Algorithms for Graphical Models

James Cussens

Preface

This book exists to support the teaching of the *Algorithms for Graphical Models (AGM)* module taught to 3rd and 4th-year undergraduates at the Department of Computer Science, University of York. Graphical models have become important to a number of disciplines, most obviously statistics, but also to computer science—particularly Artificial Intelligence (AI). There already exist numerous introductions to graphical modelling, so it behoves me to explain why this book has been written.

Firstly, this book is aimed at *teaching* graphical models to *computer science* student, who have no previous experience of this area. Since it supports introductory teaching no new material will be found here and the pace is reasonably gentle. Since computer science students are its audience the emphasis is on the algorithmic aspects of graphical models at the expense of statistical depth. This informs the decision to focus exclusively on *finite, discrete* graphical models. (A more accurate title would be *Algorithms for Discrete Graphical Models*, but we like TLAs—three letter acronyms—at York.) No great knowledge of statistical theory is needed to understand finite, discrete probability distributions.

Having ducked out of statistical theory we plunge into the algorithmic aspects of graphical models in some depth. The standard approach would be to describe the various algorithms in pseudo-code and leave it to the reader to actually implement them in the language of his/her choice. Here all algorithms are actually-executable Python programs. One big advantage of using real code is that it can be run and tested for correctness. Python has the following advantages: (1) it is stable, freely available and runs on a variety of platforms, (2) it is high-level with an uncluttered syntax leading devotees to call it 'executable pseudo-code', (3) it has good support for object-oriented programming and (4) it can be run from an interpreter. Surprisingly, I have found this last feature the most useful from a teaching perspective, since it allows demonstration of algorithms 'on the fly'. The Python code supporting this book is a Python package called `gPy` which is available for download at `http://www-users.cs.york.ac.uk/~jc/teaching/agm/gPy/`. The disadvantage of this approach is that to understand this book you need to understand Python—it is, however, very easy to learn.

A third distinguishing feature is that Bayesian networks (BNs) are not the main focus. Instead emphasis is on general *factored representations* of probability distributions, Bayesian networks being introduced as a special case. The

rationale here is that many of the algorithms applicable to BNs (e.g. variable elimination) apply to the more general case. Introducing such algorithms in the general case makes it easier to understand their essential features. A consequence of this approach is that hypergraphs play a more central role than in many other presentations. This approach has been heavily influenced by Lauritzen's *Graphical Models* [9].

Contents

List of Algorithms

List of gPy Examples

Chapter 1

Introduction

1.1 What are graphical models?

1.1.1 The 'Asia' Bayesian network

The 'Asia' Bayesian network will be used extensively in the Chapter 7 which concerns Bayesian networks. This example was introduced in [10] and is available in `gPy` as the object `gPy.Examples.asia`. The idea of the example is to provide a probabilistic expert system to help with diagnosis at some (entirely fictitious) chest clinic. The eight variables involved—`VisitAsia, Tuberculosis, XRay, Dyspnea, Bronchitis, Smoking, TbOrCa` and `Cancer`—represents attributes of patients. Typically, some values will be known (e.g. `Smoking`) and some unknown, at least initially, (e.g. `Cancer`).

`gPy` **Example 1 (Print out the 'Asia' Bayesian network)**
To print out the 'Asia' Bayesian network as seen in Fig 1.1 do:

```
>>> from gPy.Examples import asia
>>> print asia
```

1.2 The pitfalls of probabilistic reasoning

1.3 Conditional independence and making sense of the world

Smoking	Bronchitis	
	absent	present
nonsmoker	0.70	0.30
smoker	0.40	0.60

Smoking	Cancer	
	absent	present
nonsmoker	0.99	0.01
smoker	0.90	0.10

Bronchitis	TbOrCa	Dyspnea	
		absent	present
absent	false	0.90	0.10
absent	true	0.30	0.70
present	false	0.20	0.80
present	true	0.10	0.90

Smoking	
nonsmoker	smoker
0.50	0.50

Cancer	Tuberculosis	TbOrCa	
		false	true
absent	absent	1.00	0.00
absent	present	0.00	1.00
present	absent	0.00	1.00
present	present	0.00	1.00

VisitAsia	Tuberculosis	
	absent	present
no_visit	0.99	0.01
visit	0.95	0.05

VisitAsia	
no_visit	visit
0.99	0.01

TbOrCa	XRay	
	abnormal	normal
false	0.05	0.95
true	0.98	0.02

Figure 1.1: The 'Asia' Bayesian network

Chapter 2

Installing and using the software

2.1 gPy for AGM students

If you are a student taking AGM and are using a departmental machine running Linux then the `gPy` package has already been installed for you. To ensure that your Python programs and Python interpreter sessions can find the software it suffices to ensure that `/usr/course/agm/gPy` is one of the directories on Python's search path for modules. This is done by setting the environmental variable `PYTHONPATH` appropriately. For example, if you are running `tcsh` then adding the following line to your `.tcshrc` file would do:

```
setenv PYTHONPATH /usr/course/agm/gPy
```

Of course, you may want more than one directory on your module search path. For example, I have

```
setenv PYTHONPATH ~jc/lib/python:/usr/course/agm/gPy/
```

2.2 Installing the software

todo

2.3 Using the Python interpreter

The material in the rest of this book depends on you running Python commands from the Python interpreter which uses code from the `gPy` package. Assuming you have installed `gPy` somewhere where your Python process can find it, this is very easy. Using Linux, just type

```
python
```

at the prompt, and something similar to the following will appear:

```
Python 2.4.3 (#1, Jul 26 2006, 20:13:39)
[GCC 3.4.6] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

The `>>>` is the Python prompt. Under Windows, start Python from the Programs menu, and you will get the same prompt. You are now ready to give commands to the Python interpreter by simply typing them in and hitting enter.

Chapter 3

Contingency tables and joint probability distributions

We begin not with probability but with something much simpler: data. Consider the (single-table) database of fictitious health data presented in Fig 3.1. For brevity, only 5 rows are shown of what should be imagined as a database of 100 rows. There are five fields: `PatientId` which would be the primary key, the person's name and three others which record health information concerning that individual.

3.1 Contingency tables

A *contingency table* is a table of *counts* which can be produced from a database like that of Fig 3.1 by ignoring certain fields and simply counting up how often each possible combination of the remaining field values occur in the data: see Fig 3.2.

PatientId	Name	Bronchitis	Cancer	Smoking
pno001	j_smith	absent	absent	nonsmoker
pno002	r_higgins	absent	absent	nonsmoker
pno003	a_jones	present	present	smoker
pno004	i_brown	absent	present	nonsmoker
....	...	...	...	....
pno100	k_evans	present	present	smoker

Figure 3.1: Single table database of fictitious health data

Bronchitis	Cancer	Smoking	
absent	absent	nonsmoker	3
absent	absent	smoker	0
absent	present	nonsmoker	27
absent	present	smoker	15
present	absent	nonsmoker	2
present	absent	smoker	0
present	present	nonsmoker	18
present	present	smoker	35

Figure 3.2: Contingency table for fictitious health data

gPy **Example 2 (Display a contingency table)**
To grab and print the contingency table in Fig 3.2:

```
>>> from gPy.Examples import contab
>>> print contab
```

If you would prefer to have this table in a window, first make a widget in which to display it:

```
>>> from Tkinter import *
>>> win = Tk()
```

and do this:

```
>>> contab.gui_display(win)
```

So far we have been using the database term 'field' to refer to `Bronchitis`, `Cancer` and `Smoking`, but when the data forms a contingency table the term *variable* is more normal. `Bronchitis` is thus a variable with two possible *values*: `absent` and `present`. The word 'variable' makes sense since the value a variable takes varies between different patients. We will use the common statistical convention that variables begin with upper-case letters and values with lower-case ones. The variables `Cancer` and `Smoking` also have two values. (Note that in the statistical literature the values of discrete variables are often called *levels*.) Any particular patient will have a value for each variable, e.g. `(Bronchitis=absent, Cancer=present, Smoking=nonsmoker)`. Now for some terminology. A specification of a particular value for a variable is called an *instantiation*. A specification of values for several variables simultaneously is a *joint instantiation*. A specification of values for all variables is called a *full joint instantiation*. `(Bronchitis=absent, Cancer=present, Smoking=nonsmoker)` is thus an ex-

```
Bronchitis | Cancer  |
---------- | ------- | ----
absent     | absent  |  (  3 +  0 ) =  3
absent     | present |  ( 27 + 15 ) = 42
present    | absent  |  (  2 +  0 ) =  2
present    | present |  ( 18 + 35 ) = 53
```

Figure 3.3: Marginal contingency table produced from the table given in Fig 3.2. The expressions in parentheses are for explanation only and would not normally appear; only the final counts for each row would.

ample of a full joint instantiation.

3.1.1 Marginalisation of contingency tables

The contingency table in Fig 3.2 provides a count for all possible joint instantiations of the three variables. Often a more coarse-grained 'picture' of the data is wanted. Suppose we were unconcerned about `Smoking` and just wanted counts for joint instantiations of `Bronchitis` and `Cancer`. To get the count for, say, `(Bronchitis=absent, Cancer=present)`, it suffices to add the count for `(Bronchitis=absent, Cancer=present, Smoking=nonsmoker)`, which is 27, and the count for `(Bronchitis=absent, Cancer=present, Smoking=smoker)`, which is 15, giving a count of 42. Doing this for all four possible joint instantiations of `Bronchitis` and `Cancer` produces the *marginal contingency table* in Fig 3.3. (The word 'marginal' is used since one can imagine writing the results of the additions beside the original contingency table, i.e. 'in the margin'.)

gPy Example 3 (Compute a marginal contingency table)
Assuming you have ***contab*** *in your environment, the following will print out the marginal table:*

```
>>> print contab.sumout(['Smoking'])

Bronchitis | Cancer  |
---------- | ------- | ----
absent     | absent  |      3
absent     | present |     42
present    | absent  |      2
present    | present |     53
```

Assuming you have the window ***win*** *available (from* ***gPy****Example 2), you can display it in that window with:*

```
>>> contab.sumout(['Smoking']).gui_display(win)
```

Y	X		Y	
	no	yes		
no	30	40	no	70
yes	10	20	yes	30

X	
no	yes
40	60

Figure 3.4: Marginalising a 2-dimensional contingency table by projecting down onto its `X` and `Y` dimensions.

Note that what makes the contingency table in Fig 3.3 a *marginal* contingency table is simply the existence of another contingency table (that in Fig 3.2) from which it can be created by *marginalisation* which is the name for the process of producing a marginal table. If a variable has been removed by marginalisation, we say that it has been *marginalised away* or *summed out*. It is useful to take a geometric view of a contingency table, seeing each variable in the table as a 'dimension', so that a contingency table with n variables is an *n-dimensional contingency table*. Marginalisation then becomes a form of 'projecting down' onto a (discrete) space of smaller dimension. When there are only two variables the geometry of contingency tables is easy to visualise. Fig 3.4 displays a contingency table for two variables `X` and `Y` laid out in a way that emphasises its two-dimensional nature. Marginal tables produced by projecting down onto the `X` and `Y` dimension are also shown. In the original 2-dimensional table of Fig 3.4 it is easy to visualise each full joint instantiation as occupying a rectangle in 2-dimensional space. Fig 3.5 show two ways of doing this: in the LHS picture the area of 100 units is divided first by a horizontal line producing a leftside rectangle corresponding to `X=no` (area 40) and a rightside rectangle corresponding to `X=yes` (area 60). The `X=no` rectangle is then divided into a `X=no,Y=no` (upper left rectangle) and a `X=no,Y=yes` (lower left rectangle) to produce rectangles with areas corresponding to the counts for each of these full joint instantiations. The `X=yes` rectangle is similarly divided. The RHS picture shows what happens when the initial division is between the different `Y` values. When there are more than 2 variables it is harder to visualise contingency tables.

Marginal contingency tables such as that of Fig 3.3 can, of course, be marginalised further. Summing out `Cancer` generates the one variable ('one-dimensional') table in Fig 3.6. Clearly, due to the commutativity of addition it does not matter in which order variables are summed out.

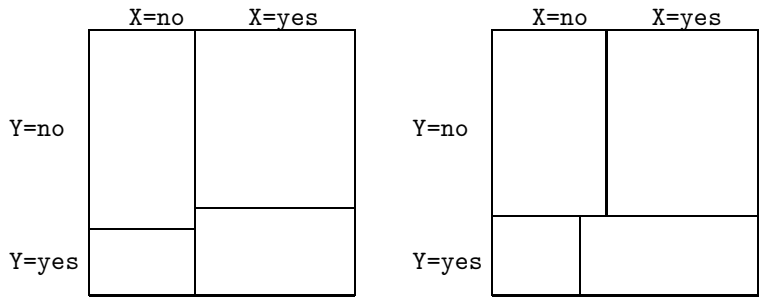

Figure 3.5: Two geometrical views of the contingency table of Fig 3.4.

```
Bronchitis |
---------- | ----
absent     |     45
present    |     55
```

Figure 3.6: Marginal contingency table which can be produced by either summing out `Smoking` and `Cancer` from the table in Fig 3.2 or summing out just `Cancer` from the table in Fig 3.3.

gPy Example 4 (Compute a marginal contingency table)
Assuming you have `contab` *in your environment, the table in Fig 3.6 can be produced in one-step:*

```
>>> print contab.sumout(['Smoking','Cancer'])
```

or in two steps

```
>>> margtab = contab.sumout(['Smoking'])
>>> print margtab.sumout(['Cancer'])
```

3.1.2 Slicing contingency tables

Marginalisation focusses attention on a subset of the variables. *Slicing* focusses attention on particular *values* of specified variables. For example, if we are only concerned about smokers, then we can form a *slice contingency table* by simply deleting all rows where `Smoking=nonsmoker`, producing the table in Fig 3.7. Choosing a single value for a variable like this, is often called *instantiation*: `Smoking` has been *instantiated* to `smoker`.

Bronchitis	Cancer	Smoking	
absent	absent	smoker	0
absent	present	smoker	15
present	absent	smoker	0
present	present	smoker	35

Figure 3.7: Slice contingency table

gPy **Example 5 (Compute a slice of a contingency table)**
To get the table in Fig 3.7 do:

```
>>> fr = FR([contab.copy(True)])
>>> print fr.condition({'Smoking':['smoker']})
```

This involves a number of issues which have yet to be covered. The basic story is that `gPy` *deliberately makes it difficult to slice a table, so instead an* `FR` *object—of which (much) more later—is created and that is sliced. Note also that the method used is called* `condition`*. This is because slicing and conditioning (which will be addressed shortly) are very closely related.*

Note that, in contrast to marginalisation, the sum of the numbers in the table will generally be reduced by slicing (since we have deleted rows), but that no variables have been removed (since we are not deleting columns). This is a non-standard definition of slicing: in most presentations (for example that of [9]) instantiated variables are removed from the sliced contingency table. The rationale for the nonstandard approach taken here is modularity of operations: slicing is defined to only remove rows; to remove columns marginalisation can be used. In the case of instantiated variables summing out is very easy: it suffices to remove the instantiated variable's column. But be clear: in this book the operation called 'slicing' does not remove any variables, only summing out (i.e. marginalisation) does that.

3.2 Probability distributions

The contingency table in Fig 3.2 records actual data about 100 individuals. Often what is needed is probabilistic information specifying, for example, the probability that the next person observed will have cancer, or the probability that the next smoker observed will have cancer. This raises the question of how such probabilistic information can be acquired. For the time being a very simple approach will be taken just so we have some probabilities to play with. From the data in Fig 3.2, probabilities for all 8 possible joint instantiations can

Bronchitis	Cancer	Smoking	
absent	absent	nonsmoker	0.03
absent	absent	smoker	0.00
absent	present	nonsmoker	0.27
absent	present	smoker	0.15
present	absent	nonsmoker	0.02
present	absent	smoker	0.00
present	present	nonsmoker	0.18
present	present	smoker	0.35

Figure 3.8: A joint probability distribution

be generated by dividing all counts by the total count: 100. This generates the table in Fig 3.8. This is an example of *maximum likelihood estimation*. Maximum likelihood estimation is examined properly in Section 9.1, for now just note that we view this derivation of probabilities as a form of *estimation*. The idea is that there is a set of 8 true probabilities whose values can only be estimated (not deduced) from the observed data.

gPy **Example 6 (Display a joint probability distribution)**
To get the table in Fig 3.8 do:

```
>>> probs = contab.normalised()
>>> print probs
```

Keep the `probs` *object around for further work.*

Fig 3.8 displays a *joint probability distribution*. A joint probability distribution specifies a probability for each full joint instantiation of a set of variables such that the probabilities add up to one. It is called a *distribution* because a 'probability mass' of one has been distributed between the 8 full joint instantiations.

In the finite, discrete case (to which this book is restricted) this is almost all there is to probability distributions: a set of mutually exclusive and exhaustive *simple events* are specified, each simple event is assigned a probability, and all the probabilities add up to one. (Simple events are also known as *sample points*.) Here, since the probability distribution is a joint probability distribution, each simple event is a full joint instantiation of the variables, and they are indeed mutually exclusive and exhaustive: any possible patient can only correspond to exactly one row of Fig 3.8.

To make sense of events other than simple events further concepts and a little more formality are needed. Firstly, define the *sample space* to be just the

set of all simple events. So if $E_1, E_2, \ldots, E_n$ are the simple events then the corresponding sample space, denoted Ω, is defined as:

$$\Omega = E_1 \cup E_2 \cup \cdots \cup E_n \tag{3.1}$$

Now for (3.1) to make sense the simple events E_i have to be seen as *subsets of* Ω. This is an important point: formally speaking probabilities are *always* defined on *subsets* of some sample space, not on *elements* of the sample space.

The reason for this approach is that it makes it straightforward to deal with events which are not necessarily simple. An *event* (not necessarily simple) is just a subset of Ω. It follows that any event is the union of simple events. For example $A = E_1 \cup E_2$ is an event. The empty set $\emptyset$ is also an event. The probability of any given event can be found by adding up the probabilities of the simple events which constitute it, so that, for example, $P(A) = P(E_1) + P(E_2)$. Note that it is always the case that $P(\Omega) = 1$ and $P(\emptyset) = 0$. (This is a simplified definition of probability adequate for finite discrete distributions. In the general case it is necessary to define something called a σ-algebra of subsets of Ω and only subsets in the σ-algebra have probabilities assigned to them.)

Although the word 'event' has a formal meaning in probability theory, it is related to the normal meaning of that word. One can think, for example, of the (formal) simple event `(Bronchitis=absent, Cancer=present,Smoking=nonsmoker)` as corresponding to the (potentially real) event that the next observed patient has those 3 specified characteristics.

Note that in our running example all events are defined in terms of instantiations of variables. This will remain the case throughout this book: simple events will always be full joint instantiations. Be aware that this is a deliberate restriction and that in general probability theory events need not correspond to instantiations of variables. However, using variables to describe probability distributions is remarkably useful and has led to the notion of a *random variable*. The formal definition of a random variable requires a consideration of 'measurable' functions which takes us further into probability theory than we need. For present purposes if suffices to note that the variables used to describe our simple events are random variables: *random* because probabilities are associated with the values they can take. Random variables will be discussed in the following sections. At this point just note that a distribution such as that displayed in Fig 3.8 is often denoted as $P(\texttt{Bronchitis}, \texttt{Cancer}, \texttt{Smoking})$. This notation serves to make clear that the distribution P is a joint distribution defined in terms of these 3 random variables. It is important to remember—since the notation hardly makes it obvious—that this notation is used to denote an entire distribution of probabilities, not just a single probability.

3.2.1 Marginal distributions

Joint probability distributions can be marginalised in exactly the same way as contingency tables, and for the same reasons: to produce information restricted to a subset of the original variables. So, for example, a *marginal distribution*

