor & Destructor Documentation

9.22.2.1 FactorExponential()

```
EFG::factor::FactorExponential::FactorExponential (
    const Factor & factor,
    float weight )
```

The same variables describing the passed factor are assumed for the object to build. The map of combinations is built by iterating all the possible ones of the group of variables describing the passed factor. The raw values are computed by evaluating the passed factor over each possible combination.

Parameters

<i>the</i>	baseline factor
<i>the</i>	weight that will be considered by the exponential evaluator

9.22.3 Member Function Documentation

9.22.3.1 getWeight()

```
float EFG::factor::FactorExponential::getWeight ( ) const
```

Returns

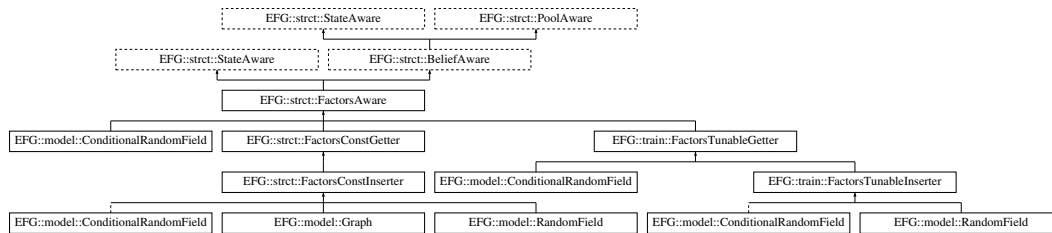
the weight used by the exponential evaluator.

The documentation for this class was generated from the following file:

- /home/andrea/Scrivania/GitProj/EFG-refactoring/src/header/EasyFactorGraph/factor/FactorExponential.h

9.23 EFG::strct::FactorsAware Class Reference

Inheritance diagram for EFG::strct::FactorsAware:



Public Member Functions

- const auto & [getAllFactors](#) () const

Protected Member Functions

- void [addDistribution](#) (const EFG::factor::ImmutablePtr &distribution)

9.23.1 Member Function Documentation

9.23.1.1 getAllFactors()

```
const auto& EFG::strct::FactorsAware::getAllFactors ( ) const [inline]
```

Returns

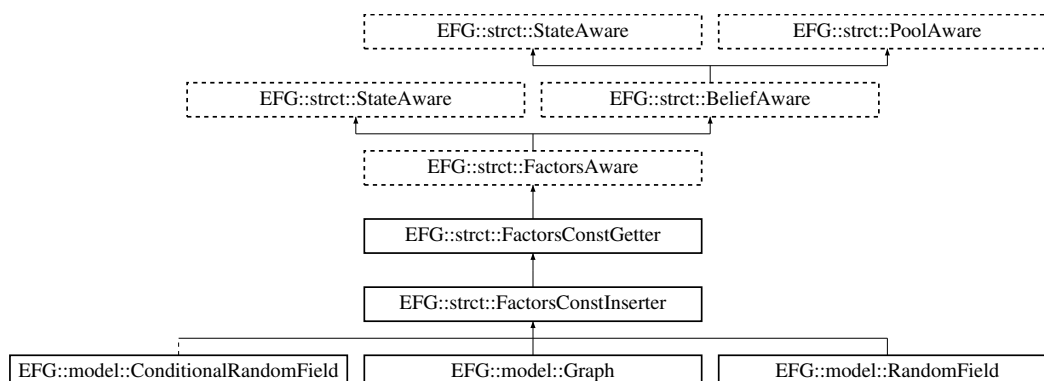
all the factors in the model, tunable and constant.

The documentation for this class was generated from the following file:

- /home/andrea/Scrivania/GitProj/EFG-refactoring/src/header/EasyFactorGraph/structure/bases/FactorsAware.h

9.24 EFG::strct::FactorsConstGetter Class Reference

Inheritance diagram for EFG::strct::FactorsConstGetter:



Public Member Functions

- `const auto & getConstFactors () const`

Protected Attributes

- `std::unordered_set< factor::ImmutablePtr > const_factors`

Additional Inherited Members

9.24.1 Member Function Documentation

9.24.1.1 getConstFactors()

```
const auto& EFG::strct::FactorsConstGetter::getConstFactors ( ) const [inline]
```

Returns

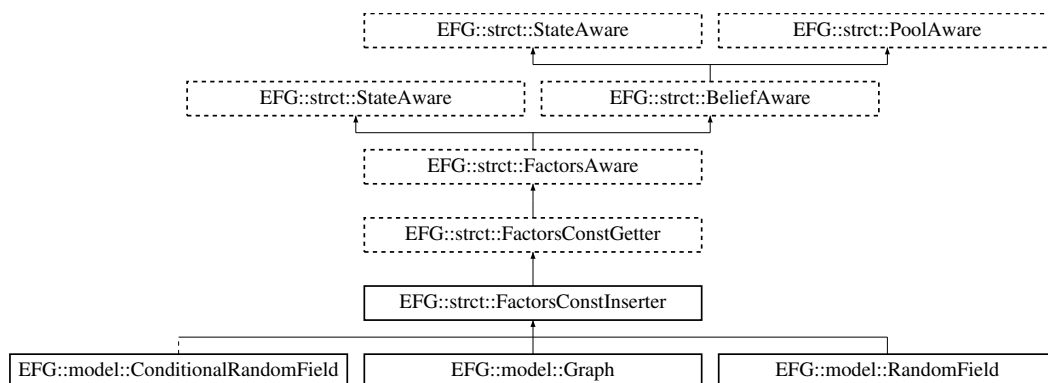
the collection of const factors that are part of the model. Tunable factors are not accounted in this collection.

The documentation for this class was generated from the following file:

- `/home/andrea/Scrivania/GitProj/EFG-refactoring/src/header/EasyFactorGraph/structure/FactorsConstManager.h`

9.25 EFG::strct::FactorsConstInserter Class Reference

Inheritance diagram for EFG::strct::FactorsConstInserter:



Public Member Functions

- void **addConstFactor** (const factor::ImmutablePtr &factor)
add a shallow copy of the passed const factor to this model
- void **copyConstFactor** (const factor::Immutable &factor)
add a deep copy of the passed const factor to this model
- template<typename DistributionIt >
void **absorbConstFactors** (const DistributionIt &begin, const DistributionIt &end, bool copy)
adds a collection of const factors ot this model. Passing copy = true, deep copies are created and inserted in this model. Passing copy = false, shallow copies are created and inserted in this model.

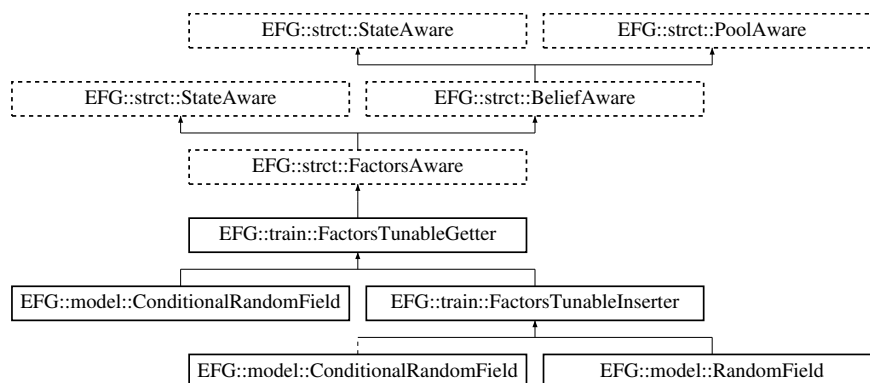
Additional Inherited Members

The documentation for this class was generated from the following file:

- /home/andrea/Scrivania/GitProj/EFG-refactoring/src/header/EasyFactorGraph/structure/FactorsConstManager.h

9.26 EFG::train::FactorsTunableGetter Class Reference

Inheritance diagram for EFG::train::FactorsTunableGetter:



Public Member Functions

- const std::unordered_set< FactorExponentialPtr > & **getTunableFactors** () const
Get the collections of tunable exponential factors.
- std::vector< std::variant< FactorExponentialPtr, TunableClusters > > **getTunableClusters** () const
Get the clusters of tunable exponential factors. Elements in the same cluster, shares the weight.
- std::vector< float > **getWeights** () const
- void **setWeights** (const std::vector< float > &weights)
- std::vector< float > **getWeightsGradient** (const TrainSet::Iterator &train_set_combinations, std::size_t threads=1)

Protected Member Functions

- virtual std::vector< float > **getWeightsGradient_** (const TrainSet::Iterator &train_set_combinations)=0

Protected Attributes

- `std::unordered_set< FactorExponentialPtr >` **tunable_factors**
- Tuners **tuners**

9.26.1 Member Function Documentation

9.26.1.1 `getWeights()`

```
std::vector<float> EFG::train::FactorsTunableGetter::getWeights ( ) const
```

Returns

the weights of all the tunable factors that are part of the model. The same order assumed by [getTunableClusters\(\)](#) is assumed.

9.26.1.2 `getWeightsGradient()`

```
std::vector<float> EFG::train::FactorsTunableGetter::getWeightsGradient (
    const TrainSet::Iterator & train_set_combinations,
    std::size_t threads = 1 )
```

Returns

the gradients of the weights of all the tunable factors that are part of the model, w.r.t a certain training set. The same order assumed by [getTunableClusters\(\)](#) is assumed.

Parameters

<i>the</i>	training set to use
<i>the</i>	number of threads to use for the gradient computation

9.26.1.3 `setWeights()`

```
void EFG::train::FactorsTunableGetter::setWeights (
    const std::vector< float > & weights )
```

Returns

sets the weights to use for of all the tunable factors that are part of the model. The same order assumed by [getTunableClusters\(\)](#) should be assumed.

Exceptions

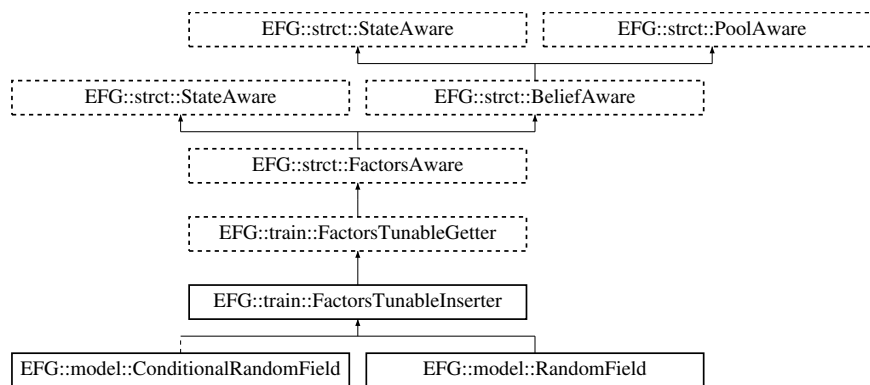
<i>in</i>	case the number of specified weights is inconsistent
-----------	--

The documentation for this class was generated from the following file:

- `/home/andrea/Scrivania/GitProj/EFG-refactoring/src/header/EasyFactorGraph/trainable/FactorsTunableManager.h`

9.27 EFG::train::FactorsTunableInserter Class Reference

Inheritance diagram for EFG::train::FactorsTunableInserter:



Public Member Functions

- void [addTunableFactor](#) (const FactorExponentialPtr &factor, const std::optional< categoric::VariablesSet > &group_sharing_weight=std::nullopt)
add a shallow copy of the passed tunable expoential factor to this model.
- void [copyTunableFactor](#) (const factor::FactorExponential &factor, const std::optional< categoric::VariablesSet > &group_sharing_weight=std::nullopt)
add a deep copy of the passed tunable expoential factor to this model.
- template<typename FactorExponentialIt >
void [absorbTunableFactors](#) (const FactorExponentialIt &begin, const FactorExponentialIt &end, bool copy)
adds a collection of tunable expoential factors ot this model. Passing copy = true, deep copies are created and inserted in this model. Passing copy = false, shallow copies are created and inserted in this model.
- void [absorbTunableClusters](#) (const FactorsTunableGetter &source, bool copy)
adds a collection of tunable expoential factors ot this model, preserving the fact that elements in the same cluster should share the weight. Passing copy = true, deep copies are created and inserted in this model. Passing copy = false, shallow copies are created and inserted in this model.

Protected Member Functions

- Tuners::iterator [findTuner](#) (const categoric::VariablesSet &tuned_vars_group)

Additional Inherited Members

9.27.1 Member Function Documentation

9.27.1.1 addTunableFactor()

```
void EFG::train::FactorsTunableInserter::addTunableFactor (
    const FactorExponentialPtr & factor,
    const std::optional< categoric::VariablesSet > & group_sharing_weight = std::↵
:nullopt )
```

add a shallow copy of the passed tunable exponential factor to this model.

Parameters

<i>the</i>	factor to insert
<i>an</i>	optional group of variables specifying the tunable factor that should share the weight with the one to insert. When passing a <code>nullopt</code> the factor will be inserted without sharing its weight.

9.27.1.2 copyTunableFactor()

```
void EFG::train::FactorsTunableInserter::copyTunableFactor (
    const factor::FactorExponential & factor,
    const std::optional< categoric::VariablesSet > & group_sharing_weight = std::↵
:nullopt )
```

add a deep copy of the passed tunable exponential factor to this model.

Parameters

<i>the</i>	factor to insert
<i>an</i>	optional group of variables specifying the tunable factor that should share the weight with the one to insert. When passing a <code>nullopt</code> the factor will be inserted without sharing its weight.

The documentation for this class was generated from the following file:

- `/home/andrea/Scrivania/GitProj/EFG-refactoring/src/header/EasyFactorGraph/trainable/FactorsTunable↵
Manager.h`

9.28 EFG::factor::Function Class Reference

Classes

- struct [CombinationHasher](#)
- struct [Info](#)

Public Member Functions

- **Function** (const [categoric::Group](#) &variables)
- void **cloneImages** (const [Function](#) &o)
- void **set** (const std::vector< std::size_t > &combination, float image)
- float **findTransformed** (const std::vector< std::size_t > &combination) const
- float **findImage** (const std::vector< std::size_t > &combination) const
- void **clear** ()
- template<bool UseTransformed, typename Pred >
void **forEachCombination** (Pred &&pred) const
- template<bool UseTransformed, typename Pred >
void **forEachNonNullCombination** (Pred &&pred) const
- const [Info](#) & **getInfo** () const
- const auto & **vars** () const
- auto & **vars** ()

Protected Types

- using **SparseContainer** = std::unordered_map< std::vector< std::size_t >, float, [CombinationHasher](#) >
- using **DenseContainer** = std::vector< float >

Protected Member Functions

- virtual float [transform](#) (float input) const
applies a specific function to obtain the image from the passed rwa value
- SparseContainer **makeSparseContainer** ()

Protected Attributes

- [categoric::Group](#) **variables_**
- std::shared_ptr< const [Info](#) > **info**
- std::variant< SparseContainer, DenseContainer > **data_**

9.28.1 Member Function Documentation

9.28.1.1 transform()

```
virtual float EFG::factor::Function::transform (
    float input ) const [inline], [protected], [virtual]
```

applies a specific function to obtain the image from the passed rwa value

Parameters

<i>the</i>	raw value to convert
------------	----------------------

Returns

the converted image

The documentation for this class was generated from the following file:

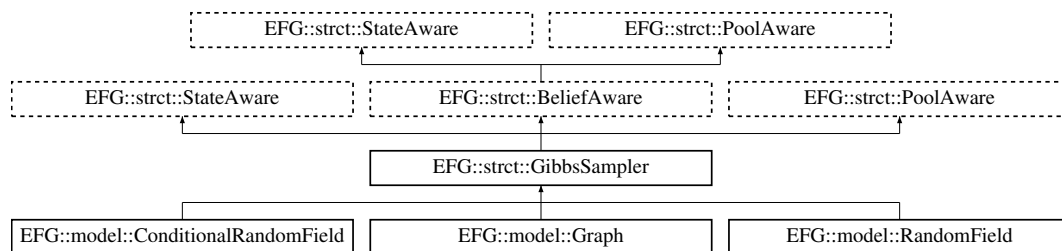
- /home/andrea/Scrivania/GitProj/EFG-refactoring/src/header/EasyFactorGraph/factor/Function.h

9.29 EFG::strct::GibbsSampler Class Reference

Refer also to https://en.wikipedia.org/wiki/Gibbs_sampling.

```
#include <GibbsSampler.h>
```

Inheritance diagram for EFG::strct::GibbsSampler:

**Classes**

- struct [SamplerNode](#)
- struct [SamplesGenerationContext](#)

Public Member Functions

- `std::vector< std::vector< std::size_t > > makeSamples (const SamplesGenerationContext &context, std::size_t threads=1)`

Use Gibbs sampling approach to draw empirical samples. Values inside the returned combination are ordered with the same order used for the variables returned by [getAllVariables\(\)](#).

Additional Inherited Members

9.29.1 Detailed Description

Refer also to https://en.wikipedia.org/wiki/Gibbs_sampling.

9.29.2 Member Function Documentation

9.29.2.1 makeSamples()

```
std::vector<std::vector<std::size_t> > EFG::strct::GibbsSampler::makeSamples (
    const SamplesGenerationContext & context,
    std::size_t threads = 1 )
```

Use Gibbs sampling approach to draw empirical samples. Values inside the returned combination are ordered with the same order used for the variables returned by [getAllVariables\(\)](#).

In case some evidences are set, their values will appear as is in the sampled combinations.

Parameters

<i>number</i>	parameters for the samples generation
<i>number</i>	of threads to use for the samples generation

The documentation for this class was generated from the following file:

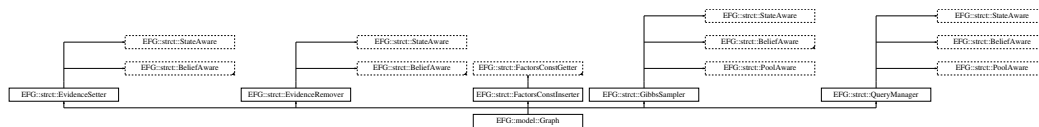
- /home/andrea/Scrivania/GitProj/EFG-refactoring/src/header/EasyFactorGraph/structure/GibbsSampler.h

9.30 EFG::model::Graph Class Reference

A simple graph object, that stores only const factors. Evidences may be changed over the time.

```
#include <Graph.h>
```

Inheritance diagram for EFG::model::Graph:



Public Member Functions

- **Graph** (const [Graph](#) &o)
 - [Graph](#) & **operator=** (const [Graph](#) &)=delete
 - void **absorb** (const [struct::FactorsAware](#) &to_absorb, bool copy)
- Gather all the factors (tunable and constant) of another model and insert/copy them into this object.*

Additional Inherited Members

9.30.1 Detailed Description

A simple graph object, that stores only const factors. Evidences may be changed over the time.