```
Bronchitis | Cancer  |
---------- | ------- | ----
absent     | absent  |  ( 0.03 + 0.00 ) = 0.03
absent     | present |  ( 0.27 + 0.15 ) = 0.42
present    | absent  |  ( 0.02 + 0.00 ) = 0.02
present    | present |  ( 0.18 + 0.35 ) = 0.53
```

Figure 3.9: Marginal distribution $P(\texttt{Bronchitis}, \texttt{Cancer})$ produced from the distribution given in Fig 3.8. The expressions in parentheses are for explanation only and would not normally appear; only the final probabilities for each row would.

involving only the variables `Bronchitis` and `Cancer` is produced by adding up the relevant probabilities as in Fig 3.9. Such a distribution is conventionally denoted by $P(\texttt{Bronchitis}, \texttt{Cancer})$.

gPy **Example 7 (Compute a marginal distribution)**
To generate the marginal distribution on `Bronchitis` *and* `Cancer` *do:*

```
>>> print probs.sumout(['Smoking'])

Bronchitis | Cancer  |
---------- | ------- | ----
absent     | absent  |   0.03
absent     | present |   0.42
present    | absent  |   0.02
present    | present |   0.53
```

Adding counts to produce a marginal contingency table is obviously a correct thing to do, but is it reasonable to add probabilities to produce a marginal distribution like this? It is: because the laws of the probability calculus have been obeyed. One of these laws states that if two events A and B are mutually exclusive (i.e. $A \cap B = \emptyset$) then $P(A \cup B) = P(A) + P(B)$. For example, in the first row of Fig 3.9 the probability of the event `(Bronchitis=absent, Cancer=absent)` has been computed by adding the probabilities of the two (mutually exclusive) events `(Bronchitis=absent, Cancer=absent,Smoking=nonsmoker)` and `(Bronchitis=absent, Cancer=absent,Smoking=smoker)`.

One can, of course, produce marginal distributions involving only a single variable such as shown in Fig 3.10. Some of the most important algorithms to be discussed later, aim to compute marginal distributions as efficiently as possible. A first stab at this task can be found in Chapter 4 where a formal definition of 'marginal distribution' can be found.

Bronchitis	
absent	0.45
present	0.55

Figure 3.10: Marginal distribution P(`Bronchitis`) for the variable `Bronchitis`

Bronchitis	Cancer	Smoking	
absent	absent	smoker	0.00
absent	present	smoker	0.15
present	absent	smoker	0.00
present	present	smoker	0.35

Figure 3.11: A slice of a joint probability distribution

gPy **Example 8 (Compute a marginal distribution)**
To get the marginal distribution in Fig 3.10, do:

```
>>> print probs.sumout(['Smoking','Cancer'])
```

3.2.2 Conditional distributions

Probability distributions can be sliced in exactly the same way as contingency tables. For example, Fig 3.11 shows a slice produced by specifying that `Smoking=smoker`.

gPy **Example 9 (Compute a slice of a joint probability distribution)**
To get the slice in Fig 3.11 do:

```
>>> fr = FR([probs.copy(True)])
>>> print fr.condition({'Smoking':['smoker']})
```

Slicing a probability distribution does not produce another probability distribution, since the remaining numbers will not generally add up to one: in Fig 3.11 they add up to 0.5. However it is simple to produce a probability distribution from a slice by simply *renormalising* the numbers so that they sum to one: the sum of all the numbers is found and then each number is divided by that sum. This produces a *conditional probability distribution*. The conditional distribution formed by applying renormalisation to the slice in Fig 3.11 is given

```
Bronchitis | Cancer  | Smoking |
---------- | ------- | ------- | ----
absent     | absent  | smoker  |   0.00
absent     | present | smoker  |   0.30
present    | absent  | smoker  |   0.00
present    | present | smoker  |   0.70
```

Figure 3.12: The conditional probability distribution $P(\texttt{Bronchitis}, \texttt{Cancer}, \texttt{Smoking} | \texttt{Smoking} = \texttt{smoker})$

in Fig 3.12 and is denoted $P(\texttt{Bronchitis}, \texttt{Cancer}, \texttt{Smoking} | \texttt{Smoking} = \texttt{smoker})$. Note that a conditional distribution is a probability distribution just like any other, the only thing that makes it 'conditional' is that it is formed by slicing a 'bigger' distribution and then renormalising.

gPy **Example 10 (Compute a conditional probability distribution)** *To get the conditional distribution displayed in Fig 3.12 do:*

```
>>> fr /= fr.z()
>>> print fr
```

`fr.z()` returns the sum of all the numbers in the table `fr` (this is 0.5). The `/=` operator just does in-place scalar division.

In particular, conditional distributions can be marginalised to project down onto subsets of the variables. Fig 3.13 shows distributions formed by marginalising away various variables.

gPy **Example 11 (Compute a series of marginal distributions)** *To produce the series of tables in Fig 3.12 do:*

```
>>> print fr.copy().sumout(['Bronchitis','Cancer'])
>>> print fr.copy().sumout(['Smoking'])
>>> print fr.copy().sumout(['Smoking','Bronchitis'])
>>> print fr.copy().sumout(['Smoking','Cancer'])
```

Note that a copy of the `fr` object is used since the `sumout` method alters the object it is called on and we want to leave `fr` intact so that it is available to compute all 4 marginal distributions.

Note from Fig 3.12 that summing out `Smoking` amounts to just dropping the `Smoking` column since `Smoking` has already been restricted to a single value.

```
Smoking |
------- | ----
smoker  |   1.00


Bronchitis | Cancer  |
---------- | ------- | ----
absent     | absent  |   0.00
absent     | present |   0.30
present    | absent  |   0.00
present    | present |   0.70


Cancer  |
------- | ----
absent  |   0.00
present |   1.00


Bronchitis |
---------- | ----
absent     |   0.30
present    |   0.70
```

Figure 3.13: Marginal distributions produced from the conditional distribution in Fig 3.12

Summing out all variables apart from `Smoking` gives the first distribution in Fig 3.12 which reveals the unsurprising fact that $P(\texttt{Smoking} = \texttt{smoker}|\texttt{Smoking} = \texttt{smoker}) = 1$. Note that had the variable `Smoking` been removed by the slicing operation (which recall was the first step to conditioning) then it would not be possible to show that $P(\texttt{Smoking} = \texttt{smoker}|\texttt{Smoking} = \texttt{smoker}) = 1$ since the variable `Smoking` would have disappeared.

Having described a conditional distribution informally and operationally a more formal definition will now be given. Recall that a probability distribution P assigns a probability to events, each of which is a subset of some sample space Ω.

Definition 1 Any event $A \subset \Omega$ such that $P(A) > 0$ defines a conditional probability distribution, denoted $P(.|A)$, as follows:

$$\forall B \subset \Omega : P(B|A) = \frac{P(A \cap B)}{P(A)} \tag{3.2}$$

□

The reason for defining conditional distributions is that they provide a way to update probability distributions when new information comes in. For example, suppose the distribution given in Fig 3.8 represented a person's—or more generally an agent's—degrees of belief about the likely characteristics of the next patient to walk into a health clinic. Suppose the next patient walks in and the agent discovers that this patient is a smoker (and nothing more). How should the agent update its probability distribution? Evidently all the joint instantiations with `Smoker=nonsmoker` must be altered to have zero probability, but what about the surviving joint instantiations with `Smoker=smoker`? What constitutes the correct thing to do is an issue for the philosophical area known as *probability kinematics*. However, the standard answer is that it is best to take a conservative approach: all the surviving probabilities should be increased proportionally so that they add up to one. This is precisely what (3.2) does.

Taking shortcuts when conditioning

The careful reader will have noticed that, strictly speaking, the table displayed in Fig 3.12 is not the conditional distribution $P(\texttt{Bronchitis}, \texttt{Cancer}, \texttt{Smoking}|\texttt{Smoking} = \texttt{smoker})$. As can be seen from the definition (3.2) conditioning does not remove any events from the sample space, it merely alters probabilities. However, Fig 3.12 only has probabilities for 4 (simple) events when there are 8 simple events in total. The formally correct conditional distribution is given in Fig 3.14, where we see that the rows missing from Fig 3.12 are those for which `Smoking=nonsmoker` all of which have value zero.

So Fig 3.12 is an informal representation of the conditional distribution where the events which must have probability zero (because they are inconsistent with the conditioning event) have been deleted. Such an approach amounts to not even considering such events as possibilities rather than the formally correct approach of continuing to consider them but assigning them zero probability. There are big computational savings to be had by taking the informal approach:

Bronchitis	Cancer	Smoking	
absent	absent	nonsmoker	0.00
absent	absent	smoker	0.00
absent	present	nonsmoker	0.00
absent	present	smoker	0.30
present	absent	nonsmoker	0.00
present	absent	smoker	0.00
present	present	nonsmoker	0.00
present	present	smoker	0.70

Figure 3.14: Conditional distribution $P(\texttt{Bronchitis}, \texttt{Cancer}, \texttt{Smoking} | \texttt{Smoking} = \texttt{smoker})$ with all rows displayed. An abbreviated version of this distribution is in Fig 3.12.

we get a more compact representation and, as will be seen later, we avoid the cost of many pointless multiplications by zero. For these reasons gPy (in common with other software for manipulating graphical models) effects conditioning by simple row deletion. From now on conditional distributions will be represented using the informal 'deleted row' representation, but it is important to remember that this is for computational convenience and contradicts the formal definition of conditioning.

gPy **Example 12 (Removing instantiations when conditioning)** *The following Python sesssion shows that, when conditioning, gPy removes instantiations inconsistent with the conditioning event. From the for loops it is evident that the fr object somehow 'contains' a single table: why this is will be discussed later. For now the important point is that this table contains a row for Bronchitis=present, Cancer=present, Smoking=smoker but not for Bronchitis=present, Cancer=present, Smoking=nonsmoker. Trying to access the value for the latter (non-existent) row causes a Python KeyError.*

```
cscipc001 ~/godot/teaching/modules/current/agm/book python
Python 2.4.3 (#1, Jul 26 2006, 20:13:39)
[GCC 3.4.6] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> from gPy.Examples import *
>>> probs = contab.normalised()
>>> fr = FR([probs.copy(True)])
>>> print fr.condition({'Smoking':['smoker']})

Bronchitis | Cancer  | Smoking |
```

```
---------- | ------- | ------- | ----
absent     | absent  | smoker  |   0.00
absent     | present | smoker  |   0.15
present    | absent  | smoker  |   0.00
present    | present | smoker  |   0.35


>>> for f in fr:
...  print f[{'Bronchitis':'present','Cancer':'present','Smoking':'smoker'}]
...
0.35
>>> for f in fr:
...  print f[{'Bronchitis':'present','Cancer':'present','Smoking':'nonsmoker'}]
...
Traceback (most recent call last):
  File "<stdin>", line 2, in ?
  File "/home/jc/godot/research/gPy/gPy/Parameters.py", line 205, in __getitem__
    return self._data[self._decode_inst(inst)]
  File "/home/jc/godot/research/gPy/gPy/Parameters.py", line 633, in _decode_inst
    raise KeyError("%s has no value called %s" % (name,val))
KeyError: 'Smoking has no value called nonsmoker'
>>>
```

Chapter 4

Factored representations of probability distributions

So far probability distributions have been represented using a single table mapping each joint instantiation to its probability. The benefit of such an approach is its simplicity, however the big problem with it is that the size of such a representation grows exponentially with the number of variables in the distribution. The key idea behind graphical models is to take advantage of structure in probability distributions to represent distributions not as a single object but as a *product*.

Before examining how to do this for probability distributions consider the advantages of representing *natural numbers* as products. Recall that any natural number can be represented as the unique product of its prime factors. For example,

$$84 = 2 \times 2 \times 3 \times 7$$

There are advantages to this factored representation of numbers. For example, if we were to multiply by, say, 5 then no real work needs to be done. We just update to:

$$2 \times 2 \times 3 \times 7 \times 5$$

Similarly, maintaining the factored representation makes it immediately obvious that the number in question is not divisible by, say, 13, but is divisible by, for example, 3. Moreover, dividing by 3 is easy:

$$2 \times 2 \times 7 \times 5$$

4.1 Factors

Although one should be careful not to draw too close an analogy between products of tables and products of numbers, there is a common theme: using a product representation to represent structure. It is thus not surprising that

Bronchitis	Cancer	Smoking
absent	absent	nonsmoker
absent	absent	smoker
absent	present	nonsmoker
absent	present	smoker
present	absent	nonsmoker
present	absent	smoker
present	present	nonsmoker
present	present	smoker

Figure 4.1: A table with variables $\Delta = \{\texttt{Bronchitis}, \texttt{Cancer}, \texttt{Smoking}\}$

similar nomenclature is used in both cases. So far, the word 'table' has been used rather loosely for both contingency tables and joint probability distributions. It is now time to give a more precise definition of *table* and introduce a new closely related term: *factor*. The terminology and notation given in [9] is followed.

Definition 2 Let Δ be a set of variables. For each variable $\delta \in \Delta$, define $\mathcal{I}_\delta$ to be the variable's values. Let $\mathcal{I} = \times_{\delta \in \Delta} \mathcal{I}_\delta$ be called a *table*. Each element i of $\mathcal{I}$ is a vector of values and is called a *cell*. A *factor* with table $\mathcal{I}$ is a function $f : \mathcal{I} \to \mathcal{R}_{\geq 0}$, where the function's values are explicitly tabulated. Both $\mathcal{I}$ and f are said to have *dimension* $|\Delta|$. □

For example, Fig 4.1 is a table where $\Delta = \{\texttt{Bronchitis}, \texttt{Cancer}, \texttt{Smoking}\}$ with $\mathcal{I}_{\texttt{Bronchitis}} = \{\texttt{absent}, \texttt{present}\}$, $\mathcal{I}_{\texttt{Cancer}} = \{\texttt{absent}, \texttt{present}\}$ and $\mathcal{I}_{\texttt{Smoking}} = \{\texttt{nonsmoker}, \texttt{smoker}\}$. The cells of the table are represented by the 8 rows. Note that the particular visual representation of the table given in Fig 4.1 (each cell represented by a row, variables and values both alphabetically ordered) is `gPy` specific.

A factor with the table from Fig 4.1 is represented in Fig 4.2. This factor is in fact the running example joint distribution from Chapter 3. However, it is important to remember that factors do not have to be probability distributions: any mapping from the cells of a table to the non-negative reals is a factor: for example, contingency tables are factors. Also note that the visual representation of a factor given in Fig 4.2 is `gPy` specific.

Note that the definition of a factor is rather odd, since a requirement is placed on how the function is *represented*. Moreover, this representational requirement is rather loose. This is, however, how the term is generally used in graphical modelling. In `gPy` there are no fewer than 3 choices of data structure to represent functions from tables to the reals. A `gPy.Parameters.Factor` object explicity lists, for every joint instantiation in the factor's table, the value in $\mathcal{R}_{\geq 0}$ for that instantiation. Such objects thus meet the requirements of Definition 2 to be factors. A `gPy.Parameters.CompactFactor` uses an ADTree representation