9.30.2 Member Function Documentation

9.30.2.1 absorb()

```
void EFG::model::Graph::absorb (
    const struct::FactorsAware & to_absorb,
    bool copy )
```

Gather all the factors (tunable and constant) of another model and insert/copy them into this object.

Parameters

<i>the</i>	model whose factors should be inserted/copied
<i>when</i>	passing true the factors are deep copied, while in the contrary case shallow copies of the smart pointers are inserted into this model.

The documentation for this class was generated from the following file:

- /home/andrea/Scrivania/GitProj/EFG-refactoring/src/header/EasyFactorGraph/model/Graph.h

9.31 EFG::strct::StateAware::GraphState Struct Reference

Public Attributes

- categoric::VariablesSoup **variables**
- Nodes **nodes**
- HiddenClusters **clusters**
- Evidences **evidences**

The documentation for this struct was generated from the following file:

- /home/andrea/Scrivania/GitProj/EFG-refactoring/src/header/EasyFactorGraph/structure/bases/StateAware.h

9.32 EFG::categoric::Group Class Reference

An ensemble of categoric variables. Each variable in the ensemble should have its own unique name.

```
#include <Group.h>
```

Public Member Functions

- [Group](#) (const VariablesSoup &group)
- [Group](#) (const VariablePtr &var)
- template<typename... Vars>
[Group](#) (const VariablePtr &varA, const VariablePtr &varB, const Vars &...vars)
- void [replaceVariables](#) (const VariablesSoup &new_variables)
replaces the group of variables.
- bool **operator==** (const [Group](#) &o) const
- void [add](#) (const VariablePtr &var)
- std::size_t [size](#) () const
- const VariablesSoup & [getVariables](#) () const
- const VariablesSet & [getVariablesSet](#) () const

Protected Attributes

- VariablesSoup **group**
- VariablesSet **group_sorted**

9.32.1 Detailed Description

An ensemble of categoric variables. Each variable in the ensemble should have its own unique name.

9.32.2 Constructor & Destructor Documentation

9.32.2.1 Group() [1/3]

```
EFG::categoric::Group::Group (
    const VariablesSoup & group )
```

Parameters

<i>the</i>	initial variables of the group
------------	--------------------------------

Exceptions

<i>when</i>	passing an empty collection
<i>when</i>	passing a collection containing multiple times a certain variable

9.32.2.2 Group() [2/3]

```
EFG::categoric::Group::Group (
    const VariablePtr & var )
```

Parameters

<i>the</i>	initial variable to put in the group
------------	--------------------------------------

9.32.2.3 Group() [3/3]

```
template<typename... Vars>
EFG::categoric::Group::Group (
```



```
const VariablePtr & varA,
const VariablePtr & varB,
const Vars &... vars ) [inline]
```

Parameters

<i>the</i>	first initial variable to put in the group
<i>the</i>	second initial variable to put in the group
<i>all</i>	the other initial variables

Exceptions

<i>when</i>	passing a collection containing multiple times a certain
-------------	--

9.32.3 Member Function Documentation**9.32.3.1 add()**

```
void EFG::categoric::Group::add (
    const VariablePtr & var )
```

Parameters

<i>the</i>	variable to add in the group
------------	------------------------------

Exceptions

<i>in</i>	case a variable with the same name is already part of the group
-----------	---

9.32.3.2 getVariables()

```
const VariablesSoup& EFG::categoric::Group::getVariables ( ) const [inline]
```

Returns

the ensamble of variables as an unsorted collection

9.32.3.3 getVariablesSet()

```
const VariablesSet& EFG::categoric::Group::getVariablesSet ( ) const [inline]
```

Returns

the ensemble of variables as a sorted collection

9.32.3.4 replaceVariables()

```
void EFG::categoric::Group::replaceVariables (
    const VariablesSoup & new_variables )
```

replaces the group of variables.

Exceptions

<i>In</i>	case of size mismatch with the previous variables set: the sizes of the 2 groups should be the same and the elements in the same positions must have the same domain size
-----------	---

9.32.3.5 size()

```
std::size_t EFG::categoric::Group::size ( ) const
```

Returns

the size of the joint domain of the group. For example the group <A,B,C> with sizes <2,4,3> will have a joint domain of size $2 \times 4 \times 3 = 24$

The documentation for this class was generated from the following file:

- /home/andrea/Scrivania/GitProj/EFG-refactoring/src/header/EasyFactorGraph/categoric/Group.h

9.33 EFG::categoric::GroupRange Class Reference

This object allows to iterate all the elements in the joint domain of a group of variables, without precomputing all the elements in such domain. For example when having a domain made by variables = { A (size = 2), B (size = 3), C (size = 2) }, the elements in the joint domain that will be iterated are: <0,0,0> <0,0,1> <0,1,0> <0,1,1> <0,2,0> <0,2,1> <1,0,0> <1,0,1> <1,1,0> <1,1,1> <1,2,0> <1,2,1> After construction, the Range object starts to point to the first element in the joint domain <0,0,...>. Then, when incrementing the object, the following element is pointed. When calling get() the current pointed element can be accessed.

```
#include <GroupRange.h>
```

Public Types

- using **iterator_category** = std::input_iterator_tag
- using **value_type** = std::vector< std::size_t >
- using **pointer** = const std::vector< std::size_t > *
- using **reference** = const std::vector< std::size_t > &

Public Member Functions

- **GroupRange** (const std::vector< std::size_t > &sizes)
- **GroupRange** (const **Group** &variables)
- **GroupRange** (const **GroupRange** &o)
- pointer **operator->** () const
- reference **operator*** () const
- **GroupRange** & **operator++** ()
Make the object to point to the next element in the joint domain.
- bool **isEqual** (const **GroupRange** &o) const

Static Public Member Functions

- static **GroupRange** **end** ()

9.33.1 Detailed Description

This object allows to iterate all the elements in the joint domain of a group of variables, without precomputing all the elements in such domain. For example when having a domain made by variables = { A (size = 2), B (size = 3), C (size = 2) }, the elements in the joint domain that will be iterated are: <0,0,0> <0,0,1> <0,1,0> <0,1,1> <0,2,0> <0,2,1> <1,0,0> <1,0,1> <1,1,0> <1,1,1> <1,2,0> <1,2,1> After construction, the Range object starts to point to the first element in the joint domain <0,0,...>. Then, when incrementing the object, the following element is pointed. When calling get() the current pointed element can be accessed.

This object should be recognized by the compiler as an stl iterator.

9.33.2 Constructor & Destructor Documentation

9.33.2.1 GroupRange()

```
EFG::categoric::GroupRange::GroupRange (
    const Group & variables )
```

Parameters

<i>the</i>	group of variables whose joint domain must be iterated
------------	--

9.33.3 Member Function Documentation

9.33.3.1 operator++()

`GroupRange& EFG::categoric::GroupRange::operator++ ()`

Make the object to point to the next element in the joint domain.

Exceptions

<i>if</i>	the current pointed element is the last one.
-----------	--

The documentation for this class was generated from the following file:

- `/home/andrea/Scrivania/GitProj/EFG-refactoring/src/header/EasyFactorGraph/categoric/GroupRange.h`

9.34 std::hash< EFG::categoric::Variable > Struct Reference

Public Member Functions

- `std::size_t operator() (const EFG::categoric::Variable &subject) const`

The documentation for this struct was generated from the following file:

- `/home/andrea/Scrivania/GitProj/EFG-refactoring/src/header/EasyFactorGraph/categoric/Variable.h`

9.35 EFG::Hasher< T > Struct Template Reference

Public Member Functions

- `std::size_t operator() (const std::shared_ptr< T > &subject) const`

The documentation for this struct was generated from the following file:

- `/home/andrea/Scrivania/GitProj/EFG-refactoring/src/header/EasyFactorGraph/misc/SmartPointerUtils.h`

9.36 EFG::strct::HiddenCluster Struct Reference

Clusters of hidden node. Each cluster is a group of connected hidden nodes. Nodes in different clusters are not currently connected, due to the model structure or the kind of evidences currently applied.

```
#include <Types.h>
```

Classes

- struct [TopologyInfo](#)

Public Member Functions

- void **updateConnectivity** ()

Public Attributes

- std::unordered_set< [Node](#) * > **nodes**
- [Cache](#)< std::vector< [TopologyInfo](#) > > **connectivity**

9.36.1 Detailed Description

Clusters of hidden node. Each cluster is a group of connected hidden nodes. Nodes in different clusters are not currently connected, due to the model structure or the kind of evidences currently applied.

The documentation for this struct was generated from the following file:

- /home/andrea/Scrivania/GitProj/EFG-refactoring/src/header/EasyFactorGraph/structure/Types.h

9.37 EFG::factor::ImageFinder Class Reference

An object used to search for the images associated to sub combinations that are part of a bigger one.

```
#include <ImageFinder.h>
```

Public Member Functions

- float **findTransformed** (const std::vector< std::size_t > &comb) const
Searches for matches. For example assume having built this object with a bigger_group equal to <A,B,C,D> while the variables describing the distribution this finder refers to is equal to <B,D>. When passing a comb equal to <0,1,2,0>, this object searches for the immagine associated to the sub combination <B,D> = <1,0>.
- float **findImage** (const std::vector< std::size_t > &comb) const

Friends

- class **Immutable**

9.37.1 Detailed Description

An object used to search for the images associated to sub combinations that are part of a bigger one.

9.37.2 Member Function Documentation

9.37.2.1 findTransformed()

```
float EFG::factor::ImageFinder::findTransformed (
    const std::vector< std::size_t > & comb ) const [inline]
```

Searches for matches. For example assume having built this object with a bigger_group equal to <A,B,C,D> while the variables describing the distribution this finder refers to is equal to <B,D>. When passing a comb equal to <0,1,2,0>, this object searches for the immagine associated to the sub combination <B,D> = <1,0>.

Parameters

<i>the</i>	combination of values referring to the bigger_group, which contains the sub combination to search.
------------	--

Returns

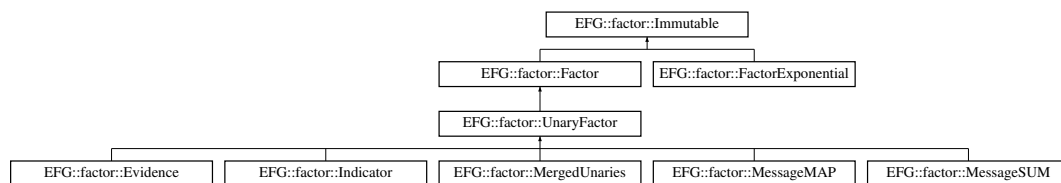
image associated to the passed combination

The documentation for this class was generated from the following file:

- /home/andrea/Scrivania/GitProj/EFG-refactoring/src/header/EasyFactorGraph/factor/ImageFinder.h

9.38 EFG::factor::Immutable Class Reference

Inheritance diagram for EFG::factor::Immutable:



Public Member Functions

- **Immutable** (const [Immutable](#) &o)=delete
- [Immutable](#) & **operator=** (const [Immutable](#) &)=delete
- **Immutable** ([Immutable](#) &&o)=delete
- [Immutable](#) & **operator=** ([Immutable](#) &&)=delete
- const auto & **function** () const
- [ImageFinder](#) **makeFinder** (const categoric::VariablesSoup &bigger_group) const
- std::vector< float > [getProbabilities](#) () const

Protected Member Functions

- **Immutable** (FunctionPtr data)

9.38.1 Member Function Documentation

9.38.1.1 getProbabilities()

```
std::vector<float> EFG::factor::Immutable::getProbabilities ( ) const
```

Returns

the probabilities associated to each combination in the domain, when assuming only the existence of this distribution. Such probabilities are actually the normalized images. The order of returned values, refer to the combinations that can be iterated by [categoric::GroupRange](#) on the variables representing this distribution.

The documentation for this class was generated from the following file:

- /home/andrea/Scrivania/GitProj/EFG-refactoring/src/header/EasyFactorGraph/factor/Immutable.h

9.39 EFG::io::json::Importer Class Reference

Static Public Member Functions

- template<typename Model >
static void [importFromFile](#) (Model &model, const std::filesystem::path &file_path)
imports the structure (variables and factors) described in an xml file and add it to the passed model
- template<typename Model >
static void [importFromJson](#) (Model &model, const nlohmann::json &source)
parse the model (variables and factors) described by the passed json and tries to add its factors to the passed model.

9.39.1 Member Function Documentation

9.39.1.1 importFromFile()

```
template<typename Model >
static void EFG::io::json::Importer::importFromFile (
    Model & model,
    const std::filesystem::path & file_path ) [inline], [static]
```

imports the structure (variables and factors) described in an xml file and add it to the passed model

Parameters

<i>the</i>	model receiving the parsed data
<i>the</i>	path storing the xml to import

9.39.1.2 importFromJson()

```
template<typename Model >
static void EFG::io::json::Importer::importFromJson (
    Model & model,
    const nlohmann::json & source ) [inline], [static]
```

parse the model (variables and factors) described by the passed json and tries to add its factors to the passed model.

Parameters

<i>recipient</i>	of the model parsed from file
<i>json</i>	describing the model to parse and add to the passed one

The documentation for this class was generated from the following file:

- /home/andrea/Scrivania/GitProj/EFG-refactoring/src/header/EasyFactorGraph/io/json/Importer.h

9.40 EFG::io::xml::Importer Class Reference

Static Public Member Functions

- template<typename Model >
static void [importFromFile](#) (Model &model, const std::filesystem::path &file_path)
parse the model (variables and factors) described by the specified file and tries to add its factors to the passed model.

9.40.1 Member Function Documentation

9.40.1.1 importFromFile()

```
template<typename Model >
static void EFG::io::xml::Importer::importFromFile (
    Model & model,
    const std::filesystem::path & file_path ) [inline], [static]
```

parse the model (variables and factors) described by the specified file and tries to add its factors to the passed model.

Parameters

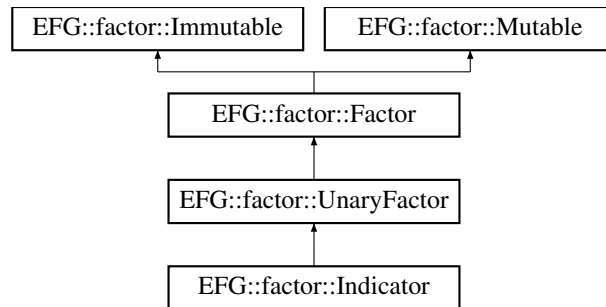
<i>recipient</i>	of the model parsed from file
<i>location</i>	of the model to parse and add to the passed one

The documentation for this class was generated from the following file:

- `/home/andrea/Scrivania/GitProj/EFG-refactoring/src/header/EasyFactorGraph/io/xml/Importer.h`

9.41 EFG::factor::Indicator Class Reference

Inheritance diagram for EFG::factor::Indicator:



Public Member Functions

- **Indicator** (const categoric::VariablePtr &var, std::size_t value)

Additional Inherited Members

The documentation for this class was generated from the following file:

- `/home/andrea/Scrivania/GitProj/EFG-refactoring/src/header/EasyFactorGraph/structure/SpecialFactors.h`

9.42 EFG::factor::Function::Info Struct Reference

Public Attributes

- `std::vector< std::size_t > sizes`
- `std::size_t totCombinations`
- `std::size_t critical_size`

The documentation for this struct was generated from the following file:

- `/home/andrea/Scrivania/GitProj/EFG-refactoring/src/header/EasyFactorGraph/factor/Function.h`

9.43 EFG::train::TrainSet::Iterator Class Reference

an object able to iterate all the combinations that are part of a training set or a sub portion of it.

```
#include <TrainSet.h>
```

Public Member Functions

- [Iterator](#) (const [TrainSet](#) &subject, float percentage)
involved train set
- template<typename Predicate >
void **forEachSample** (const Predicate &pred) const
- std::size_t [size](#) () const

9.43.1 Detailed Description

an object able to iterate all the combinations that are part of a training set or a sub portion of it.

9.43.2 Constructor & Destructor Documentation

9.43.2.1 Iterator()

```
EFG::train::TrainSet::Iterator::Iterator (
    const TrainSet & subject,
    float percentage )
```

involved train set

Parameters

<i>the</i>	percentage of combinations to extract from the passed subject. Passing a value equal to 1, means to use all the combinations of the passed subject.
------------	---

9.43.3 Member Function Documentation

9.43.3.1 size()

```
std::size_t EFG::train::TrainSet::Iterator::size ( ) const
```

Returns

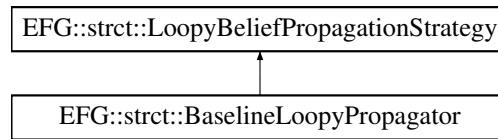
number of combinations considered by this train set iterator.

The documentation for this class was generated from the following file:

- /home/andrea/Scrivania/GitProj/EFG-refactoring/src/header/EasyFactorGraph/trainable/TrainSet.h

9.44 EFG::strct::LoopyBeliefPropagationStrategy Class Reference

Inheritance diagram for EFG::strct::LoopyBeliefPropagationStrategy:



Public Member Functions

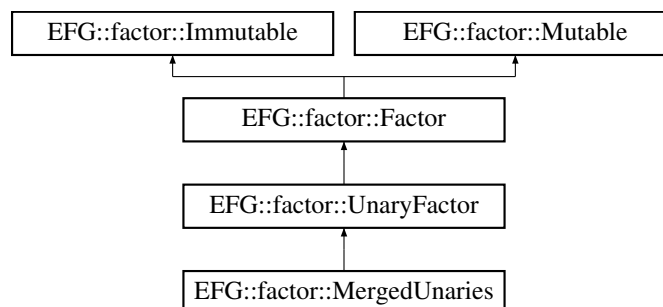
- virtual bool **propagateBelief** ([HiddenCluster](#) &subject, PropagationKind kind, const [PropagationContext](#) &context, [Pool](#) &pool)=0

The documentation for this class was generated from the following file:

- /home/andrea/Scrivania/GitProj/EFG-refactoring/src/header/EasyFactorGraph/structure/bases/BeliefAware.h

9.45 EFG::factor::MergedUnaries Class Reference

Inheritance diagram for EFG::factor::MergedUnaries:



Public Member Functions

- **MergedUnaries** (const categoric::VariablePtr &var)
- **MergedUnaries** (const std::vector< const [Immutable](#) * > &factors)
- void **merge** (const [Immutable](#) &to_merge)
- void **normalize** ()

Additional Inherited Members

The documentation for this class was generated from the following file:

- /home/andrea/Scrivania/GitProj/EFG-refactoring/src/header/EasyFactorGraph/structure/SpecialFactors.h