```
Bronchitis | Cancer  | Smoking   |
---------- | ------- | --------- | ----
absent     | absent  | nonsmoker |   0.03
absent     | absent  | smoker    |   0.00
absent     | present | nonsmoker |   0.27
absent     | present | smoker    |   0.15
present    | absent  | nonsmoker |   0.02
present    | absent  | smoker    |   0.00
present    | present | nonsmoker |   0.18
present    | present | smoker    |   0.35
```

Figure 4.2: A factor (representing a joint probability distribution)

[11] to store these values more compactly and is used mainly to represent large contingency tables of data. Finally, a `gPy.Models.FR` object represents real-valued functions as a 'product' of `gPy.Parameters.Factor` objects. This factor multiplication is the topic of Section 4.3.

4.2 Independence

Our goal is to represent joint probability distributions as products of factors, but as hinted above, for this to be possible the distribution so represented must have 'structure'. In this section we examined what sort of structure is required. The required property is that of *independence* which is now defined.

Definition 3 Two events $A, B \subset \Omega$ are *independent* in a probability distribution P iff

$$P(A \cap B) = P(A)P(B) \tag{4.1}$$

□

To define independence between random variables it is necessary to formally define what a *marginal distribution* is, and to define some useful notation.

Definition 4 First some notation:

- An expression $P(X_1, X_2, \ldots X_n)$ denotes a joint probability distribution P over variables X_1, X_2 and X_n. It is thus a function from joint instantiations of these variables to $\mathcal{R}_{\geq 0}$.
- An expression $P(X_1 = x_1, X_2 = x_2, \ldots X_n = x_n)$ denotes the value the joint probability distribution P gives to the joint instantiation $(X_1 = x_1, X_2 = x_2, \ldots X_n = x_n)$. It is thus a single number in $\mathcal{R}_{\geq 0}$. Where the variables are obvious $P(X_1 = x_1, X_2 = x_2, \ldots X_n = x_n)$ is abbreviated to $P(x_1, x_2, \ldots x_n)$.

Let $X_{i_1}, \ldots X_{i_m}$ and $X_{i'_1}, \ldots X_{i'_{m'}}$ be a partition of the variables $X_1, \ldots X_n$. The *marginal distribution* produced from $P(X_1, \ldots X_n)$ by marginalising away $X_{i'_1}, \ldots X_{i'_{m'}}$ is denoted by:

$$\sum_{X_{i'_1}, \ldots X_{i'_{m'}}} P(X_1, \ldots X_n)$$

(when we wish to emphasise which variables have been summed out) and also by:

$$P(X_{i_1}, \ldots X_{i_m})$$

(where we wish to emphasise which variables remain). The marginal distribution $P(X_{i_1}, \ldots X_{i_m})$ is defined as follows:

$$P(X_{i_1} = x_1, \ldots X_{i_m} = x_m) = \sum_{x'_1 \in \mathcal{I}_{X_{i'_1}}, \ldots, x'_{m'} \in \mathcal{I}_{X_{i'_{m'}}}} P(X_{i_1} = x_1, \ldots X_{i_m} = x_m, X_{i'_1} = x'_1, \ldots X_{i'_{m'}} = x'_{m'})$$

Marginalisation produces a distribution defined on a different sample space. The orginal function is a mapping from the n-dimensional table $\mathcal{I}_{X_1} \times \cdots \times \mathcal{I}_{X_n}$ to $\mathcal{R}_{\geq 0}$ but that the marginal distribution $P(X_{i_1}, \ldots X_{i_m})$ is a mapping from the m-dimensional table $\mathcal{I}_{X_{i_1}} \times \cdots \times \mathcal{I}_{X_{i_m}}$ to $\mathcal{R}_{\geq 0}$ where $m = n - m'$ □

Definition 5 Two random variables X and Y are *independent* in a joint distribution P iff

$$\forall x \in \mathcal{I}_X, y \in \mathcal{I}_Y : P(X = x, Y = y) = P(X = x)P(Y = y) \tag{4.2}$$

This condition can be written as:

$$P(X, Y) = P(X)P(Y) \tag{4.3}$$

($P(X)P(Y)$ is the product of two distributions with different variables. Such a product will be defined shortly and it will be seen that the definition is such that (4.2) and (4.3) are indeed equivalent.) □

The intuitive idea behind independence is that if two events are independent, then finding out whether one is true or false does not alter the probability of the other. It is not difficult to see that if A and B are independent events with positive probability then $P(A|B) = P(A)$ and $P(B|A) = P(B)$. In the case of random variables note that independence is a very strong condition: $P(X = x, Y = y)$ must equal $P(X = x)P(Y = y)$ *for all* values of x and y. Just one case to the contrary is enough for independence not to hold. If X and Y are independent it follows that $\forall x \in \mathcal{I}_X, y \in \mathcal{I}_Y : P(Y = y) \Rightarrow P(X = x|Y = y) = P(X = x)$. So if two random variables are independent, discovering the value of one gives no information about the other.

A joint distribution made up of *independent* random variables has just the sort of structure which enables compact representation. (We will later see that the more complex relationship of *conditional independence* also permits compact representation, but we will remain with the simpler notion of independence for now.) To illustrate this, consider the following example. Suppose we have two packs of cards and an agent is to select one card at random from each pack. *If we are prepared to assume that which card is selected first has no influence on which is selected second* then one option to model this situation is to use two independent random variables `Card1` and `Card2` to represent the two selected cards. `Card1` and `Card2` will have 52 values, and being computer scientists let's represent these 52 values using the numbers $0, 1, \ldots 51$. Now consider the joint probability distribution $P(\texttt{Card1}, \texttt{Card2})$. Evidently this distribution has $52 \times 52 = 2704$ simple events (i.e. joint instantiations), and, due to the independence assumption, $P(\texttt{Card1} = i, \texttt{Card2} = j) = P(\texttt{Card1} = i)P(\texttt{Card2} = j)$.

A single `Factor` object representing $P(\texttt{Card1}, \texttt{Card2})$ would thus contain 2704 values. Much better is to represent the joint distribution *implicitly* by storing two `Factor` objects for $P(\texttt{Card1})$ and $P(\texttt{Card2} = j)$ with the understanding that the joint distribution is a *product* of these factors. In other words instead of storing $P(\texttt{Card1} = i, \texttt{Card2} = j)$ for all i, j, only $P(\texttt{Card1} = i)$ and $P(\texttt{Card2} = j)$ are stored. If $P(\texttt{Card1} = i, \texttt{Card2} = j)$ is needed it can be computed by multiplying $P(\texttt{Card1} = i)$ and $P(\texttt{Card2} = j)$. This approach requires only $52 + 52 = 104$ numbers. This example has been chosen to show that big savings are possible by representing a big factor (i.e. lots of cells in its table) as the product of two others. The big factor in question is too big to display in the text, so use `gPy` Example 13 to have a look at it!

`gPy` Example 13 (Implicit representation of a big factor)
A factor representing a distribution for ***`Card1`*** *can be inspected using:*

```
>>> from gPy.Examples import card1, card2
>>> import gPy.Parameters
>>> gPy.Parameters.precision = 6
>>> print card1
```

Note that, for simplicity, ***`Card1`*** *has been given a uniform distribution with all probabilities* $= 1/52$*, but that any distribution for* ***`Card1`*** *can be represented with such a factor.* ***`Card2`*** *is similar. To generate a single* ***`Factor`*** *object explicitly representing the distribution* $P(\texttt{Card1}, \texttt{Card2})$ *do:*

```
print card1 * card2
```

4.3 Factor multiplication

In the previous section a case was made for representing joint distributions as products of factors as opposed to a single (potentially huge) `Factor` object, but what it means to multiply factors was only outlined. In this section a more precise account is given of what it means to 'multiply' factors.

Recall that factors are functions (from tables to the reals). If two real-valued functions are defined on the same domain the there is already a standard definition of their product as given in Definition 6.

Definition 6 Let $f_1 : D \to \mathcal{R}$ and $f_2 : D \to \mathcal{R}$ be two real-valued functions where D is some arbitrary domain. Then $f_1 f_2$, the product of the two functions is defined by pointwise multiplication:

$$\forall d \in D : (f_1 f_2)(d) = f_1(d) f_2(d)$$ □

If two factors have the same table, then their product is defined in the standard pointwise-multiplication way as described by Definition 6. If they have different tables a little more work has to be done. To explain factor multiplication in the general case it is useful first to formalise the notion of a *marginal table* and then define the notion of *broadcasting*.

Marginal tables have already been seen in the explanation of marginalisation of contingency tables and probability distributions. A selection of marginal tables produced from the one displayed in Fig 4.1 is given in Fig 4.3. It is evident that each cell in the original table is projected onto a unique cell in the marginal table: for example the cell ($\texttt{Bronchitis} = \texttt{absent}, \texttt{Cancer} = \texttt{absent}, \texttt{Smoking} = \texttt{smoker}$) in the original table maps (respectively, with an ordering from top to bottom) to cells ($\texttt{Smoking} = \texttt{smoker}$), ($\texttt{Bronchitis} = \texttt{absent}, \texttt{Cancer} = \texttt{absent}$), ($\texttt{Cancer} = \texttt{absent}$), ($\texttt{Bronchitis} = \texttt{absent}$) in the marginal tables in Fig 4.3. Definition 7 gives the formal definition of a marginal table.

Definition 7 Let $\mathcal{I} = \times_{\delta \in \Delta} \mathcal{I}_\delta$ be a table with variables Δ. Let $a \subset \Delta$. The *a-marginal table* $\mathcal{I}_a$ is obtained by only considering the variables in a; formally:

$$\mathcal{I}_a = \times_{\delta \in \Delta} \mathcal{I}_\delta$$

Any cell in $i \in \mathcal{I} (= \times_{\delta \in \Delta} \mathcal{I}_\delta)$ is mapped to a unique *marginal cell* in $i_a \in \mathcal{I}_a$ by projecting i onto just the variables in a. [9] □

Having defined marginal tables it is now easy to define *broadcasting* of factors. In contrast to marginalisation, where the dimension of a factor is decreased, broadcasting increases (or occasionally keeps constant) a factor's dimension. Any extra values required by an increase in dimension are provided by repeating values. To define broadcasting formally some new notation is useful: if f is a factor denote f's table as $\mathcal{I}(f)$ and f's variables as $\Delta(f)$. Definition 8 then defines broadcasting formally and Fig 4.4 provides an example.

Smoking
smoker
nonsmoker

Bronchitis	Cancer
absent	absent
absent	present
present	absent
present	present

Cancer
absent
present

Bronchitis
absent
present

Figure 4.3: Marginal tables produced from the table in Fig 4.1. Each is a a-marginal table, with (ordering from top to bottom) $a = \{\texttt{Smoking}\}, \{\texttt{Bronchitis}, \texttt{Cancer}\}, \{\texttt{Cancer}\}, \{\texttt{Bronchitis}\}$, respectively.

Bronchitis	
absent	0.30
present	0.70

Bronchitis	Cancer	Smoking	
absent	absent	nonsmoker	0.30
absent	absent	smoker	0.30
absent	present	nonsmoker	0.30
absent	present	smoker	0.30
present	absent	nonsmoker	0.70
present	absent	smoker	0.70
present	present	nonsmoker	0.70
present	present	smoker	0.70

Figure 4.4: The factor on the right is produced by broadcasting the factor on the left

```
Cancer  |                    VisitAsia |
------- | ----               --------- | ----
absent  |   0.95             no_visit  |   0.99
present |   0.06             visit     |   0.01

broadcast to ..              broadcast to ...

Cancer  | VisitAsia |        Cancer  | VisitAsia |
------- | --------- | ----   ------- | --------- | ----
absent  | no_visit  |   0.95 absent  | no_visit  |   0.99
absent  | visit     |   0.95 absent  | visit     |   0.01
present | no_visit  |   0.06 present | no_visit  |   0.99
present | visit     |   0.06 present | visit     |   0.01

Pointwise multiplication ...

Cancer  | VisitAsia |          Cancer  | VisitAsia |          Cancer  | VisitAsia |
------- | --------- | ----     ------- | --------- | ----     ------- | --------- | ----
absent  | no_visit  |   0.95   absent  | no_visit  |   0.99   absent  | no_visit  |   0.94
absent  | visit     |   0.95 * absent  | visit     |   0.01 = absent  | visit     |   0.01
present | no_visit  |   0.06   present | no_visit  |   0.99   present | no_visit  |   0.05
present | visit     |   0.06   present | visit     |   0.01   present | visit     |   0.00
```

Figure 4.5: Factor multiplication (with rounding errors!) by broadcasting followed by pointwise multiplication

Definition 8 Let f be a factor and let Δ' be a set of variables such that $\Delta' \supseteq \Delta(f)$. Let $\mathcal{I}' = \times_{\delta \in \Delta'} \mathcal{I}_\delta$. Define $f' : \mathcal{I}' \to \mathcal{R}_{\geq 0}$ to be the function

$$\forall i' \in \mathcal{I}' : f'(i') = f(i'_\Delta)$$

then f' is the factor produced by *broadcasting* f to table $\mathcal{I}'$. □

Factor multiplication is now easy to define. Given two factors f_1, f_2 defined using variables Δ_1 and Δ_2, respectively: first broadcast both to have variables $\{\Delta_1 \cup \Delta_2\}$ giving factors f'_1 and f'_2 with the same table. The product of f_1 and f_2 is then given by pointwise-multiplication of f'_1 and f'_2. Fig 4.5 shows the steps in factor multiplication. These factors can also be displayed via `gPy` Example 14. Definition 9 gives the formal general definition of factor multiplication.

`gPy` **Example 14 (Factor multiplication)**
First create the factors `cancer` *and* `visit` *like this:*

```
>>> from gPy.Examples import asia
>>> cancer = asia['Smoking'] * asia['Cancer']
>>> cancer = cancer.sumout(['Smoking'])
>>> visit = asia['VisitAsia'] * 1
```

Have a look at them with:

```
>>> print cancer

Cancer  |
------- | ----
absent  |   0.95
present |   0.06

>>> print visit

VisitAsia |
--------- | ----
no_visit  |   0.99
visit     |   0.01
```

It is instructive to do factor multiplication 'by hand'. First create an appropriate set of variables to broadcast to (| is Python syntax for set union):

```
>>> all_variables = visit.variables() | cancer.variables()
>>> all_variables
frozenset(['VisitAsia', 'Cancer'])
```

Now have a look at the broadcast factors (making a copy to leave the original factors intact):

```
>>> print visit.copy().broadcast(all_variables)

Cancer  | VisitAsia |
------- | --------- | ----
absent  | no_visit  |   0.99
absent  | visit     |   0.01
present | no_visit  |   0.99
present | visit     |   0.01

>>> print cancer.copy().broadcast(all_variables)

Cancer  | VisitAsia |
------- | --------- | ----
absent  | no_visit  |   0.95
absent  | visit     |   0.95
present | no_visit  |   0.06
present | visit     |   0.06
```

Finally, have a look at the product:

```
>>> print cancer * visit

Cancer  | VisitAsia |
------- | --------- | ----
absent  | no_visit  |   0.94
absent  | visit     |   0.01
present | no_visit  |   0.05
present | visit     |   0.00
```

A lazier option is just to use a canned demo:

```
>>> from gPy.Examples import factor_mult
>>> factor_mult()
```

Definition 9 Let f_1 and f_2 be factors. Let f_1' and f_2' be the factors produced by broadcasting f_1 and f_2 to both have variables $\Delta(f_1) \cup \Delta(f_2)$. Let $\mathcal{I} = \times_{\delta \in (\Delta(f_1) \cup \Delta(f_2))}$ The *product of* f_1 *and* f_2 is then defined as follows:

$$\forall i \in \mathcal{I} : (f_1 f_2)(i) = f_1'(i) f_2'(i) \qquad \square$$

Now that factor multiplication has been defined properly it is worth returning to the example of two independent cards. It is evident that if f_1 is a factor representing the distribution $P(\texttt{Card1})$ and f_2 is another factor representing $P(\texttt{Card2})$ then the factor $f_1 f_2$ will represent the joint distribution $P(\texttt{Card1}, \texttt{Card2})$, if the two random variables `Card1` and `Card2` are independent. This is because, assuming independence, $P(\texttt{Card1} = i, \texttt{Card2} = j) = P(\texttt{Card1} = i) P(\texttt{Card2} = j)$: independence is defined pointwise, and this is why the pointwise multiplication in factor multiplication gives rise to the appropriate factor. As another example, the factor in the bottom right corner of Fig 4.5 represents the correct joint distribution $P(\texttt{Cancer}, \texttt{VisitAsia})$ on the assumption that (i) these two random variables are independent and (ii) the two factors at the top of Fig 4.5 represent the two relevant marginal distributions.

In the general case suppose that $X_1, X_2, \ldots, X_n$ are all mutually independent variables with marginal distributions represented by factors $f_1, f_2, \ldots, f_n$, then the joint distribution is represented by the factor $f_1 f_2 \ldots f_n$. Crucially, the joint distribution can be represented without ever constructing the `Factor` object $f_1 f_2 \ldots f_n$, it is enough to store f_1, f_2 and f_n (this is how a `FR` object is implemented). This allows a great saving in storage space. Suppose that each X_i has m values, then the table for $f_1 f_2 \ldots f_n$ has m^n cells, whereas each individual factor f_i has a table with m cells giving nm cells in total.