9.46 EFG::detail::Merger< Separator > Struct Template Reference

Public Member Functions

- `template<typename Last >`
`void join_ (Last &&last)`
- `template<typename First , typename... Args>`
`void join_ (First &&first, Args &&...args)`

Public Attributes

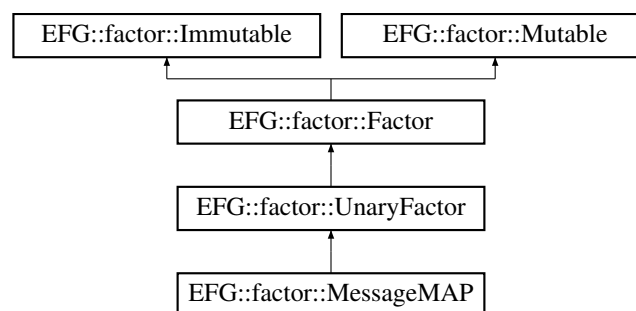
- `std::ostream & recipient`

The documentation for this struct was generated from the following file:

- `/home/andrea/Scrivania/GitProj/EFG-refactoring/src/header/EasyFactorGraph/misc/Strings.h`

9.47 EFG::factor::MessageMAP Class Reference

Inheritance diagram for EFG::factor::MessageMAP:



Public Member Functions

- **MessageMAP** (const [UnaryFactor](#) &merged_unaries, const [Immutable](#) &binary_factor)

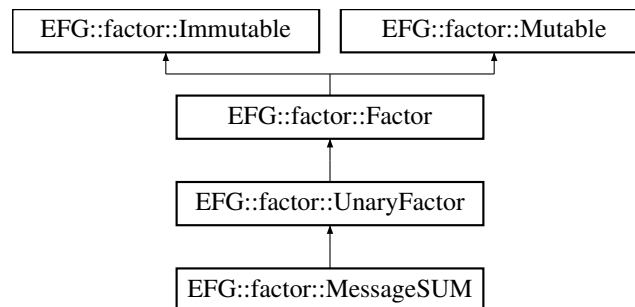
Additional Inherited Members

The documentation for this class was generated from the following file:

- `/home/andrea/Scrivania/GitProj/EFG-refactoring/src/header/EasyFactorGraph/structure/SpecialFactors.h`

9.48 EFG::factor::MessageSUM Class Reference

Inheritance diagram for EFG::factor::MessageSUM:



Public Member Functions

- **MessageSUM** (const [UnaryFactor](#) &merged_unaries, const [Immutable](#) &binary_factor)

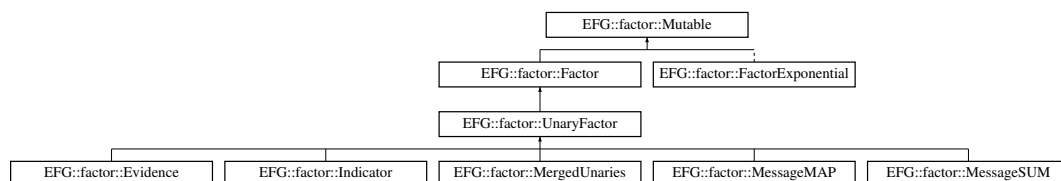
Additional Inherited Members

The documentation for this class was generated from the following file:

- /home/andrea/Scrivania/GitProj/EFG-refactoring/src/header/EasyFactorGraph/structure/SpecialFactors.h

9.49 EFG::factor::Mutable Class Reference

Inheritance diagram for EFG::factor::Mutable:



Public Member Functions

- **Mutable** (const [Mutable](#) &o)=delete
- [Mutable](#) & **operator=** (const [Mutable](#) &)=delete
- **Mutable** ([Mutable](#) &&o)=delete
- [Mutable](#) & **operator=** ([Mutable](#) &&)=delete
- void **set** (const std::vector< std::size_t > &comb, float value)
sets the raw value of the image related to the passed combination. In case the combination is currently not part of the distribution, it is added to the combinations map, with the passed raw image value.
- void **clear** ()
Removes all the combinations from the combinations map.
- void **replaceVariables** (const categoric::VariablesSoup &new_variables)
Replaces the variables this distribution should refer to.

Protected Member Functions

- **Mutable** (FunctionPtr data)
- auto & **functionMutable** ()

9.49.1 Member Function Documentation

9.49.1.1 set()

```
void EFG::factor::Mutable::set (
    const std::vector< std::size_t > & comb,
    float value )
```

sets the raw value of the image related to the passed combination. In case the combination is currently not part of the distribution, it is added to the combinations map, with the passed raw image value.

Parameters

<i>the</i>	combination whose raw image must be set
<i>the</i>	raw image value to assume

Exceptions

<i>passing</i>	a negative number for value
----------------	-----------------------------

The documentation for this class was generated from the following file:

- /home/andrea/Scrivania/GitProj/EFG-refactoring/src/header/EasyFactorGraph/factor/Mutable.h

9.50 EFG::detail::Node< T > Struct Template Reference

Public Attributes

- Predicate< T > **pred_**

The documentation for this struct was generated from the following file:

- /home/andrea/Scrivania/GitProj/EFG-refactoring/src/header/EasyFactorGraph/misc/Visitor.h

9.51 EFG::strct::Node Struct Reference

Classes

- struct [Connection](#)

Public Member Functions

- void **updateMergedUnaries** ()

Static Public Member Functions

- static std::pair< [Connection](#) *, [Connection](#) * > **activate** ([Node](#) &a, [Node](#) &b, factor::ImmutablePtr factor)
- static std::pair< [Connection](#) *, [Connection](#) * > **disable** ([Node](#) &a, [Node](#) &b, factor::ImmutablePtr factor=nullptr)

Public Attributes

- categoric::VariablePtr **variable**
- std::unordered_map< [Node](#) *, [Connection](#) > **active_connections**
- std::unordered_map< [Node](#) *, [Connection](#) > **disabled_connections**
- std::vector< factor::ImmutablePtr > **unary_factors**
- [Cache](#)< const [factor::UnaryFactor](#) > **merged_unaries**

The documentation for this struct was generated from the following file:

- /home/andrea/Scrivania/GitProj/EFG-refactoring/src/header/EasyFactorGraph/structure/Types.h

9.52 EFG::detail::NodeConst< T > Struct Template Reference

Public Attributes

- PredicateConst< T > **pred_**

The documentation for this struct was generated from the following file:

- /home/andrea/Scrivania/GitProj/EFG-refactoring/src/header/EasyFactorGraph/misc/Visitor.h

9.53 EFG::strct::StateAware::NodeLocation Struct Reference

Public Attributes

- [Node](#) * **node**
- std::variant< HiddenClusters::iterator, Evidences::iterator > **location**

The documentation for this struct was generated from the following file:

- /home/andrea/Scrivania/GitProj/EFG-refactoring/src/header/EasyFactorGraph/structure/bases/StateAware.h

9.54 EFG::strct::Pool Class Reference

Public Member Functions

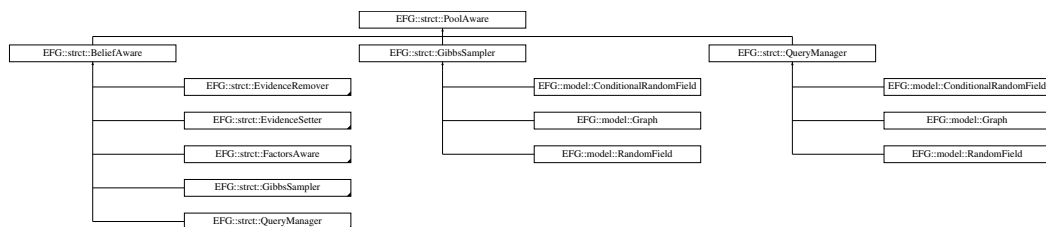
- **Pool** (std::size_t size)
- void **parallelFor** (const Tasks &tasks)
- std::size_t **size** () const

The documentation for this class was generated from the following file:

- /home/andrea/Scrivania/GitProj/EFG-refactoring/src/header/EasyFactorGraph/structure/bases/PoolAware.h

9.55 EFG::strct::PoolAware Class Reference

Inheritance diagram for EFG::strct::PoolAware:



Classes

- class [ScopedPoolActivator](#)

Protected Member Functions

- void **resetPool** ()
- [Pool](#) & **getPool** ()
- void **setPoolSize** (std::size_t new_size)

The documentation for this class was generated from the following file:

- /home/andrea/Scrivania/GitProj/EFG-refactoring/src/header/EasyFactorGraph/structure/bases/PoolAware.h

9.56 EFG::strct::PropagationContext Struct Reference

Public Attributes

- std::size_t [max_iterations_loopy_propagation](#)
maximum number of iterations to use when trying to calibrate a loopy graph

The documentation for this struct was generated from the following file:

- /home/andrea/Scrivania/GitProj/EFG-refactoring/src/header/EasyFactorGraph/structure/Types.h

9.57 EFG::strct::PropagationResult Struct Reference

a structure that can be exposed after having propagated the belief, providing info on the encountered structure.

```
#include <Types.h>
```

Classes

- struct [ClusterInfo](#)

Public Attributes

- PropagationKind **propagation_kind_done**
- bool **was_completed**
- std::vector< [ClusterInfo](#) > **structures**

9.57.1 Detailed Description

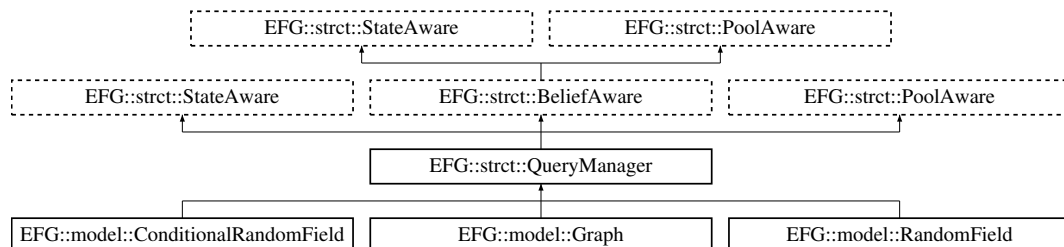
a structure that can be exposed after having propagated the belief, providing info on the encountered structure.

The documentation for this struct was generated from the following file:

- /home/andrea/Scrivania/GitProj/EFG-refactoring/src/header/EasyFactorGraph/structure/Types.h

9.58 EFG::strct::QueryManager Class Reference

Inheritance diagram for EFG::strct::QueryManager:



Public Member Functions

- std::vector< float > [getMarginalDistribution](#) (const categoric::VariablePtr &var, std::size_t threads=1)
- std::vector< float > [getMarginalDistribution](#) (const std::string &var, std::size_t threads=1)
same as getMarginalDistribution(const categoric::VariablePtr &, std::size_t), but passing the name of the variable, which is internally searched.
- factor::Factor [getJointMarginalDistribution](#) (const categoric::Group &subgroup, std::size_t threads=1)
- factor::Factor [getJointMarginalDistribution](#) (const std::vector< std::string > &subgroup, std::size_t threads=1)
same as getJointMarginalDistribution(const categoric::VariablesSet &, std::size_t), but passing the names of the variables, which are internally searched.
- std::size_t [getMAP](#) (const categoric::VariablePtr &var, std::size_t threads=1)
- std::size_t [getMAP](#) (const std::string &var, std::size_t threads=1)
same as getMAP(const categoric::VariablePtr &, std::size_t), but passing the name of the variable, which is internally searched.
- std::vector< size_t > [getHiddenSetMAP](#) (std::size_t threads=1)

Additional Inherited Members

9.58.1 Member Function Documentation

9.58.1.1 getHiddenSetMAP()

```
std::vector<size_t> EFG::strct::QueryManager::getHiddenSetMAP (
    std::size_t threads = 1 )
```

Returns

the Maximum a Posteriori estimation of the hidden variables, conditioned to the last set of evidences. Values are ordered with the same order used by the set of variables returned in [getHiddenVariables\(\)](#)

Parameters

<i>the</i>	number of threads to use for propagating the belief before returning the result.
------------	--

9.58.1.2 getJointMarginalDistribution() [1/2]

```
factor::Factor EFG::strct::QueryManager::getJointMarginalDistribution (
    const categoric::Group & subgroup,
    std::size_t threads = 1 )
```

Returns

a factor representing the joint distribution of the subgraph described by the passed set of variables.

Parameters

<i>the</i>	involved variables
<i>the</i>	number of threads to use for propagating the belief before returning the result.

Exceptions

<i>when</i>	some of the passed variable names are not found
-------------	---

9.58.1.3 getJointMarginalDistribution() [2/2]

```
factor::Factor EFG::strct::QueryManager::getJointMarginalDistribution (
```

```
const std::vector< std::string > & subgroup,
std::size_t threads = 1 )
```

same as `getJointMarginalDistribution(const categoric::VariablesSet &, std::size_t)`, but passing the names of the variables, which are internally searched.

Exceptions

<i>in</i>	case the passed set of variables is not representative of a valid group
-----------	---

9.58.1.4 getMAP()

```
std::size_t EFG::strct::QueryManager::getMAP (
    const categoric::VariablePtr & var,
    std::size_t threads = 1 )
```

Returns

the Maximum a Posteriori estimation of a specific variable in the model, conditioned to the last set of evidences.

Parameters

<i>the</i>	involved variable
<i>the</i>	number of threads to use for propagating the belief before returning the result.

Exceptions

<i>when</i>	the passed variable name is not found
-------------	---------------------------------------

9.58.1.5 getMarginalDistribution()

```
std::vector<float> EFG::strct::QueryManager::getMarginalDistribution (
    const categoric::VariablePtr & var,
    std::size_t threads = 1 ) [inline]
```

Returns

the marginal probability of the passed variable, i.e. $P(\text{var}|\text{observations})$, conditioned to the last set of evidences.

Parameters

<i>the</i>	involved variable
<i>the</i>	number of threads to use for propagating the belief before returning the result.

Parameters

<i>the</i>	model whose factors should be inserted/copied
<i>when</i>	passing true the factors are deep copied, while in the contrary case shallow copies of the smart pointers storing the factors are inserted into this model.

The documentation for this class was generated from the following file:

- /home/andrea/Scrivania/GitProj/EFG-refactoring/src/header/EasyFactorGraph/model/RandomField.h

9.60 EFG::strct::GibbsSampler::SamplerNode Struct Reference

Classes

- struct [DynamicDependency](#)

Public Member Functions

- bool **noChangingDeps** (const std::unordered_set< const std::size_t * > &will_change) const

Public Attributes

- std::size_t * **value_in_combination**
- const [factor::UnaryFactor](#) * **static_dependencies**
- std::vector< [DynamicDependency](#) > **dynamic_dependencies**

The documentation for this struct was generated from the following file:

- /home/andrea/Scrivania/GitProj/EFG-refactoring/src/header/EasyFactorGraph/structure/GibbsSampler.h

9.61 EFG::strct::GibbsSampler::SamplesGenerationContext Struct Reference

Public Attributes

- std::size_t **samples_number**
- std::optional< std::size_t > [delta_iterations](#)
number of iterations used to evolve the model between the drawing of one sample and another
- std::optional< std::size_t > [seed](#)
sets the seed of the random engine. Passing a nullopt will make the sampler to generate a random seed by using the current time.
- std::optional< std::size_t > [transient](#)
number of samples to discard before actually starting the sampling procedure.

9.61.1 Member Data Documentation

9.61.1.1 transient

`std::optional<std::size_t> EFG::strct::GibbsSampler::SamplesGenerationContext::transient`

number of samples to discard before actually starting the sampling procedure.

When nothing is specified, 10 times `delta_iterations` is assumed.

The documentation for this struct was generated from the following file:

- `/home/andrea/Scrivania/GitProj/EFG-refactoring/src/header/EasyFactorGraph/structure/GibbsSampler.h`

9.62 EFG::strct::PoolAware::ScopedPoolActivator Class Reference

Public Member Functions

- **ScopedPoolActivator** ([PoolAware](#) &subject, `std::size_t new_size`)

The documentation for this class was generated from the following file:

- `/home/andrea/Scrivania/GitProj/EFG-refactoring/src/header/EasyFactorGraph/structure/bases/PoolAware.h`

9.63 EFG::factor::Factor::SimplyAntiCorrelatedTag Struct Reference

The documentation for this struct was generated from the following file:

- `/home/andrea/Scrivania/GitProj/EFG-refactoring/src/header/EasyFactorGraph/factor/Factor.h`

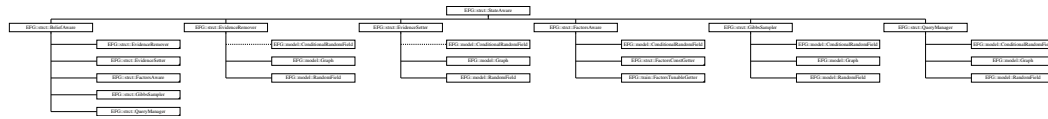
9.64 EFG::factor::Factor::SimplyCorrelatedTag Struct Reference

The documentation for this struct was generated from the following file:

- `/home/andrea/Scrivania/GitProj/EFG-refactoring/src/header/EasyFactorGraph/factor/Factor.h`

9.65 EFG::strct::StateAware Class Reference

Inheritance diagram for EFG::strct::StateAware:



Classes

- struct [GraphState](#)
- struct [NodeLocation](#)

Public Member Functions

- const auto & [getAllVariables](#) () const
- categoric::VariablesSet [getHiddenVariables](#) () const
- categoric::VariablesSet [getObservedVariables](#) () const
- const auto & [getEvidences](#) () const
- categoric::VariablePtr [findVariable](#) (const std::string &name) const
- **StateAware** (const [StateAware](#) &)=delete
- [StateAware](#) & **operator=** (const [StateAware](#) &)=delete
- **StateAware** ([StateAware](#) &&)=delete
- [StateAware](#) & **operator=** ([StateAware](#) &&)=delete

Protected Member Functions

- const [GraphState](#) & **state** () const
- [GraphState](#) & **stateMutable** ()
- std::optional< [NodeLocation](#) > **locate** (const categoric::VariablePtr &var) const
- std::optional< [NodeLocation](#) > **locate** (const std::string &var_name) const

9.65.1 Member Function Documentation

9.65.1.1 findVariable()

```
categoric::VariablePtr EFG::strct::StateAware::findVariable (
    const std::string & name ) const
```

Returns

the variable in the model with the passed name

Exceptions

<i>in</i>	case no variable with the specified name exists in this model
-----------	---

9.65.1.2 getAllVariables()

```
const auto& EFG::strct::StateAware::getAllVariables ( ) const [inline]
```

Returns

all the variables that are part of the model.