```
Bronchitis | Cancer  |            Bronchitis | Smoking   |
---------- | ------- | ----       ---------- | --------- | ----
absent     | absent  |   1.05     absent     | nonsmoker |   0.70
absent     | present |   0.05  *  absent     | smoker    |   0.40
present    | absent  |   0.84     present    | nonsmoker |   0.30
present    | present |   0.06     present    | smoker    |   0.60

   Bronchitis | Cancer  | Smoking   |
   ---------- | ------- | --------- | ----
   absent     | absent  | nonsmoker |   0.74
   absent     | absent  | smoker    |   0.42
   absent     | present | nonsmoker |   0.03
 = absent     | present | smoker    |   0.02
   present    | absent  | nonsmoker |   0.25
   present    | absent  | smoker    |   0.50
   present    | present | nonsmoker |   0.02
   present    | present | smoker    |   0.04
```

Figure 4.6: Factor multiplication with overlapping variables. Note that the first factor has a value above 1 and thus does not represent a marginal distribution.

4.4 Defining distributions with factors

So far we have used factors to define joint distributions in the very limited case where all variables are mutually independent. In such cases there is no overlap between the variables of the factors being multiplied. But from its definition (Definition 9) factor multiplication is defined in the general case where variables do overlap. Fig 4.6 shows an example of factor multiplication in such a case. Three further examples can be displayed by following `gPy` Example 15.

Since factor multiplication is defined for arbitrary factors, it is not difficult to see that *almost any* collection of factors defines a joint probability distribution. Let $f_1, f_2, \ldots, f_n$ be an arbitrary set of factors and let $f = f_1 f_2 \ldots f_n$ be their product. Let Δ_i denote the variables used in f_i, then clearly f has $\bigcup_i \Delta_i$ as its variables. Let $\Delta = \bigcup_i \Delta_i$. f will be a function mapping each joint instantiation of Δ to a non-negative real number, but it will not in general be a joint probability distribution, since these numbers will generally not add up to 1. The bottom factor in Fig 4.6 is an example of a factor whose values do not sum to 1. However, even if not all non-negative real functions are probability distributions, almost any such function *determines* a unique probability distribution: the one produced by normalising it so that all values sum to 1. The exceptional cases are those functions all of whose values are zero. Call a factor all of whose values are zero a *zero factor* and any factor containing at least one non-zero value a *non-zero factor*. So any collection of factors containing at least one non-zero factor determines a probability distribution: it is the distribution produced by multiplying all the factors and then normalising the

resulting (non-zero) factor to get a probability distribution.

To define this process formally some new notation concerning normalisation (and marginalisation) will be useful. Suppose f is some factor with variables A, B and C. A new (marginal) factor, call it f', can be produced by, say, summing out A and B. f' will be denoted as:

$$f' = \sum_{A,B} f \tag{4.4}$$

of, if we wish to emphasise the variables involved in a factor, like this:

$$f'(C) = \sum_{A,B} f(A, B, C) \tag{4.5}$$

This notation is quite standard and reasonably clear, but it is a little informal. For example, writing $f(A, B, C)$ is a statement that A, B and C are the variables used to define f's table, but be careful not to interpret f as being a function of (A, B, C), it is a function from vectors of *values* of these variables to the reals, not a function of the variables themselves. Similarly, care must be taken when interpreting an equation such as $f' = \sum_{A,B} f$. Remember that this represents marginalisation of one factor (which is a function) to produce another factor (again a function).

To normalise any factor it is necessary first to add up the values associated with each cell in its table. For a factor f denote this non-negative real number by $Z(f)$. To normalise f, f is replaced by $f/Z(f)$ where $f/Z(f)$ is produced by dividing each value in f by the scalar $Z(f)$. With this notation Definition 10 formally defines factored representations.

Definition 10 The *probability distribution determined by* a collection of factors $(f_1, f_2, \ldots, f_n)$, containing at least one non-zero factor, is

$$P = Z^{-1} f_1 f_2 \ldots f_n$$

where $Z = Z(f_1 f_2 \ldots f_n)$. In this case we say $(f_1, f_2, \ldots, f_n)$ is a *factored representation* of P. Z is called the *the normalising constant for the factored representation.* □

gPy **Example 15 (Factor multiplication)**
To see 3 examples of factor multiplication with overlapping variables do:

```
>>> from gPy.Examples import prod_gen
>>> prod_gen()
```

Computing Z, the normalising constant, is computationally heavy. If, as is often the case, probabilities are only required up to a normalising constant then

```
A | B |          A | D |          B | C |          C | D |
- | - | ----     - | - | ----     - | - | ----     - | - | ----
0 | 0 |   0.10   0 | 0 |   0.90   0 | 0 |   0.40   0 | 0 |   0.50
0 | 1 |   0.20 * 0 | 1 |   0.20 * 0 | 1 |   0.70 * 0 | 1 |   0.20
1 | 0 |   0.30   1 | 0 |   0.70   1 | 0 |   0.30   1 | 0 |   0.40
1 | 1 |   0.20   1 | 1 |   0.10   1 | 1 |   0.10   1 | 1 |   0.10

Z = 0.2165, dividing the 1st factor by Z gives

A | B |          A | D |          B | C |          C | D |
- | - | ----     - | - | ----     - | - | ----     - | - | ----
0 | 0 |   0.46   0 | 0 |   0.90   0 | 0 |   0.40   0 | 0 |   0.50
0 | 1 |   0.92 * 0 | 1 |   0.20 * 0 | 1 |   0.70 * 0 | 1 |   0.20
1 | 0 |   1.39   1 | 0 |   0.70   1 | 0 |   0.30   1 | 0 |   0.40
1 | 1 |   0.92   1 | 1 |   0.10   1 | 1 |   0.10   1 | 1 |   0.10
```

Figure 4.7: Absorbing the normalising factor Z into a factor

Z need not be computed. (For reasons connected to statistical physics, Z is also known as the *partition function*. It makes sense to call it a function since it can be seen as a function of the values in the factors.) Note that *any* distribution P has at least one factored representation, namely (P) the representation using a single factor which explicitly tabulates the entire probability distribution.

4.4.1 Non-uniqueness of factored representations

Let $P = Z^{-1} f_1 f_2 \dots f_n$ so that $(f_1, f_2, \dots, f_n)$ is a factored representation of P. Now choose some arbitrary $i \in \{1, 2, \dots, n\}$ and let $f'_i = f_i/Z$, where f_i/Z is the factor produced by dividing each value of f_i by Z. Then clearly $P = f_1 f_2 \dots f'_i \dots f_n$. It follows that it is always possible to alter a factored representation so that the product of factors is exactly equal to the joint probability distribution without the normalising constant Z. The Z can be 'absorbed' into an arbitrary factor. Fig 4.7 gives an example of such an absorption and `gPy` Example 16 shows how to display this example using `gPy`.

`gPy` **Example 16 (Normalisation)**
To generate a demonstration of absorbing the Z normalising constant do:

```
>>> from gPy.Examples import norming
>>> norming()
```

Z is absorbed into the 3rd factor.

Also note that if $(f_1, f_2, \ldots, f_n)$ is a factored representation then so, for example, is $((f_1 f_2), \ldots, f_n)$. In general, it is obvious that any pair of factors can be replaced by that single factor which is their product. This process can continue until we end up with (P) as a (trivial) factored representation. The basic point is that there is a lot of 'room for manoeuvre' in factored representations. This flexibility is needed to allow efficient marginalisation of factored representations.

4.5 Marginalising factored representations

As argued previously, factored representations can save a lot of storage, but this naturally gives rise to the suspicion that we may have to pay for this saving in space when with a heavy price in time when it comes to performing computations with factored representations. In particular, how expensive is it to compute marginal distributions from factored representations? The good news is that it is *faster* to marginalise with a factored representation than with a single-factor representation of a probability distribution.

Factored representations can be marginalised efficiently due to the following facts:

1. If several variables are to be summed out, they can be summed out one at a time, in any order.
2. A subset of factors in a factored representation can be replaced by the single factor which is the product of these factors without altering the probability distribution represented.
3. If a variable appears in only one factor then summing it out of this factor is equivalent to summing it out of the probability distribution represented by the factored representation.

The first two points are fairly obvious and follows from elementary properties of addition and multiplication such as commutativity and associativity. The third point is less obvious and requires some justfication, as provided by Theorem 1.

Theorem 1 *Let $P(X_1, X_2, \ldots X_m) = Z^{-1} \prod_{i=1}^{n} f_i$ be a joint probability distribution over joint instantiations of the variables $X_1, X_2, \ldots, X_n$. Suppose that the variable X_1 only appears in factor f_1. Then*

$$P(X_2, \ldots, X_m) = Z^{-1} \left(\sum_{X_1} f_1 \right) \times \left(\prod_{i=2}^{n} f_i \right)$$

□

PROOF Consider an arbitrary joint instantiation $X_1 = x_1, X_2 = x_2, X_3 = x_3, \ldots, X_m = x_m$ which we will abbreviate to $(x_1, x_2, \ldots, x_m)$. Since $P = Z^{-1} \prod_{i=1}^{n} f_i$:

$$P(x_1, x_2, \ldots, x_m) = Z^{-1} f_1(x_1, \ldots) \prod_{i=2}^{n} f_i(\mathbf{x}_i)$$

where $\mathbf{x}_i$ denotes the joint instantiation $(x_1, x_2, \ldots, x_m)$ restricted to just the variables involved in factor f_i and where the expression $f_1(x_1, \ldots)$ indicates that factor f_1 has X_1 as a variable and possibly other variables as well. Note that the product $\prod_{i=2}^{n} f_i(\mathbf{x}_i)$ contains no factors involving the variables X_1. Thus

$$\begin{aligned} & P(x_2, \ldots, x_m) \\ = & \sum_{x_1 \in \mathcal{I}_{X_1}} P(x_1, x_2, \ldots, x_m) \\ = & \sum_{x_1 \in \mathcal{I}_{X_1}} \left(Z^{-1} f_1(x_1, \ldots) \prod_{i=2}^{n} f_i(\mathbf{x}_i) \right) \\ = & Z^{-1} \left(\sum_{x_1 \in \mathcal{I}_{X_1}} f_1(x_1, \ldots) \right) \prod_{i=2}^{n} f_i(\mathbf{x}_i) \end{aligned}$$

The last equation follows because x_1 does not appear in $\prod_{i=2}^{n} f_i(\mathbf{x}_i)$. Since $(x_2, \ldots x_m)$ was an arbitrary instantiation, it has been shown that

$$P(X_2, \ldots, X_m) = Z^{-1} \left(\sum_{X_1} f_1 \right) \times \left(\prod_{i=2}^{n} f_i \right)$$

■

For simplicity of exposition, Theorem 1 concerned variable X_1 appearing uniquely in factor f_1, but since it does not matter how variables and factors are numbered, Theorem 1 suffices to establish the more general claim that to sum out any variable that appears in just one factor it is enough to sum it out of that factor.

Armed with this result, an efficient algorithm for summing out variables from a factored representation is beginning to emerge. If a variable X_j to be summed out appears in just one factor f_i, replace f_i with $\sum_{X_j} f_i$ in the factored representation and we're done. But what to do if a variable appears in more than one factor? The answer is simple: if a variable X_j appears in k factors $f_{i_1}, f_{i_2}, \ldots f_{i_k}$ just replace these k factors with the single factor that is their product. Once this is done X_j does just appear in a single factor (the newly formed product) and can be summed out of the distribution by being summed out of this factor.

Now that an algorithm has been outlined the next step is to consider how it should be implemented (remember that this is a book for computer science students). This requires deciding on a data structure for factored representations, and it is clear that this data structure should make it quick to identify which factors a given variable is in. One natural way to do this is to make use of *hypergraphs*, the subject of the next section. Here the bare essentials of hypergraphs are presented and they are used merely to organise a marginalisation algorithm. Later we will see that they play a much more fundamental role in graphical models.

4.6 Introduction to hypergraphs

When analysing the structure of a factored representation, what matters is not so much the numbers in the factors but the relationship between the *variables* in the various factors. Consider the collection of subsets of the variables in a factored distribution: $\mathcal{H} = (\Delta(f_i))_i$. So, for example, in the factored representation from Fig 4.7, $\mathcal{H} = (\{A, B\}, \{B, C\}, \{C, D\}, \{A, D\})$. $\mathcal{H}$ ignores (i) the numbers in the factors and (ii) the values the variables can take and just focusses attention on the variables in the various factors. $\mathcal{H}$ is our first example of a *hypergraph*. Hypergraphs are very straightforward: a hypergraph is simply a collection of subsets of some set. Each subset is called a *hyperedge*. (Some authors sometimes refer to hyperedges as *edges* but in this book they are always called hyperedges.) Definition 11 provides the formal definition and `gPy` Example 17 shows how to create some example hypergraphs using `gPy`.

Definition 11 A *hypergraph* $\mathcal{H}$ is a collection of subsets of a finite set H, the *base set*. The elements $h \in \mathcal{H}$ are called the *hyperedges*. The elements of H are called *vertices*, so every hyperedge is a set of vertices. Following [9] it will be assumed throughout this book that $H = \cup_{h \in \mathcal{H}} h$, in other words, every vertex will be contained in some hyperedge. □

In general, hypergraphs are allowed to have repeated hyperedges [2, 7], so that the collection $(\{A, B\}, \{A, B\})$ constitutes a valid hypergraph. A hypergraph with no repeated hyperedges is called *simple* as formally defined in Definition 12.

Definition 12 A *simple hypergraph* is one without repeated hyperedges [2, 7].□

A simple hypergraph can be represented as a *set* of hyperedges and this representation will be adopted throughout for simple hypergraphs. Note that Lauritzen [9] uses the term 'simple hypergraph' to mean a hypergraph with exactly one hyperedge, a quite different meaning. In this book a 'simple hypergraph' is always as defined in Definition 12.

A simple hypergraph may still have *redundant hyperedges*: hyperedges that are subsets of other hyperedges. A hypergraph with no redundant hyperedges is *reduced*. Definitions 13–15 provide the necessary definitions.

Definition 13 A hyperedge $h \in \mathcal{H}$ is *redundant* if there exists another $h' \in \mathcal{H}$ such that $h \subseteq h'$. □

Definition 14 A hypergraph is *reduced* if it contains no redundant hyperedges. (So a reduced hypergraph is always simple.) Synonyms for 'reduced hypergraph' include: *clutter*, *antichain* and *Sperner system*. □

Definition 15 The *reduction* $\text{red}(\mathcal{H})$ of a hypergraph $\mathcal{H}$ is the reduced hypergraph produced by removing any redundant hyperedges from $\mathcal{H}$. □

It is clear that every factored representation $(f_1, f_2, \ldots f_n)$ has an associated hypergraph, and that every factor in a factored representation has an associated hyperedge. Definition 16 formalises this.

Definition 16 Let $(f_1, f_2, \ldots f_n)$ be a factored representation. The *hypergraph of the factored representation* is $\mathcal{H}((f_1, f_2, \ldots f_n)) = (\Delta(f_i))_{i=1,2,\ldots,n}$. The hyperedge $\Delta(f_i)$ is the *hyperedge for the factor* f_i. □

A factored representations is called *simple* if its associated hypergraph is simple, as defined in Definition 17. (The term 'simple factored representation' is not a standard one, and is, as far as I know, unique to this book.)

Definition 17 A factored representation is *simple* if its associated hypergraph is simple. It follow that if $(f_1, f_2, \ldots, f_n)$ is simple then $\forall i, j : i \neq j \Rightarrow \Delta(f_i) \neq \Delta(f_j)$. □

Similarly, a factored representations is called *reduced* if its associated hypergraph is reduced, as defined in Definition 18. (Again, the term 'reduced factored representation' is not a standard one.)

Definition 18 A factored representation is *reduced* if its associated hypergraph is reduced. It follow that if $(f_1, f_2, \ldots, f_n)$ is reduced then $\forall i, j : i \neq j \Rightarrow \Delta(f_i) \not\subseteq (f_j)$. □

gPy **Example 17 (Creating hypergraphs)**
In ***gPy****, general hypergraphs, simple hypergraphs and reduced hypergraphs are objects of class* ***HyperGraph****,* ***SimpleHyperGraph*** *and* ***ReducedHyperGraph*** *resepectively. Both classes are defined in the* ***Structures*** *module. To create some hypergraphs, first import these classes into your Python session:*

```
>>> from gPy.Structures import HyperGraph, SimpleHyperGraph, ReducedHyperGraph
```

Now instances of hypergraphs can be made. For example, to create the simple hypergraph $\{\{A, B\}, \{B, C\}, \{C, D\}, \{A, D\}\}$ *and print it out do:*

```
>>> hg1 = SimpleHyperGraph(['AB','BC','CD','AD'])
>>> print hg1
{ {B, C}, {A, D}, {A, B}, {C, D} }
```

(This works because ***gPy*** *expects a sequence of hyperedges, and so interprets e.g.* ***'AB'*** *as a hyperedge and thus* ***'A'*** *and* ***'B'*** *as vertices.) To create the non-simple hypergraph* $(\{A, B\}, \{B, C\}, \{C, D\}, \{A, D\}, \{A, D\})$ *do:*

```
>>> ns_hg = HyperGraph(['AB','BC','CD','AD', 'AD'])
>>> print ns_hg
( {B, C}, {A, D}, {A, D}, {A, B}, {C, D} )
```

There is no particular ordering of hyperedges so the ordering of hyperedges (and also the variables) in the call to ***HyperGraph*** *does not matter. (To get the documentation on how to create hypergraphs, type* `help(<class>.__init__)` *where* `<class>` *is* ***HyperGraph****,*

`SimpleHyperGraph` or `ReducedHyperGraph`. To see the gruesome details of how `HyperGraph` instances are implemented type `hg1` without the `print`.)

In this book the elements in the base set will typically be strings naming random variables. However, other base sets are permissible. Here's how to create a hypergraph using integers (and Python's builtin `range` function).