9.65.1.3 getEvidences()

```
const auto& EFG::strct::StateAware::getEvidences ( ) const [inline]
```

Returns

all the variables defining the evidence set, together with the associated values

9.65.1.4 getHiddenVariables()

```
categoric::VariablesSet EFG::strct::StateAware::getHiddenVariables ( ) const
```

Returns

all the variables defining the hidden set of variables

9.65.1.5 getObservedVariables()

```
categoric::VariablesSet EFG::strct::StateAware::getObservedVariables ( ) const
```

Returns

all the variables defining the evidence set

The documentation for this class was generated from the following file:

- /home/andrea/Scrivania/GitProj/EFG-refactoring/src/header/EasyFactorGraph/structure/bases/StateAware.h

9.66 EFG::strct::HiddenCluster::TopologyInfo Struct Reference

Public Member Functions

- bool **canUpdateMessage** () const
- std::optional< float > **updateMessage** (PropagationKind kind)

Public Attributes

- [Node](#) * **sender**
- [Node::Connection](#) * **connection**
- std::vector< const [Node::Connection](#) * > **dependencies**

The documentation for this struct was generated from the following file:

- /home/andrea/Scrivania/GitProj/EFG-refactoring/src/header/EasyFactorGraph/structure/Types.h

9.67 EFG::train::TrainInfo Struct Reference

Public Attributes

- std::size_t **threads** = 1
Number of threads to use for the training procedure.
- float **stochastic_percentage** = 1.f
1 means actually to use all the train set, adopting a classical gradient based approach. A lower value implies to a stochastic gradient based approach.

The documentation for this struct was generated from the following file:

- /home/andrea/Scrivania/GitProj/EFG-refactoring/src/header/EasyFactorGraph/trainable/ModelTrainer.h

9.68 EFG::train::TrainSet Class Reference

Classes

- class [Iterator](#)
an object able to iterate all the combinations that are part of a training set or a sub portion of it.

Public Member Functions

- [TrainSet](#) (const std::vector< std::vector< std::size_t >> &combinations)
- const auto & **getCombinations** () const
- [Iterator](#) **makeIterator** () const
- [Iterator](#) **makeSubSetIterator** (float percentage) const

9.68.1 Constructor & Destructor Documentation

9.68.1.1 TrainSet()

```
EFG::train::TrainSet::TrainSet (
    const std::vector< std::vector< std::size_t >> & combinations )
```

Parameters

<i>the</i>	set of combinations that will be part of the train set.
------------	---

Exceptions

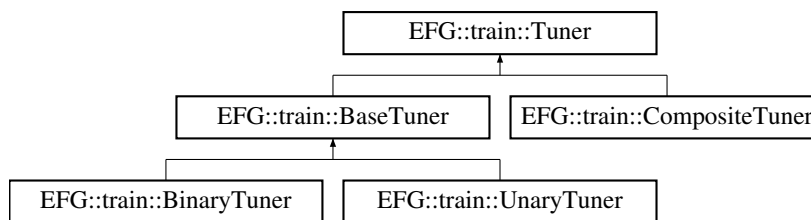
<i>if</i>	the combinations don't have all the same size
<i>if</i>	the combinations container is empty

The documentation for this class was generated from the following file:

- /home/andrea/Scrivania/GitProj/EFG-refactoring/src/header/EasyFactorGraph/trainable/TrainSet.h

9.69 EFG::train::Tuner Class Reference

Inheritance diagram for EFG::train::Tuner:



Public Member Functions

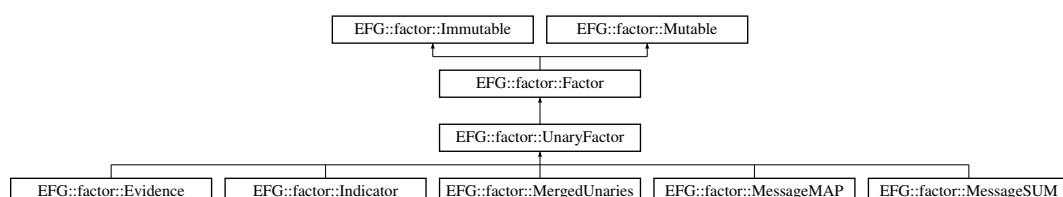
- virtual float **getGradientAlpha** (const [TrainSet::Iterator](#) &iter)=0
- virtual float **getGradientBeta** ()=0
- virtual void **setWeight** (float w)=0
- virtual float **getWeight** () const =0

The documentation for this class was generated from the following file:

- /home/andrea/Scrivania/GitProj/EFG-refactoring/src/header/EasyFactorGraph/trainable/tuners/Tuner.h

9.70 EFG::factor::UnaryFactor Class Reference

Inheritance diagram for EFG::factor::UnaryFactor:



Public Member Functions

- const categoric::VariablePtr & **getVariable** () const
- float **diff** (const [UnaryFactor](#) &o) const

Protected Member Functions

- **UnaryFactor** (FunctionPtr data)

Protected Attributes

- categoric::VariablePtr **variable**

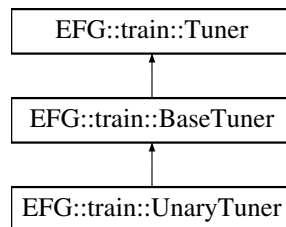
Additional Inherited Members

The documentation for this class was generated from the following file:

- /home/andrea/Scrivania/GitProj/EFG-refactoring/src/header/EasyFactorGraph/structure/SpecialFactors.h

9.71 EFG::train::UnaryTuner Class Reference

Inheritance diagram for EFG::train::UnaryTuner:



Public Member Functions

- **UnaryTuner** ([strct::Node](#) &node, const std::shared_ptr< [factor::FactorExponential](#) > &factor, const categoric::VariablesSoup &variables_in_model)
- float **getGradientBeta** () final

Protected Attributes

- [strct::Node](#) & **node**

Additional Inherited Members

The documentation for this class was generated from the following file:

- /home/andrea/Scrivania/GitProj/EFG-refactoring/src/header/EasyFactorGraph/trainable/tuners/Unary↔
Tuner.h

9.72 EFG::strct::UniformSampler Class Reference

Public Member Functions

- `std::size_t sampleFromDiscrete (const std::vector< float > &distribution) const`
- `void resetSeed (std::size_t newSeed)`

The documentation for this class was generated from the following file:

- `/home/andrea/Scrivania/GitProj/EFG-refactoring/src/header/EasyFactorGraph/structure/GibbsSampler.h`

9.73 EFG::categoric::Variable Class Reference

An object representing an immutable categoric variable.

```
#include <Variable.h>
```

Public Member Functions

- `Variable (std::size_t size, const std::string &name)`
- `std::size_t size () const`
- `const std::string & name () const`
- `bool operator== (const Variable &o) const`

Protected Attributes

- `const size_t size_`
- `const std::string name_`

9.73.1 Detailed Description

An object representing an immutable categoric variable.

9.73.2 Constructor & Destructor Documentation

9.73.2.1 `Variable()`

```
EFG::categoric::Variable::Variable (
    std::size_t size,
    const std::string & name )
```

Parameters

<i>domain</i>	size of this variable
<i>name</i>	used to label this variable.

Exceptions

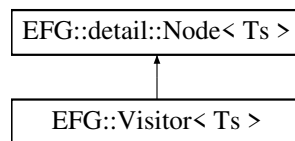
<i>passing</i>	0 as size
<i>passing</i>	an empty string as name

The documentation for this class was generated from the following file:

- /home/andrea/Scrivania/GitProj/EFG-refactoring/src/header/EasyFactorGraph/categoric/Variable.h

9.74 EFG::Visitor< Ts > Struct Template Reference

Inheritance diagram for EFG::Visitor< Ts >:



Public Member Functions

- **Visitor** (detail::Predicate< Ts > &&...predicates)
- template<typename T >
void **operator()** (T &subject) const
- void **visit** (std::variant< Ts... > &subject) const
- template<typename T , typename... Tothers>
void **set** (detail::Predicate< T > &&first, detail::Predicate< Tothers > &&...preds)
- template<typename T >
void **set** (detail::Predicate< T > &&last)

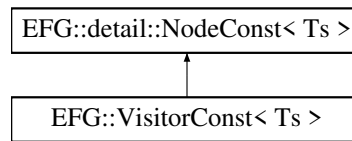
Additional Inherited Members

The documentation for this struct was generated from the following file:

- /home/andrea/Scrivania/GitProj/EFG-refactoring/src/header/EasyFactorGraph/misc/Visitor.h

9.75 EFG::VisitorConst< Ts > Struct Template Reference

Inheritance diagram for EFG::VisitorConst< Ts >:



Public Member Functions

- **VisitorConst** (detail::PredicateConst< Ts > &&...predicates)
- template<typename T >
void **operator()** (const T &subject) const
- void **visit** (const std::variant< Ts... > &subject) const
- template<typename T , typename... Tothers>
void **set** (detail::PredicateConst< T > &&first, detail::PredicateConst< Tothers > &&...preds)
- template<typename T >
void **set** (detail::PredicateConst< T > &&last)

Additional Inherited Members

The documentation for this struct was generated from the following file:

- /home/andrea/Scrivania/GitProj/EFG-refactoring/src/header/EasyFactorGraph/misc/Visitor.h

Index

- absorb
 - EFG::model::Graph, [92](#)
 - EFG::model::RandomField, [114](#)
- add
 - EFG::categorical::Group, [95](#)
- addTunableFactor
 - EFG::train::FactorsTunableInserter, [89](#)
- cloneWithPermutedGroup
 - EFG::factor::Factor, [81](#)
- ConditionalRandomField
 - EFG::model::ConditionalRandomField, [71](#)
- copyTunableFactor
 - EFG::train::FactorsTunableInserter, [89](#)
- EFG, [55](#)
 - SmartMap, [56](#)
 - SmartSet, [56](#)
- EFG::Cache< T >, [67](#)
- EFG::categorical, [57](#)
 - get_complementary, [57](#)
- EFG::categorical::Group, [93](#)
 - add, [95](#)
 - getVariables, [95](#)
 - getVariablesSet, [95](#)
 - Group, [94](#)
 - replaceVariables, [96](#)
 - size, [96](#)
- EFG::categorical::GroupRange, [96](#)
 - GroupRange, [97](#)
 - operator++, [98](#)
- EFG::categorical::Variable, [122](#)
 - Variable, [122](#)
- EFG::Comparator< T >, [69](#)
- EFG::detail::Merger< Separator >, [106](#)
- EFG::detail::Node< T >, [108](#)
- EFG::detail::NodeConst< T >, [109](#)
- EFG::Error, [73](#)
- EFG::factor, [58](#)
- EFG::factor::Evidence, [73](#)
- EFG::factor::Factor, [79](#)
 - cloneWithPermutedGroup, [81](#)
 - Factor, [80](#)
- EFG::factor::Factor::CloneTrasformedImagesTag, [68](#)
- EFG::factor::Factor::SimplyAntiCorrelatedTag, [116](#)
- EFG::factor::Factor::SimplyCorrelatedTag, [116](#)
- EFG::factor::FactorExponential, [82](#)
 - FactorExponential, [83](#)
 - getWeight, [83](#)
- EFG::factor::Function, [89](#)
 - transform, [90](#)
- EFG::factor::Function::CombinationHasher, [68](#)
- EFG::factor::Function::Info, [103](#)
- EFG::factor::ImageFinder, [99](#)
 - findTransformed, [99](#)
- EFG::factor::Immutable, [100](#)
 - getProbabilities, [100](#)
- EFG::factor::Indicator, [103](#)
- EFG::factor::MergedUnaries, [105](#)
- EFG::factor::MessageMAP, [106](#)
- EFG::factor::MessageSUM, [107](#)
- EFG::factor::Mutable, [107](#)
 - set, [108](#)
- EFG::factor::UnaryFactor, [120](#)
- EFG::Hasher< T >, [98](#)
- EFG::io, [59](#)
 - import_train_set, [59](#)
- EFG::io::json, [60](#)
- EFG::io::json::Exporter, [76](#)
 - exportToFile, [77](#)
 - exportToJson, [77](#)
- EFG::io::json::Importer, [101](#)
 - importFromFile, [101](#)
 - importFromJson, [101](#)
- EFG::io::xml, [60](#)
- EFG::io::xml::Exporter, [77](#)
 - exportToFile, [78](#)
 - exportToString, [78](#)
- EFG::io::xml::ExportInfo, [78](#)
- EFG::io::xml::Importer, [102](#)
 - importFromFile, [102](#)
- EFG::model, [60](#)
- EFG::model::ConditionalRandomField, [70](#)
 - ConditionalRandomField, [71](#)
 - makeTrainSet, [71](#)
 - setEvidences, [72](#)
- EFG::model::Graph, [92](#)
 - absorb, [92](#)
- EFG::model::RandomField, [114](#)
 - absorb, [114](#)
- EFG::strct, [61](#)
- EFG::strct::BaselineLoopyPropagator, [65](#)
- EFG::strct::BeliefAware, [66](#)
- EFG::strct::EvidenceRemover, [74](#)
 - removeEvidence, [74](#)
 - removeEvidences, [75](#)
- EFG::strct::EvidenceSetter, [75](#)
 - setEvidence, [76](#)
- EFG::strct::FactorsAware, [84](#)

- getAllFactors, 84
- EFG::strct::FactorsConstGetter, 84
 - getConstFactors, 85
- EFG::strct::FactorsConstInserter, 85
- EFG::strct::GibbsSampler, 91
 - makeSamples, 91
- EFG::strct::GibbsSampler::SamplerNode, 115
- EFG::strct::GibbsSampler::SamplerNode::DynamicDependencies, 73
- EFG::strct::GibbsSampler::SamplesGenerationContext, 115
 - transient, 116
- EFG::strct::HiddenCluster, 98
- EFG::strct::HiddenCluster::TopologyInfo, 119
- EFG::strct::LoopyBeliefPropagationStrategy, 105
- EFG::strct::Node, 108
- EFG::strct::Node::Connection, 72
- EFG::strct::Pool, 110
- EFG::strct::PoolAware, 110
- EFG::strct::PoolAware::ScopedPoolActivator, 116
- EFG::strct::PropagationContext, 110
- EFG::strct::PropagationResult, 111
- EFG::strct::PropagationResult::ClusterInfo, 68
- EFG::strct::QueryManager, 111
 - getHiddenSetMAP, 112
 - getJointMarginalDistribution, 112
 - getMAP, 113
 - getMarginalDistribution, 113
- EFG::strct::StateAware, 117
 - findVariable, 117
 - getAllVariables, 118
 - getEvidences, 118
 - getHiddenVariables, 118
 - getObservedVariables, 118
- EFG::strct::StateAware::GraphState, 93
- EFG::strct::StateAware::NodeLocation, 109
- EFG::strct::UniformSampler, 122
- EFG::train, 62
 - set_ones, 62
 - train_model, 63
- EFG::train::BaseTuner, 65
- EFG::train::BinaryTuner, 67
- EFG::train::CompositeTuner, 69
- EFG::train::FactorsTunableGetter, 86
 - getWeights, 87
 - getWeightsGradient, 87
 - setWeights, 87
- EFG::train::FactorsTunableInserter, 88
 - addTunableFactor, 89
 - copyTunableFactor, 89
- EFG::train::TrainInfo, 119
- EFG::train::TrainSet, 119
 - TrainSet, 119
- EFG::train::TrainSet::Iterator, 103
 - Iterator, 104
 - size, 104
- EFG::train::Tuner, 120
- EFG::train::UnaryTuner, 121
- EFG::Visitor< Ts >, 123
- EFG::VisitorConst< Ts >, 124
- exportToFile
 - EFG::io::json::Exporter, 77
 - EFG::io::xml::Exporter, 78
- exportToJson
 - EFG::io::json::Exporter, 77
- exportToString
 - EFG::io::xml::Exporter, 78
- Factor
 - EFG::factor::Factor, 80
- FactorExponential
 - EFG::factor::FactorExponential, 83
- findTransformed
 - EFG::factor::ImageFinder, 99
- findVariable
 - EFG::strct::StateAware, 117
- get_complementary
 - EFG::categoric, 57
- getAllFactors
 - EFG::strct::FactorsAware, 84
- getAllVariables
 - EFG::strct::StateAware, 118
- getConstFactors
 - EFG::strct::FactorsConstGetter, 85
- getEvidences
 - EFG::strct::StateAware, 118
- getHiddenSetMAP
 - EFG::strct::QueryManager, 112
- getHiddenVariables
 - EFG::strct::StateAware, 118
- getJointMarginalDistribution
 - EFG::strct::QueryManager, 112
- getMAP
 - EFG::strct::QueryManager, 113
- getMarginalDistribution
 - EFG::strct::QueryManager, 113
- getObservedVariables
 - EFG::strct::StateAware, 118
- getProbabilities
 - EFG::factor::Immutable, 100
- getVariables
 - EFG::categoric::Group, 95
- getVariablesSet
 - EFG::categoric::Group, 95
- getWeight
 - EFG::factor::FactorExponential, 83
- getWeights
 - EFG::train::FactorsTunableGetter, 87
- getWeightsGradient
 - EFG::train::FactorsTunableGetter, 87
- Group
 - EFG::categoric::Group, 94
- GroupRange
 - EFG::categoric::GroupRange, 97
- import_train_set

- EFG::io, [59](#)
- importFromFile
 - EFG::io::json::Importer, [101](#)
 - EFG::io::xml::Importer, [102](#)
- importFromJson
 - EFG::io::json::Importer, [101](#)
- Iterator
 - EFG::train::TrainSet::Iterator, [104](#)
- makeSamples
 - EFG::strct::GibbsSampler, [91](#)
- makeTrainSet
 - EFG::model::ConditionalRandomField, [71](#)
- operator++
 - EFG::categoric::GroupRange, [98](#)
- removeEvidence
 - EFG::strct::EvidenceRemover, [74](#)
- removeEvidences
 - EFG::strct::EvidenceRemover, [75](#)
- replaceVariables
 - EFG::categoric::Group, [96](#)
- set
 - EFG::factor::Mutable, [108](#)
- set_ones
 - EFG::train, [62](#)
- setEvidence
 - EFG::strct::EvidenceSetter, [76](#)
- setEvidences
 - EFG::model::ConditionalRandomField, [72](#)
- setWeights
 - EFG::train::FactorsTunableGetter, [87](#)
- size
 - EFG::categoric::Group, [96](#)
 - EFG::train::TrainSet::Iterator, [104](#)
- SmartMap
 - EFG, [56](#)
- SmartSet
 - EFG, [56](#)
- std::hash< EFG::categoric::Variable >, [98](#)
- train_model
 - EFG::train, [63](#)
- TrainSet
 - EFG::train::TrainSet, [119](#)
- transform
 - EFG::factor::Function, [90](#)
- transient
 - EFG::strct::GibbsSampler::SamplesGenerationContext, [116](#)
- Variable
 - EFG::categoric::Variable, [122](#)