```
>>> hg2 = SimpleHyperGraph((range(8),range(1,4),range(1,6,2)))
>>> print hg2
{ {1, 2, 3}, {1, 3, 5}, {0, 1, 2, 3, 4, 5, 6, 7} }
```

Note that here a tuple of hyperedges was used, rather than a string as in the previous case. In general, any iterable structure (e.g. list, tuple, set, string) can be used.

Attempting to create a reduced hypergraph from a list of hyperedges containing a redundant hyperedge will cause an error:

```
>>> rg = ReducedHyperGraph(['AB','ABC','CD'])
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
  File "/home/jc/godot/research/gPy/gPy/Structures.py", line 1292, in __init__
    raise RedundancyError("%s is not reduced" % hypergraph)
gPy.Structures.RedundancyError: ( {A, B, C}, {A, B}, {C, D} ) is not
reduced
```

If you need to create a reduced hypergraph from hyperedges which may contain redundant ones use the `modify=True` optional argument. The redundant hyperedges are just removed:

```
>>> rg = ReducedHyperGraph(['AB','ABC','CD'],modify=True)
>>> print rg
{ {A, B, C}, {C, D} }
```

It is often important to find all the factors containing a given variable. The hypergraph notion of the *star* of a vertex in a hypergraph, as defined in Definition 19, can help in this task.

Definition 19 The *star* $\mathcal{H}(v)$ of a vertex v in a hypergraph $\mathcal{H}$ is the collection of all hyperedges containing v. If $\mathcal{H}$ is simple then $\mathcal{H}(v) = \{h \in \mathcal{H} : v \in h\}$. □

4.7 Data structures for factored representations

In `gPy` simple hypergraphs are objects with two attributes: (i) the set of hyperedges and (ii) a mapping from each element to its star. `gPy` Example 18 shows the attributes and some of the methods available for hypergraphs. (General hypergraphs have a third attribute mapping each distinct hyperedge to the

number of times it is repeated. To keep things simple the discussion will focus on simple hypergraph since they are simpler.)

gPy **Example 18 (Data structure for hypergraphs)**
Assuming that you still have ***hg1*** *from* ***gPy*** *Example 17 you can do the following to inspect the attributes of a* ***gPy*** *hypergraph:*

```
>>> vars(hg1)
{'_hyperedges':
set([frozenset(['C', 'B']), frozenset(['A', 'D']),
frozenset(['A', 'B']), frozenset(['C', 'D'])]),

'_star':
{'A': set([frozenset(['A', 'D']), frozenset(['A', 'B'])]),
'C': set([frozenset(['C', 'B']), frozenset(['C', 'D'])]),
'B': set([frozenset(['C', 'B']), frozenset(['A', 'B'])]),
'D': set([frozenset(['A', 'D']), frozenset(['C', 'D'])])}}
```

(I have added line breaks to the Python output for clarity.) So there is an attribute `hyperedges` *which is just the set of hyperedges where each hyperedge is an immutable* ***frozenset****. And there is also* `_star` *which is a dictionary mapping each element to its star.*

Hypergraphs can be inspected in various ways and be altered by adding and removing hyperedges:

```
>>> print hg1
{ {B, C}, {A, D}, {A, B}, {C, D} }
>>> hg1.star('A')
set([frozenset(['A', 'D']), frozenset(['A', 'B'])])
>>> hg1.add_hyperedge(['A'])
>>> print hg1
{ {B, C}, {A, D}, {A, B}, {A}, {C, D} }
>>> hg1.remove_hyperedge(['C','D'])
>>> print hg1
{ {B, C}, {A, D}, {A, B}, {A} }
>>> print hg1.star('A')
set([frozenset(['A', 'D']), frozenset(['A', 'B']), frozenset(['A'])])
```

Factored representations in `gPy` use hypergraphs extensively. A factored representation is an instance of class `FR`. If the factored representation is constrained to be simple then the subclass `SFR` is appropriate. Factored representations required to be reduced are represented as `RFR` instances. Objects of class `FR` (and of its subclasses) have a number of attributes, but the two of interest are: (i) a hypergraph and (ii) a mapping (implemented as a Python dictionary) asso-

ciating each hyperedge with its factor. (How factors are implemented is too low-level to get into.) gPy Example 19 shows how to inspect a gPy factored representation (FR instance).

gPy Example 19 (A factored representation and its hypergraph)
First grab an FR instance to play with:

```
>>> from gPy.Examples import nondecomp
```

Then print it out and check that it is indeed an FR instance:

```
>>> print nondecomp

A | B |
- | - | ----
0 | 0 |   0.10
0 | 1 |   0.20
1 | 0 |   0.30
1 | 1 |   0.20


A | D |
- | - | ----
0 | 0 |   0.90
0 | 1 |   0.20
1 | 0 |   0.70
1 | 1 |   0.10


B | C |
- | - | ----
0 | 0 |   0.40
0 | 1 |   0.70
1 | 0 |   0.30
1 | 1 |   0.10


C | D |
- | - | ----
0 | 0 |   0.50
0 | 1 |   0.20
1 | 0 |   0.40
1 | 1 |   0.10
```

```
>>> type(nondecomp)
<class 'gPy.Models.FR'>
```

The `FR` *instance* ***`nondecomp`*** *has a hypergraph attribute and a mapping (called* `_factors`*) from hyperedges of this hypergraph to its factors:*

```
>>> print nondecomp._hypergraph
( {B, C}, {A, D}, {A, B}, {C, D} )
>>> for hyperedge, factor in nondecomp.items():
...  print hyperedge, factor
...
frozenset(['C', 'B'])
B | C |
- | - | ----
0 | 0 |   0.40
0 | 1 |   0.70
1 | 0 |   0.30
1 | 1 |   0.10

frozenset(['A', 'D'])
A | D |
- | - | ----
0 | 0 |   0.90
0 | 1 |   0.20
1 | 0 |   0.70
1 | 1 |   0.10

frozenset(['A', 'B'])
A | B |
- | - | ----
0 | 0 |   0.10
0 | 1 |   0.20
1 | 0 |   0.30
1 | 1 |   0.20

frozenset(['C', 'D'])
C | D |
- | - | ----
0 | 0 |   0.50
0 | 1 |   0.20
1 | 0 |   0.40
1 | 1 |   0.10
```

4.8 Naïve variable elimination

Resurfacing from `gPy` specific choices of data structure, we are now in a position to present our first real algorithm. (Marginalisation, slicing and normalising of factors evidently all require an algorithm, but are too low-level to be of any interest.) This algorithm is *naïve variable elimination*. It simply sums out variables from a factored representation in a fixed order and is displayed in Algorithm 1.

Algorithm 1 Naïve variable elimination

```
def variable_elimination(self,variables):
    for variable in variables:
        self.eliminate_variable(variable)

def eliminate_variable(self,variable):
    prod_factor = 1
    hyperedges = self._hypergraph.star(variable)
    for hyperedge in hyperedges:
        prod_factor *= self.factor(hyperedge)
        self.remove(hyperedge)
    message = prod_factor.sumout([variable])
    self *= message
```

As can be seen from Algorithm 1, naïve variable elimination is a very simple algorithm. Both `variable_elimination` and `eliminate_variable` are `FR` methods so `self` is an `FR` instance—a factored representation of a probability distribution (it may additionally be an instance of the class `SFR` or `RFR`). `variable_elimination` just plods through the supplied sequence of `variables` eliminating (i.e. summing out) each `variable` in turn. To eliminate a single variable `eliminate_variable` uses the associated hypergraph to find the `hyperedges` containing the variable to be eliminated (the star of the variable). It then uses these `hyperedges` to grab and delete the associated factors while multiplying them together to get the new factor `prod_factor`. `variable` is then summed out of `prod_factor` producing a new factor `message` which is multiplied into the factored representation.

How the new factor `message` is 'multiplied in' to the factored representation `self` depends on the restrictions, if any, that have been placed on the representation. If the factored representation is unrestricted then the new factor is just added to the factors already present as described in Definition 20.

Definition 20 Let $(f_1, f_2, \ldots, f_n)$ be a factored representation. If no restrictions are placed on the factored representation then the result of *multiplying in* a new factor f is the factored representation $(f_1, f_2, \ldots, f_n, f)$ □

If the factored representation is required to be simple then factor multiplication may have to be performed to retain simplicity, as described in Definition 21.

Definition 21 Let $\{f_1, f_2, \ldots, f_n\}$ be a simple factored representation. If simplicity must be maintained then the result of *multiplying in* a new factor f depends on whether there is an existing factor f_i with the same variables as f. If there is such an f_i then the result is $\{f_1, f_2, \ldots (f_i f), \ldots, f_n\}$. If not the result is $\{f_1, f_2, \ldots, f_n, f\}$ □

If the factored representation is required to be reduced then factor multiplication may have to be performed to retain reducedness, as described in Definition 22.

Definition 22 Let $\{f_1, f_2, \ldots, f_n\}$ be a reduced factored representation. If reducedness must be maintained then the result of *multiplying in* a new factor f depends on which of these three cases obtain: (i) there is an existing factor f_i whose variables contain those of f, or (ii) the variables of f are a superset of the variables of one or more existing factors, or (iii) neither (i) nor (ii) is the case.

In case (i) the result is $\{f_1, f_2, \ldots (f_i f), \ldots, f_n\}$. In case (ii) the result is $\{f_1, f_2, \ldots (f' f'' \ldots f), \ldots, f_n\}$ where f contains the variables of factors $f', f'' \ldots$. In case (iii) the result is $\{f_1, f_2, \ldots, f_n, f\}$. □

Figs 4.8–4.15 provide an example of how a simple (but not reduced) factored representation changes as variables are eliminated from it. (Figs 4.13 and 4.14 provide an example of a new factor being multiplied into an existing one to maintain simplicity.) Fig 4.8 shows the original distribution: it is the 'Asia' Bayesian network from Section 1.1.1 viewed as a factored representation. As will be explained in Chapter 7, Bayesian networks are factored representations with certain restrictions: the chief one being that the factors are conditional probability tables (CPTs). The factors in Fig 4.8 are the same as the CPTs in Fig 1.1 but their visual representation is different—they are displayed like any other factor—since here it suits us to ignore the fact that they are CPTs and treat them just like any other factor.

Fig 4.15 displays the marginal distribution $P(\texttt{Cancer})$ produced by summing out all variables except `Cancer`. Normally naïve variable elimination only produces marginal distributions up to normalisation. This is simply because factored representations only represent distributions up to normalisation. However, in this particular case because the original factored representation happened to be a Bayesian network and because no conditioning was effected, no normalisation was required. In Section 4.9 an example of needing a final normalisation step is shown.

In Figs 4.8–4.15 the variables were summed out in this order: `VisitAsia, Tuberculosis, XRay, Dyspnea, Bronchitis, Smoking, TbOrCa`. Such an order is known as an *elimination ordering*. Clearly, due to the commutativity of addition, these variables can be summed out in any order to produce $P(\texttt{Cancer})$. The question then is: does it matter in which order the variables are summed out? In fact it matters a great deal: although the same result is obtained no matter what the order, the computational complexity of naïve variable elimination depends very much on the elimination ordering. To see this consider Figs 4.16–4.22 where the ordering `TbOrCa, VisitAsia, Tuberculosis,`

Bronchitis	Dyspnea	TbOrCa	
absent	absent	false	0.90
absent	absent	true	0.30
absent	present	false	0.10
absent	present	true	0.70
present	absent	false	0.20
present	absent	true	0.10
present	present	false	0.80
present	present	true	0.90

Bronchitis	Smoking	
absent	nonsmoker	0.70
absent	smoker	0.40
present	nonsmoker	0.30
present	smoker	0.60

Cancer	Smoking	
absent	nonsmoker	0.99
absent	smoker	0.90
present	nonsmoker	0.01
present	smoker	0.10

Cancer	TbOrCa	Tuberculosis	
absent	false	absent	1.00
absent	false	present	0.00
absent	true	absent	0.00
absent	true	present	1.00
present	false	absent	0.00
present	false	present	0.00
present	true	absent	1.00
present	true	present	1.00

Smoking	
nonsmoker	0.50
smoker	0.50

TbOrCa	XRay	
false	abnormal	0.05
false	normal	0.95
true	abnormal	0.98
true	normal	0.02

Tuberculosis	VisitAsia		ABOUT TO BE USED
absent	no_visit	0.99	
absent	visit	0.95	
present	no_visit	0.01	
present	visit	0.05	

VisitAsia		ABOUT TO BE USED
no_visit	0.99	
visit	0.01	

Figure 4.8: The 'Asia' distribution as a product of factors. About to eliminate `VisitAsia` by multiplying the factors marked `ABOUT TO BE USED` and then summing out `VisitAsia`.

```
Bronchitis | Dyspnea | TbOrCa |
---------- | ------- | ------ | ----
absent     | absent  | false  |   0.90
absent     | absent  | true   |   0.30
absent     | present | false  |   0.10
absent     | present | true   |   0.70
present    | absent  | false  |   0.20
present    | absent  | true   |   0.10
present    | present | false  |   0.80
present    | present | true   |   0.90


Bronchitis | Smoking   |
---------- | --------- | ----
absent     | nonsmoker |   0.70
absent     | smoker    |   0.40
present    | nonsmoker |   0.30
present    | smoker    |   0.60


Cancer  | Smoking   |
------- | --------- | ----
absent  | nonsmoker |   0.99
absent  | smoker    |   0.90
present | nonsmoker |   0.01
present | smoker    |   0.10


Cancer  | TbOrCa | Tuberculosis |           ABOUT TO BE USED
------- | ------ | ------------ | ----
absent  | false  | absent       |   1.00
absent  | false  | present      |   0.00
absent  | true   | absent       |   0.00
absent  | true   | present      |   1.00
present | false  | absent       |   0.00
present | false  | present      |   0.00
present | true   | absent       |   1.00
present | true   | present      |   1.00


Smoking   |
--------- | ----
nonsmoker |   0.50
smoker    |   0.50


TbOrCa | XRay     |
------ | -------- | ----
false  | abnormal |   0.05
false  | normal   |   0.95
true   | abnormal |   0.98
true   | normal   |   0.02


Tuberculosis |                       NEW, ABOUT TO BE USED
------------ | ----
absent       |   0.99
present      |   0.01
```

Figure 4.9: The 'Asia' distribution after eliminating: `VisitAsia`. The new factor produced by eliminating `VisitAsia` is labelled `NEW`. About to eliminate `Tuberculosis` by multiplying the factors marked `ABOUT TO BE USED` and then summing out `Tuberculosis`.

Bronchitis	Dyspnea	TbOrCa	
absent	absent	false	0.90
absent	absent	true	0.30
absent	present	false	0.10
absent	present	true	0.70
present	absent	false	0.20
present	absent	true	0.10
present	present	false	0.80
present	present	true	0.90

Bronchitis	Smoking	
absent	nonsmoker	0.70
absent	smoker	0.40
present	nonsmoker	0.30
present	smoker	0.60

Cancer	Smoking	
absent	nonsmoker	0.99
absent	smoker	0.90
present	nonsmoker	0.01
present	smoker	0.10

Cancer	TbOrCa	NEW
absent	false	0.99
absent	true	0.01
present	false	0.00
present	true	1.00

Smoking	
nonsmoker	0.50
smoker	0.50

TbOrCa	XRay	ABOUT TO BE USED
false	abnormal	0.05
false	normal	0.95
true	abnormal	0.98
true	normal	0.02

Figure 4.10: The 'Asia' distribution after eliminating: `VisitAsia`, `Tuberculosis`. The new factor produced by eliminating `Tuberculosis` is labelled `NEW`. About to eliminate `XRay` by summing it out of the factor marked `ABOUT TO BE USED`.

Bronchitis	Dyspnea	TbOrCa	ABOUT TO BE USED
absent	absent	false	0.90
absent	absent	true	0.30
absent	present	false	0.10
absent	present	true	0.70
present	absent	false	0.20
present	absent	true	0.10
present	present	false	0.80
present	present	true	0.90

Bronchitis	Smoking	
absent	nonsmoker	0.70
absent	smoker	0.40
present	nonsmoker	0.30
present	smoker	0.60

Cancer	Smoking	
absent	nonsmoker	0.99
absent	smoker	0.90
present	nonsmoker	0.01
present	smoker	0.10

Cancer	TbOrCa	
absent	false	0.99
absent	true	0.01
present	false	0.00
present	true	1.00

Smoking	
nonsmoker	0.50
smoker	0.50

TbOrCa	NEW
false	1.00
true	1.00

Figure 4.11: The 'Asia' distribution after eliminating: `VisitAsia, Tuberculosis, XRay`. The new factor produced by eliminating `XRay` is labelled `NEW`. About to eliminate `Dyspnea` by summing it out of the factor marked `ABOUT TO BE USED`.

Bronchitis	Smoking	ABOUT TO BE USED
absent	nonsmoker	0.70
absent	smoker	0.40
present	nonsmoker	0.30
present	smoker	0.60

Bronchitis	TbOrCa	NEW, ABOUT TO BE USED
absent	false	1.00
absent	true	1.00
present	false	1.00
present	true	1.00

Cancer	Smoking	
absent	nonsmoker	0.99
absent	smoker	0.90
present	nonsmoker	0.01
present	smoker	0.10

Cancer	TbOrCa	
absent	false	0.99
absent	true	0.01
present	false	0.00
present	true	1.00

Smoking	
nonsmoker	0.50
smoker	0.50

TbOrCa	
false	1.00
true	1.00

Figure 4.12: The 'Asia' distribution after eliminating: `VisitAsia, Tuberculosis, XRay, Dyspnea`. The new factor produced by eliminating `Dyspnea` is labelled `NEW`. About to eliminate `Bronchitis` by multiplying the factors marked `ABOUT TO BE USED` and then summing out `Bronchitis`.

```
Cancer  | Smoking   |           ABOUT TO BE USED
------- | --------- | ----
absent  | nonsmoker |   0.99
absent  | smoker    |   0.90
present | nonsmoker |   0.01
present | smoker    |   0.10


Cancer  | TbOrCa |
------- | ------ | ----
absent  | false  |   0.99
absent  | true   |   0.01
present | false  |   0.00
present | true   |   1.00


Smoking   |                     ABOUT TO BE USED
--------- | ----
nonsmoker |   0.50
smoker    |   0.50


Smoking   | TbOrCa |             NEW, ABOUT TO BE USED
--------- | ------ | ----
nonsmoker | false  |   1.00
nonsmoker | true   |   1.00
smoker    | false  |   1.00
smoker    | true   |   1.00


TbOrCa |
------ | ----
false  |   1.00
true   |   1.00
```

Figure 4.13: The 'Asia' distribution after eliminating: `VisitAsia, Tuberculosis, XRay, Dyspnea, Bronchitis`. The new factor produced by eliminating `Bronchitis` is labelled `NEW`. About to eliminate `Smoking` by multiplying the factors marked `ABOUT TO BE USED` and then summing out `Smoking`.

Cancer	TbOrCa	MODIFIED, ABOUT TO BE USED
absent	false	0.94
absent	true	0.01
present	false	0.00
present	true	0.05

TbOrCa	ABOUT TO BE USED
false	1.00
true	1.00

Figure 4.14: The 'Asia' distribution after eliminating: `VisitAsia`, `Tuberculosis`, `XRay`, `Dyspnea`, `Bronchitis`, `Smoking`. The new factor produced by eliminating `Smoking` had variables {`Cancer`, `TbOrCa`}. Since there already existed a factor with exactly these variables (see Fig 4.13) to maintain simplicity the new factor was multiplied into the existing factor producing the factor labelled `MODIFIED`. About to eliminate `TbOrCa` by multiplying the factors marked `ABOUT TO BE USED` and then summing out `TbOrCa`.

Cancer	
absent	0.94
present	0.05

Figure 4.15: The 'Asia' distribution after eliminating: `VisitAsia`, `Tuberculosis`, `XRay`, `Dyspnea`, `Bronchitis`, `Smoking`, `TbOrCa`

`XRay, Bronchitis, Smoking, Dyspnea` is used to compute $P($`Cancer`$)$. Starting out by eliminating `TbOrCa` turns out to be a big mistake since it creates the enormous factor at the top of Fig 4.16. Running `gPy`Example 20 is a good way of understanding the computational difference between the two elimination orderings considered here.

Bronchitis	Cancer	Dyspnea	Tuberculosis	XRay	
absent	absent	absent	absent	abnormal	0.05
absent	absent	absent	absent	normal	0.85
absent	absent	absent	present	abnormal	0.29
absent	absent	absent	present	normal	0.01
absent	absent	present	absent	abnormal	0.01
absent	absent	present	absent	normal	0.10
absent	absent	present	present	abnormal	0.69
absent	absent	present	present	normal	0.01
absent	present	absent	absent	abnormal	0.29
absent	present	absent	absent	normal	0.01
absent	present	absent	present	abnormal	0.29
absent	present	absent	present	normal	0.01
absent	present	present	absent	abnormal	0.69
absent	present	present	absent	normal	0.01
absent	present	present	present	abnormal	0.69
absent	present	present	present	normal	0.01
present	absent	absent	absent	abnormal	0.01
present	absent	absent	absent	normal	0.19
present	absent	absent	present	abnormal	0.10
present	absent	absent	present	normal	0.00
present	absent	present	absent	abnormal	0.04
present	absent	present	absent	normal	0.76
present	absent	present	present	abnormal	0.88
present	absent	present	present	normal	0.02
present	present	absent	absent	abnormal	0.10
present	present	absent	absent	normal	0.00
present	present	absent	present	abnormal	0.10
present	present	absent	present	normal	0.00
present	present	present	absent	abnormal	0.88
present	present	present	absent	normal	0.02
present	present	present	present	abnormal	0.88
present	present	present	present	normal	0.02

Bronchitis	Smoking	
absent	nonsmoker	0.70
absent	smoker	0.40
present	nonsmoker	0.30
present	smoker	0.60

Cancer	Smoking	
absent	nonsmoker	0.99
absent	smoker	0.90
present	nonsmoker	0.01
present	smoker	0.10

Smoking	
nonsmoker	0.50
smoker	0.50

Tuberculosis	VisitAsia	
absent	no_visit	0.99
absent	visit	0.95
present	no_visit	0.01
present	visit	0.05

VisitAsia	
no_visit	0.99
visit	0.01

Figure 4.16: The 'Asia' distribution after eliminating: `TbOrCa`

Bronchitis	Cancer	Dyspnea	Tuberculosis	XRay	
absent	absent	absent	absent	abnormal	0.05
absent	absent	absent	absent	normal	0.85
absent	absent	absent	present	abnormal	0.29
absent	absent	absent	present	normal	0.01
absent	absent	present	absent	abnormal	0.01
absent	absent	present	absent	normal	0.10
absent	absent	present	present	abnormal	0.69
absent	absent	present	present	normal	0.01
absent	present	absent	absent	abnormal	0.29
absent	present	absent	absent	normal	0.01
absent	present	absent	present	abnormal	0.29
absent	present	absent	present	normal	0.01
absent	present	present	absent	abnormal	0.69
absent	present	present	absent	normal	0.01
absent	present	present	present	abnormal	0.69
absent	present	present	present	normal	0.01
present	absent	absent	absent	abnormal	0.01
present	absent	absent	absent	normal	0.19
present	absent	absent	present	abnormal	0.10
present	absent	absent	present	normal	0.00
present	absent	present	absent	abnormal	0.04
present	absent	present	absent	normal	0.76
present	absent	present	present	abnormal	0.88
present	absent	present	present	normal	0.02
present	present	absent	absent	abnormal	0.10
present	present	absent	absent	normal	0.00
present	present	absent	present	abnormal	0.10
present	present	absent	present	normal	0.00
present	present	present	absent	abnormal	0.88
present	present	present	absent	normal	0.02
present	present	present	present	abnormal	0.88
present	present	present	present	normal	0.02

Bronchitis	Smoking	
absent	nonsmoker	0.70
absent	smoker	0.40
present	nonsmoker	0.30
present	smoker	0.60

Cancer	Smoking	
absent	nonsmoker	0.99
absent	smoker	0.90
present	nonsmoker	0.01
present	smoker	0.10

Smoking	
nonsmoker	0.50
smoker	0.50

Tuberculosis	
absent	0.99
present	0.01

Figure 4.17: The 'Asia' distribution after eliminating: `TbOrCa, VisitAsia`

Bronchitis	Cancer	Dyspnea	XRay	
absent	absent	absent	abnormal	0.05
absent	absent	absent	normal	0.85
absent	absent	present	abnormal	0.01
absent	absent	present	normal	0.09
absent	present	absent	abnormal	0.29
absent	present	absent	normal	0.01
absent	present	present	abnormal	0.69
absent	present	present	normal	0.01
present	absent	absent	abnormal	0.01
present	absent	absent	normal	0.19
present	absent	present	abnormal	0.05
present	absent	present	normal	0.75
present	present	absent	abnormal	0.10
present	present	absent	normal	0.00
present	present	present	abnormal	0.88
present	present	present	normal	0.02

Bronchitis	Smoking	
absent	nonsmoker	0.70
absent	smoker	0.40
present	nonsmoker	0.30
present	smoker	0.60

Cancer	Smoking	
absent	nonsmoker	0.99
absent	smoker	0.90
present	nonsmoker	0.01
present	smoker	0.10

Smoking	
nonsmoker	0.50
smoker	0.50

Figure 4.18: The 'Asia' distribution after eliminating: `TbOrCa, VisitAsia, Tuberculosis`

```
Bronchitis | Cancer  | Dyspnea |
---------- | ------- | ------- | ----
absent     | absent  | absent  |   0.89
absent     | absent  | present |   0.11
absent     | present | absent  |   0.30
absent     | present | present |   0.70
present    | absent  | absent  |   0.20
present    | absent  | present |   0.80
present    | present | absent  |   0.10
present    | present | present |   0.90


Bronchitis | Smoking   |
---------- | --------- | ----
absent     | nonsmoker |   0.70
absent     | smoker    |   0.40
present    | nonsmoker |   0.30
present    | smoker    |   0.60


Cancer  | Smoking   |
------- | --------- | ----
absent  | nonsmoker |   0.99
absent  | smoker    |   0.90
present | nonsmoker |   0.01
present | smoker    |   0.10


Smoking   |
--------- | ----
nonsmoker |   0.50
smoker    |   0.50
```

Figure 4.19: The 'Asia' distribution after eliminating: `TbOrCa`, `VisitAsia`, `Tuberculosis`, `XRay`

gPy **Example 20 (Variable elimination demo)**
To see the steps of the naïve variable elimination algorithm for the 'Asia' distribution using the elimination orderings `VisitAsia`, `Tuberculosis`, `XRay`, `Dyspnea`, `Bronchitis`, `Smoking`, `TbOrCa` *and* `TbOrCa`, `VisitAsia`, `Tuberculosis`, `XRay`, `Bronchitis`, `Smoking`, `Dyspnea` *do:*

```
>>> from gPy.Examples import ve_demo
>>> ve_demo()
```

Factors about to produce a product factor are in blue. Those produced by summing a variable out of this product factor are in red. (The product factor itself is not shown.) Click on Done *or* Remove message *to collapse*

Cancer	Dyspnea	Smoking	
absent	absent	nonsmoker	0.69
absent	absent	smoker	0.48
absent	present	nonsmoker	0.31
absent	present	smoker	0.52
present	absent	nonsmoker	0.24
present	absent	smoker	0.18
present	present	nonsmoker	0.76
present	present	smoker	0.82

Cancer	Smoking	
absent	nonsmoker	0.99
absent	smoker	0.90
present	nonsmoker	0.01
present	smoker	0.10

Smoking	
nonsmoker	0.50
smoker	0.50

Figure 4.20: The 'Asia' distribution after eliminating: `TbOrCa`, `VisitAsia`, `Tuberculosis`, `XRay`, `Bronchitis`

Cancer	Dyspnea	
absent	absent	0.55
absent	present	0.39
present	absent	0.01
present	present	0.04

Figure 4.21: The 'Asia' distribution after eliminating: `TbOrCa`, `VisitAsia`, `Tuberculosis`, `XRay`, `Bronchitis`, `Smoking`

Cancer	
absent	0.94
present	0.05

Figure 4.22: The 'Asia' distribution after eliminating: `TbOrCa`, `VisitAsia`, `Tuberculosis`, `XRay`, `Bronchitis`, `Smoking`, `Dyspnea`

the display. Clicking on Quit *removes all stages of the relevant run of naïve variable elimination.*

To see why the creation of big factors is best avioded, let us consider the computational complexity of naïve variable elimination. It is enough to consider the complexity of factor multiplication and summing out, since updating the associated hypergraph data structure takes negligible time. For factor multiplication, any time taken to perform broadcasting or something similar will also be ignored. Now consider computing $f = f_1 f_2 \dots f_n$ (f is called `prod_factor` in Algorithm 1). There are $|\mathcal{I}_f|$ values to compute for f (where $\mathcal{I}_f$ is f's table). Each one of these values is computed by multiplying n numbers from $f_1, f_2, \dots f_n$. It follows that computing f has complexity $O(n|\mathcal{I}_f|)$. $|\mathcal{I}_f|$, the size of f's table, grows exponentially with the number of variables in f, so it follows that factors with many variables are very expensive. For example the factor produced by multiplying all 'Asia' factors containing `TbOrCa` has table size 2^6 since it has 6 variables (`Bronchitis, Cancer, Dyspnea, Tuberculosis, XRay` and `TbOrCa`) all of which are binary. To produce this factor 3×2^6 (scalar) multplications are required since 3 factors are involved. Big factors are naturally also bad news when it comes to marginalisation since summing a single variable out of a factor f is $O(|\mathcal{I}_f|)$.

Evidently, it would be better to have a variable elimination algorithm which could work out an optimal ordering. The bad news is that finding an optimal elimination ordering is an NP-complete problem. The good news is that there are various heuristic approaches which usually find a good (if not optimal) ordering quickly. To address this important issue it is necessary first to study hypergraphs and also graphs in more detail: this is the subject of the next chapter.

The use of a fixed user-supplied ordering is one element of the naïvety of naïve variable elimination but there are further problems. Suppose a joint distribution $P(X_1, X_2, \dots X_{100})$ is such that all variables are mutually independent and so it has a factored representation $P(X_1, X_2, \dots X_{100}) = Z^{-1} f_1(X_1) f_2(X_2) \dots f_{100}(X_{100})$. Consider computing the marginal distribution of, say, X_1 using naïve variable elimination using any elimination ordering of the other variables. The algorithm will dutifully sum out each of the 99 variables even though it is obvious that $P(X_1) = f_1(X_1)/Z(f_1(X_1))$: due to independence it is not necessary to even look at the other variables. An example of this phenomenon using only 3 variables is given in Fig 4.23. The marginal distribution $P(X_1)$ is evidently $P(X_1 = 0) = 0.1, P(X_1 = 1) = 0.9$ but naïve variable elimination does not know to compute this directly without any summing out. (Note, from Fig 4.23, that summing out *all* the variables in a factor produces an *zero-dimensional factor*: a function mapping the empty set $\emptyset$ to a single value. The hyperedge associated with a zero-dimensional factor is $\emptyset$.) Clearly a non-naïve variable elimination algorithm should take advantage of independence. We will see later that intelligent variable elimination can take advantage of a much more common

```
X1 |               X2 |               X3 |
-- | ----         -- | ----         -- | ----
0  |      1    *    0  |      4    *    0  |      7
1  |      9         1  |      6         1  |      3

Summing out X3 gives ...

X1 |               X2 |
-- | ----         -- | ----         ----
0  |      1    *    0  |      4    *        10
1  |      9         1  |      6

Summing out X2 gives ...

X1 |
-- | ----           ----
0  |      1    *        100
1  |      9
```

Figure 4.23: Naive variable elimination failing to take advantage of independence. The hypergraphs for the 3 factored representations are: $\{\{X_1\},\{X_2\},\{X_3\}\}$, $\{\{X_1\},\{X_2\},\emptyset\}$, and $\{\{X_1\},\emptyset\}$, respectively. When summing out X_2 the zero-dimensional factor produced is multiplied into the existing zero-dimensional factor to maintain simplicity.

structural form: *conditional independence*.

gPy **Example 21 (Naïve variable elimination being naïve)**
The distribution from Fig 4.23 is available as follows:

```
>>> from gPy.Examples import indep_hm
>>> print indep_hm
```

To eliminate `X3` and then `X2` (and print out the results) do:

```
>>> indep_hm.eliminate_variable('X3')
>>> print indep_hm
>>> indep_hm.eliminate_variable('X2')
>>> print indep_hm
```

There is a third problem with naïve variable elimination which arises if the marginal distributions of several variables are required. This is a common requirement: Bayesian network software usually updates the marginal

distributions of *all* variables as evidence is added or retracted. Consider using naïve variable elimination to first compute $P(\texttt{Cancer})$ with elimination ordering `VisitAsia, Tuberculosis, XRay, Dyspnea, Bronchitis, Smoking, TbOrCa` and then computing $P(\texttt{TbOrCa})$ with elimination ordering `VisitAsia, Tuberculosis, XRay, Dyspnea, Bronchitis, Smoking, Cancer`. The first computation has already been detailed in Figs 4.8–4.15. It is obvious that the second computation would be exactly the same except for the last step where `Cancer` would be summed out of the last remaining factor instead of `TbOrCa`. So doing these two runs of naïve variable elimination in succession would result in redoing a lot of computation. In Chapter 6 a more sophisticated approach to computing marginal distributions will be given where such wastefulness is avoided.

4.9 Conditioning factored representations

Conditioning a factored representation is a trivial operation, it suffices to just condition each factor appropriately. The top two factored representations of Fig 4.24 shows how a factored distribution for $P(A, B, C, D|A = 0)$ is produced from one for $P(A, B, C, D)$. Fig 4.24 also shows how easy it is to eliminate (sum out) an instantiated variable: producing a factored representation of the marginal distribution $P(B, C, D|A = 0)$ from one for the $P(A, B, C, D|A = 0)$. To eliminate an instantiated variable it suffices to drop it from any factor containing it.

Note that conditioning reduces the Z normalising factor unless, unusually, the conditioned-on event has probability one. Usually, neither the original Z value nor the one which obtains after conditioning is actually computed. In such cases conditioning amounts to nothing more than deleting cells from the tables of factors.

gPy **Example 22 (Conditioning a distribution)**
To condition the distribution in Fig 4.24 do:

```
>>> from gPy.Examples import *
>>> print nondecomp
>>> print nondecomp.z()
>>> tmp = nondecomp.copy(copy_domain=True)
>>> print tmp.condition({'A':'0'})
>>> print tmp.z()
```

Recall that conditioning simply removes values for a variable (using the shortcut advocated in Section 3.2.2). Making `tmp` *a 'deep' copy of* `nondecomp` *by using* `copy_domain=True` *leaves the variables of* `nondecomp` *unscathed when conditioning is performed on* `tmp`*. (If you have no reason to keep* `nondecomp` *around for further work, there is no need to make a copy.)*

```
A | B |          A | D |          B | C |          C | D |
- | - | ----     - | - | ----     - | - | ----     - | - | ----
0 | 0 |   0.10   0 | 0 |   0.90   0 | 0 |   0.40   0 | 0 |   0.50
0 | 1 |   0.20 * 0 | 1 |   0.20 * 0 | 1 |   0.70 * 0 | 1 |   0.20
1 | 0 |   0.30   1 | 0 |   0.70   1 | 0 |   0.30   1 | 0 |   0.40
1 | 1 |   0.20   1 | 1 |   0.10   1 | 1 |   0.10   1 | 1 |   0.10

Z = 0.2165

A | B |          A | D |          B | C |          C | D |
- | - | ----     - | - | ----     - | - | ----     - | - | ----
0 | 0 |   0.10   0 | 0 |   0.90   0 | 0 |   0.40   0 | 0 |   0.50
0 | 1 |   0.20 * 0 | 1 |   0.20 * 0 | 1 |   0.70 * 0 | 1 |   0.20
                                    1 | 0 |   0.30   1 | 0 |   0.40
                                    1 | 1 |   0.10   1 | 1 |   0.10

Z = 0.0832

    B |              D |          B | C |          C | D |
    - | ----         - | ----     - | - | ----     - | - | ----
    0 |   0.10       0 |   0.90   0 | 0 |   0.40   0 | 0 |   0.50
    1 |   0.20 *     1 |   0.20 * 0 | 1 |   0.70 * 0 | 1 |   0.20
                                    1 | 0 |   0.30   1 | 0 |   0.40
                                    1 | 1 |   0.10   1 | 1 |   0.10

Z = 0.0832
```

Figure 4.24: The top four factors define a distribution $P(A, B, C, D)$, the middle four define $P(A, B, C, D|A = 0)$ and the bottom four define $P(B, C, D|A = 0)$. Note that the computational trick of deleting zero probabilities is being used here.

To compute any marginal conditional distribution three steps are enough. Firstly the original distribution is conditioned as described above, secondly any unwanted variables are summed out using, for example, the naïve variable elimination algorithm, and finally normalisation is carried out. If the marginal conditional distribution is only required up to normalisation then this last step can be skipped. Note that normalisation is *delayed* until after any marginalisation is done . This is the case whether or not there has been any earlier conditioning. This is for computational reasons: the fewer the number of variables in a factored representation, the quicker it will be to compute the normalising constant Z.

There is another important computational saving that can be effected as a result of conditioning: eliminating an instantiated variable can be done substantially more quickly than eliminating any other variable. The bottom two rows of Fig 4.24 show the basic idea: to eliminate an instantiated variable it is enough to delete the variable from any factor in which it appears. There is no need to perform factor multiplication or factor marginalisation as in the general case. Algorithm 2 shows how eliminating a variable is altered to account for instantiated variables. If a variable is instantiated then any factor including that variable is removed and replaced by a factor without the variable.

Algorithm 2 Naïve variable elimination accounting for instantiated variables

```
def eliminate_variable(self,variable):
    hyperedges = self._hypergraph.star(variable)
    if variable in self._instd:
        vset = set([variable])
        for hyperedge in hyperedges:
            nf = SubDomain.marginalise_away(
                self.factor(hyperedge),vset)
            self.remove(hyperedge)
            self *= nf
    else:
        prod_factor = 1
        for hyperedge in hyperedges:
            prod_factor *= self.factor(hyperedge)
            self.remove(hyperedge)
        message = prod_factor.sumout([variable])
        self *= message
```

gPy Example 23 shows how the marginal conditional distribution $P(\texttt{Cancer}|\texttt{Bronchitis} = \texttt{absent}, \texttt{XRay} = \texttt{normal})$ is computed by conditioning and marginalisation. Firstly, the conditional distribution

$$P(\texttt{VisitAsia}, \ldots \texttt{Smoking}, \texttt{TbOrCa}, \texttt{Cancer}|\texttt{Bronchitis} = \texttt{absent}, \texttt{XRay} = \texttt{normal})$$

(displayed in Fig 4.25) is generated. Secondly, naïve variable elimination is

used to produce a factor representing $P(\texttt{Cancer}|\texttt{Bronchitis} = \texttt{absent}, \texttt{XRay} = \texttt{normal})$ up to normalisation. This can then be normalised if required.

gPy **Example 23 (Variable elimination with conditioning)** *This example is identical to gPyExample 20 except that the initial distribution is the conditional distribution* $P(\texttt{VisitAsia}, \texttt{Tuberculosis}, \texttt{XRay}, \texttt{Dyspnea}, \texttt{Bronchitis}, \texttt{Smoking}, \texttt{TbOrCa}, \texttt{Cancer}|\texttt{Bronchitis} = \texttt{absent}, \texttt{XRay} = \texttt{normal})$*, which is displayed in Fig 4.25. Just do:*

```
>>> from gPy.Examples import ve_demo2
>>> ve_demo2()
```

Note that when an instantiated variable is eliminated no new factor (no new 'message') is created.

4.10 Towards join forest calibration

After going through the necessary mathematical material in Chapter 5 an important algorithm: *join forest calibration* will be described in Chapter 6. This algorithm avoids many of the deficiencies of variable elimination (whether 'naïve' as described here or more sophisticated as described in Chapter 5). Despite its superiority to variable elimination join forest calibration has very close connections to variable elimination and so we close this chapter with an example of variable elimination which anticipates the join forest calibration algorithm.

Consider the factored representation in Fig 4.26 which has the associated (reduced) hypergraph:

$$\begin{array}{l}\{\{\texttt{Bronchitis}, \texttt{Dyspnea}, \texttt{TbOrCa}\},\\ \{\texttt{Bronchitis}, \texttt{Smoking}, \texttt{TbOrCa}\},\\ \{\texttt{Cancer}, \texttt{Smoking}, \texttt{TbOrCa}\},\\ \{\texttt{Cancer}, \texttt{TbOrCa}, \texttt{Tuberculosis}\},\\ \{\texttt{TbOrCa}, \texttt{XRay}\},\\ \{\texttt{Tuberculosis}, \texttt{VisitAsia}\}\}\end{array} \tag{4.6}$$

Now suppose we wanted to compute $P(\texttt{Cancer})$ from this factored representation. Note that the variables `VisitAsia`, `XRay` and `Dyspnea` all appear in only one factor/hyperedge. It thus makes sense to sum these variables out first. If we do this then the three factors created (`message` factors in the language of Algorithm 1) have variables $\{\texttt{Tuberculosis}\}$, $\{\texttt{TbOrCa}\}$ and $\{\texttt{Bronchitis}, \texttt{TbOrCa}\}$, respectively. All three of these are redundant factors. A redundant factor f can be 'absorbed' into any factor f' whose variables are a superset of the f's variables without altering the distribution represented: f' is replaced by $f \times f'$ and f is removed. If we absorb redundant factors to maintain reducedness, a

```
Bronchitis | Dyspnea | TbOrCa |
---------- | ------- | ------ | ----
absent     | absent  | false  |   0.90
absent     | absent  | true   |   0.30
absent     | present | false  |   0.10
absent     | present | true   |   0.70


Bronchitis | Smoking   |
---------- | --------- | ----
absent     | nonsmoker |   0.70
absent     | smoker    |   0.40


Cancer  | Smoking   |
------- | --------- | ----
absent  | nonsmoker |   0.99
absent  | smoker    |   0.90
present | nonsmoker |   0.01
present | smoker    |   0.10


Cancer  | TbOrCa | Tuberculosis |
------- | ------ | ------------ | ----
absent  | false  | absent       |   1.00
absent  | false  | present      |   0.00
absent  | true   | absent       |   0.00
absent  | true   | present      |   1.00
present | false  | absent       |   0.00
present | false  | present      |   0.00
present | true   | absent       |   1.00
present | true   | present      |   1.00


Smoking   |
--------- | ----
nonsmoker |   0.50
smoker    |   0.50


TbOrCa | XRay   |
------ | ------ | ----
false  | normal |   0.95
true   | normal |   0.02


Tuberculosis | VisitAsia |
------------ | --------- | ----
absent       | no_visit  |   0.99
absent       | visit     |   0.95
present      | no_visit  |   0.01
present      | visit     |   0.05


VisitAsia |
--------- | ----
no_visit  |   0.99
visit     |   0.01
```

Figure 4.25: A factored representation of the distribution $P(\texttt{VisitAsia}, \texttt{Tuberculosis}, \texttt{XRay}, \texttt{Dyspnea}, \texttt{Bronchitis}, \texttt{Smoking}, \texttt{TbOrCa}, \texttt{Cancer} | \texttt{Bronchitis} = \texttt{absent}, \texttt{XRay} = \texttt{normal})$

Bronchitis	Dyspnea	TbOrCa	
absent	absent	false	0.90
absent	absent	true	0.30
absent	present	false	0.10
absent	present	true	0.70
present	absent	false	0.20
present	absent	true	0.10
present	present	false	0.80
present	present	true	0.90

Bronchitis	Smoking	TbOrCa	
absent	nonsmoker	false	0.70
absent	nonsmoker	true	0.70
absent	smoker	false	0.40
absent	smoker	true	0.40
present	nonsmoker	false	0.30
present	nonsmoker	true	0.30
present	smoker	false	0.60
present	smoker	true	0.60

Cancer	Smoking	TbOrCa	
absent	nonsmoker	false	0.49
absent	nonsmoker	true	0.49
absent	smoker	false	0.45
absent	smoker	true	0.45
present	nonsmoker	false	0.01
present	nonsmoker	true	0.01
present	smoker	false	0.05
present	smoker	true	0.05

Cancer	TbOrCa	Tuberculosis	
absent	false	absent	1.00
absent	false	present	0.00
absent	true	absent	0.00
absent	true	present	1.00
present	false	absent	0.00
present	false	present	0.00
present	true	absent	1.00
present	true	present	1.00

TbOrCa	XRay	
false	abnormal	0.05
false	normal	0.95
true	abnormal	0.98
true	normal	0.02

Tuberculosis	VisitAsia	
absent	no_visit	0.98
absent	visit	0.01
present	no_visit	0.01
present	visit	0.00

Figure 4.26: Yet another factored representation of the 'Asia' distribution.

factored representation with the following hypergraph is created:

$$\{\{\texttt{Bronchitis}, \texttt{Smoking}, \texttt{TbOrCa}\},\\ \{\texttt{Cancer}, \texttt{Smoking}, \texttt{TbOrCa}\},\\ \{\texttt{Cancer}, \texttt{TbOrCa}, \texttt{Tuberculosis}\}\}$$

Note that now `Bronchitis` and `Tuberculosis` appear in only one factor. Summing these two out and maintaining reducedness gives a factored representation with only one factor. The associated hypergraph is:

$$\{\{\texttt{Cancer}, \texttt{Smoking}, \texttt{TbOrCa}\}\} \tag{4.7}$$

$P(\texttt{Cancer})$ is then obtained by summing out `Smoking` and `TbOrCa` from this final factor.

So, in this case, an elimination ordering exists which allows very efficient variable elimination. Each time a variable in this ordering is summed out it is only in one factor and so all that needs to be done is to sum it out of this factor—no factor multiplication is required at this point. Factor multiplication only happens when redundant factors get 'mopped up' by being multiplied into existing factors: this is the minimal level of factor multiplication possible in variable elimination. Also, note that finding this good elimination ordering is computationally undemanding: it suffices to find a variable contained in only one factor and choose it as the next one for elimination. It would be easy to alter Algorithm 1 to use this dynamic method of constructing an elimination ordering.

By inspection of the final hypergraph (4.7) it is evident that computing $P(\texttt{Smoking})$ and $P(\texttt{TbOrCa})$ is just as easy as computing $P(\texttt{Cancer})$. In fact, *no matter which variable* it is possible to find an elimination ordering to compute that variable's marginal where all other variables are contained in only one factor at the time of their elimination. Clearly, there is something 'nice' about the structure of the hypergraph (4.6). It is, in fact, our first example of a *decomposable hypergraph*. Decomposable hypergraphs are the key data structure for the join forest calibration algorithm which will be the subject of Chapter 6.

Chapter 5

Graphs, hypergraphs and conditional independence

As was frequently mentioned in Chapter 4, further theory on graphs and hypergraphs is required (i) to analyse the crucial notion of conditional independence and (ii) to design more sophisticated algorithms. This chapter is devoted to the first of these goals, the second is the topic of Chapter 6.

5.1 Conditional independence

As noted in Section 4.2 joint probability distributions defined by independent random variables have an appealingly simple structure, and consequently computing marginal distributions is trivial. The big problem with such distributions is that very few phenomena in the real world are modelled by such distributions: random variables usually do depend on each other.

This can be illustrated by a simple example. There was a time when I was living in Crowthorne, Berkshire (my home town) and working in Oxford. I used to take the train every day. Whether I got to work on time was influenced by whether I managed to catch a particular train from Crowthorne station: if this train was missed there was still some chance of making it in time but it was reduced. Conversely, even if I got the train there might be a delay and I might still get to work late. Whether I got this train had a lot to do with whether I left the house in time. Leaving on time increased my chances of reaching the station to catch the necessary train but did not guarantee it. If I left late I might still make it to the station in time (by running for example). This situation can be modelled by 3 binary variables: `LeaveOnTime, CatchTrain` and `WorkOnTime`, and it is intuitive to say that `CatchTrain` depends on `LeaveOnTime` and `WorkOnTime` depends on `CatchTrain`. Suppose we were to model this situation using a joint probability distribution over the variables `LeaveOnTime, CatchTrain` and `WorkOnTime`. A distribution where the 3 variables are independent cannot be right, but there is some independence structure here to exploit:

CatchTrain	LeaveOnTime			CatchTrain	WorkOnTime	
n	n	0.32		n	n	0.70
n	y	0.12	*	n	y	0.30
y	n	0.08		y	n	0.10
y	y	0.48		y	y	0.90

Figure 5.1: Joint probability distribution where LeaveOnTime $\perp$ WorkOnTime|CatchTrain.

conditional independence. Although WorkOnTime depends on LeaveOnTime it only does so via CatchTrain. Once I have either caught or failed to catch the train, then whether I left on time is of no relevance. We say WorkOnTime is independent of LeaveOnTime *conditional on CatchTrain*. Now it is time to formalise this notion.

Definition 23 Two events $A, B \subset \Omega$ are *independent conditional on a third event* C in a probability distribution P iff $P(C) > 0$ and

$$P(A \cap B|C) = P(A|C)P(B|C) \tag{5.1}$$

The statement that "A and B are independent conditional on C in distribution P" is abbreviated to $A \perp B|C\,[P]$. If the identity of the probability distribution is obvious this can be further abbreviated to $A \perp B|C$. □

Definition 24 Two random variables X and Y are *independent conditional on a third random variable* Z in a joint distribution P iff $\forall x \in \mathcal{I}_X, y \in \mathcal{I}_Y, z \in \mathcal{I}_Z$:

$$P(X = x, Y = y|Z = z) = P(X = x|Z = z)P(Y = y|Z = z) \tag{5.2}$$

where the equation holds whenever $P(Z = z) > 0$. Since $P(X, Y|Z)$, $P(X|Z)$, $P(Y|Z)$ denote functions (factors) with domains $\mathcal{I}_X \times \mathcal{I}_Y \times \mathcal{I}_Z$, $\mathcal{I}_X \times \mathcal{I}_Z$, and $\mathcal{I}_Y \times \mathcal{I}_Z$, respectively, this condition can be abbreviated to:

$$P(X, Y|Z) = P(X|Z)P(Y|Z) \tag{5.3}$$

If conditional indpendence obtains between X, Y and Z in this way then write $X \perp Y|Z\,[P]$ or just $X \perp Y|Z$ if P is obvious. □

Fig 5.1 shows (a factored representation of) a joint probability distribution where WorkOnTime is independent of LeaveOnTime conditional on CatchTrain, i.e. LeaveOnTime $\perp$ WorkOnTime|CatchTrain. (The numbers are just made up.) gPy Example 24 proves that although WorkOnTime and LeaveOnTime are dependent, they are independent conditional on CatchTrain.

gPy Example 24 (Conditional independence and train catching) *Firstly construct and print out the joint distribution from Fig 5.1 as follows:*

```
>>> from gPy.Examples import *
>>> fr = FR([train['LeaveOnTime']*train['CatchTrain'],train['WorkOnTime']*1])
>>> print fr
```

Firstly, use variable elimination to compute $P($`LeaveOnTime`$)$*:*

```
>>> l = fr.copy()
>>> l.variable_elimination(['CatchTrain','WorkOnTime'])
>>> print l

LeaveOnTime |
----------- | ----
n           |   0.40
y           |   0.60
```

and then do the same for $P($`WorkOnTime`$)$*:*

```
>>> w = fr.copy()
>>> w.variable_elimination(['CatchTrain','LeaveOnTime'])
>>> print w

WorkOnTime |
---------- | ----
n          |   0.36
y          |   0.64
```

Now, compute the marginal distribution $P($`LeaveOnTime`$,$ `WorkOnTime`$)$*:*

```
>>> lw = fr.copy()
>>> lw.variable_elimination(['CatchTrain'])
>>> print lw

LeaveOnTime | WorkOnTime |
----------- | ---------- | ----
n           | n          |   0.23
n           | y          |   0.17
y           | n          |   0.13
y           | y          |   0.47
```

It is now easy to show that `LeaveOnTime` *and* `WorkOnTime` *are not independent. If they were then we would have that:* $P($`LeaveOnTime`$,$ `WorkOnTime`$) = P($`LeaveOnTime`$)P($`WorkOnTime`$)$*. The LHS is* `lw` *as computed just above. To get the RHS do:*

```
>>> print l[['LeaveOnTime']] * w[['WorkOnTime']]

LeaveOnTime | WorkOnTime |
----------- | ---------- | ----
n           | n          |   0.15
n           | y          |   0.25
y           | n          |   0.22
y           | y          |   0.38
```

Sure enough these two factors are different so independence does not hold.

As for conditional independence we will show that $P(\texttt{LeaveOnTime}, \texttt{WorkOnTime}|\texttt{CatchTrain} = \texttt{n}) = P(\texttt{LeaveOnTime}|\texttt{CatchTrain} = \texttt{n})P(\texttt{WorkOnTime}|\texttt{CatchTrain} = \texttt{n})$. *The proof for* $\texttt{CatchTrain} = \texttt{y}$ *is left as an exercise.*

Step 1 is to construct the conditional joint distribution:

```
>>> cond = fr.copy(copy_domain=True)
>>> print cond.condition({'CatchTrain':'n'})

CatchTrain | LeaveOnTime |
---------- | ---------- | ----
n          | n          |   0.32
n          | y          |   0.12


CatchTrain | WorkOnTime |
---------- | ---------- | ----
n          | n          |   0.70
n          | y          |   0.30
```

Next compute (up to normalisation) $P(\texttt{LeaveOnTime}|\texttt{CatchTrain} = \texttt{n})$, $P(\texttt{WorkOnTime}|\texttt{CatchTrain} = \texttt{n})$ *and* $P(\texttt{LeaveOnTime}, \texttt{WorkOnTime}|\texttt{CatchTrain} = \texttt{n})$. *These will be objects* `l_c`, `w_c` *and* `lw_c`, *respectively.*

```
>>> l_c = cond.copy()
>>> l_c.variable_elimination(['CatchTrain','WorkOnTime'])
>>> print l_c

LeaveOnTime |
----------- | ----
n           |   0.32
y           |   0.12


>>> w_c = cond.copy()
```

```
>>> w_c.variable_elimination(['CatchTrain','LeaveOnTime'])
>>> print w_c

WorkOnTime |
---------- | ----
n          |   0.31
y          |   0.13


>>> lw_c = cond.copy()
>>> lw_c.variable_elimination(['CatchTrain'])
>>> print lw_c

LeaveOnTime | WorkOnTime |
----------- | ---------- | ----
n           | n          |   0.22
n           | y          |   0.10
y           | n          |   0.08
y           | y          |   0.04
```

If you want to see these distributions properly normalised just do:

```
>>> print l_c/l_c.z()
>>> print w_c/w_c.z()
>>> print lw_c/lw_c.z()
```

It is worth considering in some detail what happens when `lw_c.variable_elimination(['CatchTrain'])` *is called. Since* ***CatchTrain*** *is in both the factors of* **cond**, *these two factors must be multiplied. The first step in this multiplication is to broadcast these two factors to get:*

```
>>> for factor in cond:
...  print factor.broadcast(cond.variables())
...

CatchTrain | LeaveOnTime | WorkOnTime |
---------- | ----------- | ---------- | ----
n          | n           | n          |   0.70
n          | n           | y          |   0.30
n          | y           | n          |   0.70
n          | y           | y          |   0.30


CatchTrain | LeaveOnTime | WorkOnTime |
---------- | ----------- | ---------- | ----
```

```
n          | n          | n          |   0.32
n          | n          | y          |   0.32
n          | y          | n          |   0.12
n          | y          | y          |   0.12
```

Next pointwise multiplication is used to multiply the numbers giving:

```
>>> prod = 1
>>> for factor in cond:
...  prod *= factor.broadcast(cond.variables())
...
>>> print prod

CatchTrain | LeaveOnTime | WorkOnTime |
---------- | ----------- | ---------- | ----
n          | n           | n          |   0.22
n          | n           | y          |   0.10
n          | y           | n          |   0.08
n          | y           | y          |   0.04
```

Finally, ***CatchTrain*** *is summed out. Since it is instantiated this amounts to just dropping its column.*

To prove conditional independence do:

```
>>> tmp = l_c[['LeaveOnTime']] * w_c[['WorkOnTime']]
>>> print tmp/tmp.z()

LeaveOnTime | WorkOnTime |
----------- | ---------- | ----
n           | n          |   0.51
n           | y          |   0.22
y           | n          |   0.19
y           | y          |   0.08

>>> print lw_c/lw_c.z()

LeaveOnTime | WorkOnTime |
----------- | ---------- | ----
n           | n          |   0.51
n           | y          |   0.22
y           | n          |   0.19
y           | y          |   0.08
```

So $P(\texttt{CatchTrain}, \texttt{LeaveOnTime}, \texttt{WorkOnTime})$ is such that $\texttt{LeaveOnTime} \perp$

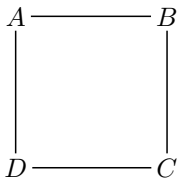

Figure 5.2: An undirected graph where $V = \{A, B, C, D\}$ and $E = \{(A, B), (B, A), (B, C), (C, B), (C, D), (D, C), (A, D), (D, A)\}$.

`WorkOnTime`|`CatchTrain`, but the computations in `gPy` Example 24 fail to provide much insight into *why* this particular probability distribution has this property: is it due to the particular numbers chosen for the distribution? something to do with the variables being binary? or something more structural? In fact, as you may well have guessed by now, it is due to the way the distribution's variables are distributed between the two factors, in other words, it is down to the hypergraph associated with the distribution. However, it is easier to use a *graph* rather than a hypergraph to visualise the conditional independence properties of a factored representation of a distribution, and it is graphs to which we now turn.

5.2 Graphs

Definition 25 packages together the basic definitions for graphs.

Definition 25 A *graph* is a pair $\mathcal{G} = (V, E)$ where V is a finite set of *vertices* and E is a set of *edges* which is a subset of $V \times V$, the set of ordered pairs of vertices. An edge can thus be seen as a directed link between two vertices. Throughout only *simple graphs* will be considered so that loops, which are edges connecting a vertex to itself, are not allowed. An edge $(\alpha, \beta) \in E$ such that (β, α) is also in E is called an *undirected edge*. An edge $(\alpha, \beta) \in E$ such that (β, α) is *not* also in E is called an *directed edge*. A graph only containing undirected edges is called an *undirected graph*. A graph only containing directed edges is called a *directed graph* or *digraph* for short. □

Of course, one of the most attractive features of graphs is their intuitive visual representation. Fig 5.2 provides an example of the standard presentation of an undirected graph and Fig 5.3 shows a directed graph. Note that formally an undirected edge is equivalent to a pair of directed edges but that this is not reflected in the visual representation of undirected graphs.

5.3 Connections between graphs and hypergraphs

5.3.1 2-sections and interaction graphs

Every hypergraph $\mathcal{H}$ has an associated undirected graph: its *2-section* $\mathcal{H}_{[2]}$ formed by connecting vertices with an undirected edge if they are both contained

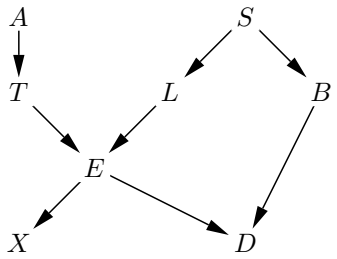

Figure 5.3: A directed graph where $V = \{A, B, D, E, L, S, T, X\}$ and $E = \{(A, T), (T, E), (E, X), (L, E), (S, L), (S, B), (B, D), (E, D)\}$

.

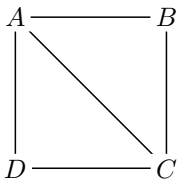

Figure 5.4: The 2-section of the hypergraph $\{\{A, B, C\}, \{A, C, D\}\}$.

in some hyperedge. Definition 26 gives the formal definition.

Definition 26 Let $\mathcal{H}$ be a hypergraph. The *2-section of* $\mathcal{H}$, denoted $\mathcal{H}_{[2]}$ is defined as follows. $\mathcal{H}_{[2]} = (V, E)$ where the vertices V of $\mathcal{H}_{[2]}$ are the same as the vertices of $\mathcal{H}$, and where $(\alpha, \beta) \in E$ iff $\alpha \neq \beta$ and there is some hyperedge $h \in \mathcal{H}$ such that $\alpha \in h$ and $\beta \in h$. (Berge [2] defines a hypergraph's 2-section without the $\alpha \neq \beta$ condition so that the 2-section has loops on each vertex. According to the definition used here no loops exist in the 2-section.) □

So, for example the graph in Fig 5.2 is the 2-section of the hypergraph $\{\{A, B\}, \{B, C\}, \{C, D\}, \{A, D\}\}$ and also of the hypergraph

$$\{\{A, B\}, \{B, C\}, \{C, D\}, \{A, D\}, \{A\}, \{C\}, \emptyset\}$$

. The 2-section of the hypergraph $\{\{A, B, C\}, \{A, C, D\}\}$ is displayed in Fig 5.4 and the 2-section of the hypergraph $\{\{A, B\}, \{B, C\}, \{A, C\}\}$ can be found in Fig 5.5. (Some authors refer to the 2-section of a hypergraph as 'the graph of the hypergraph' and denote it $\mathcal{G}(\mathcal{H})$ [14, 9]. We avoid this since it is liable to cause confusion between the 2-section of a hypergraph and its *representative graph* which is quite different. Representative graphs of hypergraphs will be introduced in Chapter 6.)

Since each factored representation has an associated hypergraph, and each hypergraph has an associated 2-section graph, it follows that each factored representation defines a specific graph: its *interaction graph* as defined formally in Definition 27. `gPy` Example 25 shows how to print and display interaction graphs for some example factored representations.

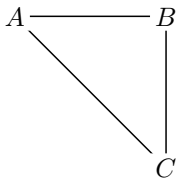

Figure 5.5: The 2-section of the hypergraph $\{\{A, B\}, \{B, C\}, \{A, C\}\}$.

Definition 27 The *interaction graph* for the factored representation $\{f_1, f_2, \ldots, f_n\}$ is $\mathcal{H}(\{f_1, f_2, \ldots, f_n\})_{[2]}$, the 2-section of its hypergraph. □

gPy **Example 25 (Interaction graphs)**
There are various example factored representations (`FR` objects) in `gPy.Examples`. To get the interaction graph for any of them just use the `interaction_graph` *method. There is also a* `hypergraph` *method to generate the hypergraph. Here's how to print the factored representation* `nondecomp` *followed by its hypergraph and then its interaction graph.*

```
>>> from gPy.Examples import *
>>> print nondecomp

A | B |
- | - | ----
0 | 0 |   0.10
0 | 1 |   0.20
1 | 0 |   0.30
1 | 1 |   0.20


A | D |
- | - | ----
0 | 0 |   0.90
0 | 1 |   0.20
1 | 0 |   0.70
1 | 1 |   0.10


B | C |
- | - | ----
0 | 0 |   0.40
0 | 1 |   0.70
1 | 0 |   0.30
1 | 1 |   0.10
```

```
C | D |
- | - | ----
0 | 0 |   0.50
0 | 1 |   0.20
1 | 0 |   0.40
1 | 1 |   0.10

>>> print nondecomp.hypergraph()
{ {C, D}, {B, C}, {A, D}, {A, B} }
>>> print nondecomp.interaction_graph()
Vertices:
['A', 'B', 'C', 'D']
Lines:
A - B
A - D
B - C
C - D
```

You can do the same for the other `FR` *objects. To see what there is, do (don't forget the identations):*

```
>>> for x in dir():
...  if isinstance(eval(x),FR):
...   print x
...
asia
indep_hm
minibn
nondecomp
nondecomp_norm
train
```

To see the graphs in a GUI, first grab the necessary objects and create a window:

```
>>> from Tkinter import *
>>> root = Tk()
```

Then just use the `gui_display` *method to display the graphs. For example:*

```
>>> nondecomp.interaction_graph().gui_display(root)
<gPy.Structures.GraphCanvas instance at 0xb780d7ec>
>>> asia.interaction_graph().gui_display(root)
<gPy.Structures.GraphCanvas instance at 0xb780d9cc>
```

You will need to move the vertices around to get a nice layout. Do this by selecting the vertex with the left mouse button and drag while keeping this

button pressed. A lazier option is to run some canned examples (you will have to kill the windows before proceeding further):

```
>>> igs()
>>> igs2()
```

The interaction graph of a factored representation is crucially important because it is the object which allows us to 'read off' conditional independence properties of the distribution. Basically, if two variables are 'separated' by a third in the interaction graph then conditioning on this third variable makes the first two independent in the distribution. Now this notion of separation in a graph will be formalised.

Definition 28 Let $\mathcal{G} = (V, E)$ be a graph. A path in $\mathcal{G}$ from $\alpha \in V$ to $\beta \in V$ is a sequence $\alpha = \alpha_0, \alpha_1, \ldots, \alpha_n = \beta$ such that $(\alpha_i, \alpha_{i+1}) \in E$ for all $i = 1, \ldots, n$. A subset C is said to be a (α, β)-*separator* if all paths from α to β intersect C. A single vertex γ is said to be an (α, β)-separator if $\{\gamma\}$ is one. Let $A, B \subset V$. C *separates* A from B if C is an (α, β)-separator for all $\alpha \in A, \beta \in B$. C is said to separate α from β if it separates $\{\alpha\}$ from $\{\beta\}$. A single vertex γ is said to separate A from B if $\{\gamma\}$ does. □

So, for example, in the graph of Fig 5.2 there are two paths from A to C: (A, B, C) and (A, D, C). (Note that here A, B, C, D refer to vertices not, as in Definition 28, to sets of vertices. This is unfortunate clash is due to the common practice of using upper-case letters both for random variables and for set of vertices.) It follows that neither B nor D are (A, C)-separators but that $\{B, D\}$ is. In Fig 5.4 there is a edge between A and C so there can be no (A, C)-separator. Note that in undirected graphs there is a path from α to β iff there is a path from β to α, so that (α, β)-separators are also (β, α)-separators.

This path-related symmetry does not obtain in directed graphs. For example in Fig 5.3 there is a path from S to X but not from X to S. The single path from S to X goes via L and E, and so both these vertices are (S, X)-separators. L and E are both on one path from S to D, but since there is another via B neither is a (S, D)-separator. S and D are separated by, for example, $\{B, L\}$.

Separators allow a graph to represent the conditional independence properties of a probability distribution. Our first example of using a graph in this way is provided by the *global Markov property for undirected graphs* as defined in Definition 29.

Definition 29 A joint distribution P over variables V obeys the *global Markov property* relative to an undirected graph $\mathcal{G} = (V, E)$, if for any triple (A, B, S) of disjoint subsets of V such that S separates A from B in $\mathcal{G}$, we have:

$$A \perp B | S \, [P]$$

□

For example any distribution P obeying the global Markov property with respect to the graph in Fig 5.2 has the following conditional independence properties:

$$A \perp C|\{B, D\}$$
$$B \perp D|\{A, C\}$$

So, if a distribution obeys the global Markov property relative to some graph, the graph provides a useful picture of the (conditional independence) structure of the distribution. You will not be surprised to learn that the graph also provides a crucial data structure for designing efficicent algorithms for manipulating the distribution. Now it is necessary to establish *why* the interaction graph represents conditional independence relations. To do this we first define *clique hypergraphs* and the notion of distribution *factorising according to a graph*.

5.3.2 Clique hypergraphs

We have seen that each hypergraph $\mathcal{H}$ has an associated graph $\mathcal{H}_{[2]}$. But it is also the case that each undirected graph has an associated hypergraph defined in terms of its *cliques*. First, we define what a clique is.

Definition 30 Let $\mathcal{G} = (V, E)$ be an undirected graph. A set of vertices $A \subseteq V$ is *complete* if all vertices in A are connected by an edge. A set $C \subseteq V$ is a *clique* if it is maximally complete—it is complete but not properly contained in another complete set. (To understand why the word 'clique' is used, think about graphs where vertices represent people and edges represent friendships.)□

So, for example, the 4 cliques of the graph in Fig 5.2 are $\{A, B\}, \{B, C\}, \{C, D\}$ and $\{A, D\}$ whereas the 2 cliques of the graph in Fig 5.4 are $\{A, B, C\}$ and $\{A, C, D\}$. Note that $\{A, B\}$ is a complete subset of the graph in Fig 5.4 but it is not a clique since it is properly contained in the complete set $\{A, B, C\}$. Since the cliques of a graph $\mathcal{G}$ are sets of vertices it is evident that the set of all cliques constitute a hypergraph. This hypergraph is known as the *clique hypergraph* of the graph and is denoted $\mathcal{C}(\mathcal{G})$.

Definition 31 The *clique hypergraph* of a graph $\mathcal{G}$ is the set of cliques of $\mathcal{G}$. It is denoted $\mathcal{C}(\mathcal{G})$. □

Clique hypergraphs are guaranteed to have certain properties which not all hypergraphs have. Firstly, they are always reduced (and thus also simple). This is because each hyperedge in a clique hypergraph is a clique and no clique can be contained within another.

A clique hypergraph is also a *graphical hypergraph*. To understand this concept consider an arbitrary hypergraph $\mathcal{H}$, and compare it to $\mathcal{C}(\mathcal{H}_{[2]})$ the clique hypergraph of its 2-section. It is not difficult to see that every hyperedge in $\mathcal{H}$

is contained in some hyperedge of $\mathcal{C}(\mathcal{H}_{[2]})$ which makes $\mathcal{C}(\mathcal{H}_{[2]})$ a *cover* of $\mathcal{H}$. Definition 32 provides the formal definition of cover and some related terms.

Definition 32 A *partial hyperedge* of $\mathcal{H}$ is a subset (proper or improper) of a hyperedge of $\mathcal{H}$. A *cover* of $\mathcal{H}$ is a hypergraph $\mathcal{H}'$ with the same vertices as $\mathcal{H}$ and such that every hyperedge of $\mathcal{H}$ is a partial hyperedge of $\mathcal{H}'$. Write $\mathcal{H}_2 \preceq \mathcal{H}_1$ if $\mathcal{H}_1$ is a cover of $\mathcal{H}_2$. □

As previously noted it is the case that $\mathcal{H} \preceq \mathcal{C}(\mathcal{H}_{[2]})$. Now, is it the case that $\mathcal{C}(\mathcal{H}_{[2]}) \preceq \mathcal{H}$? In general, the answer is no. For example, consider the hypergraph $\mathcal{H} = \{\{A, B\}, \{B, C\}, \{A, C\}\}$. $\mathcal{H}_{[2]}$, the 2-section, is the graph displayed in Fig 5.5. This graph has a single clique $\{A, B, C\}$ so $\mathcal{C}(\mathcal{H}_{[2]}) = \{\{A, B, C\}\} \neq \mathcal{H}$. All the hyperedges of $\mathcal{H}$ are contained in $\{A, B, C\}$ so, as expected, $\mathcal{H} \preceq \mathcal{C}(\mathcal{H}_{[2]})$. But $\mathcal{C}(\mathcal{H}_{[2]}) \not\preceq \mathcal{H}$ since $\{A, B, C\}$ is not contained in any hyperedge of $\mathcal{H}$. In contrast, for $\mathcal{H}' = \{\{A, B\}, \{B, C\}, \{B, C\}, \{C, D\}\}$, it is the case that $\mathcal{C}(\mathcal{H}'_{[2]}) \preceq \mathcal{H}'$. $\mathcal{H}'$ is thus an example of a *graphical hypergraph* which is formally defined in Definition 33.

Definition 33 A hypergraph $\mathcal{H}$ is *graphical* if $\mathcal{C}(\mathcal{H}_{[2]}) \preceq \mathcal{H}$. A common synonym for 'graphical hypergraph' is *conformal hypergraph*. □

If a hypergraph is graphical, then its reduction is exactly the clique hypergraph of its 2-section: $\text{red}(\mathcal{H}) = \mathcal{C}(\mathcal{H}_{[2]})$. It is obvious that any clique hypergraph is graphical.

5.3.3 Factorising according to a graph

As promised at the end of Section 5.3.1 the notion of factorising according to a graph is now given.

Definition 34 A joint probability distribution P *factorises according to the graph* $\mathcal{G}$ if there are factors f_c such that

$$P = \prod_{c \in \mathcal{C}(\mathcal{G})} f_c$$

where each $\Delta(f_c) = c$, for each clique $c \in \mathcal{C}(\mathcal{G})$. □

Note that for any distribution P, there exists $\{f_c\}_{c \in \mathcal{C}}$ such that $P = \prod_{c \in \mathcal{C}} f_c$ iff there exists $\{f'_c\}_{c \in \mathcal{C}}$ and a normalising constant Z such that $P = Z^{-1} \prod_{c \in \mathcal{C}} f'_c$. This is why there is no need to include a normalising constant Z in Definition 34.

So a distribution factorises according to a graph if it has a factored representation whose hypergraph is the graph's clique hypergraph. If a distribution P is *defined* by some factored distribution, then it is obvious that P factorises according to that factored representation's interaction graph.

5.3.4 The Hammersley-Clifford theorem

The Hammersley-Clifford theorem establishes the connection between factorising according to a graph and the conditional independence relations which the graph represents. Firstly, one connection between factorising according to a graph and the global Markov property is given.

Theorem 2 *If a distribution P factorises according to an undirected graph $\mathcal{G}$, then P obeys the global Markov property with respect to $\mathcal{G}$.* □

The proof of Theorem 2 can be found in [9, p.35]. The question now is whether the global Markov property implies factorising. To answer this two weaker forms of Markov property are introduced.

Definition 35 A joint distribution P over variables V obeys the *pairwise Markov property* relative to an undirected graph $\mathcal{G} = (V, E)$, if for any pair of non-adjacent vertices (α, β) in V, we have:

$$\alpha \perp \beta | V \setminus \{\alpha, \beta\} \, [P]$$ □

Definition 36 If α and β are connected by an undirected edge they are said to be *adjacent* or *neighbours*. Let A be a set of vertices in a graph $\mathcal{G} = (V, E)$. The *boundary* of A, denoted $\mathrm{bd}(A)$, is the set of vertices in $V \setminus A$ which are neighbours to vertices in A. The boundary of a single vertex α is $\mathrm{bd}(\{\alpha\})$ and is denoted $\mathrm{bd}(\alpha)$. The *closure* of A, denoted $\mathrm{cl}(A)$ is $A \cup \mathrm{bd}(A)$. The closure of a single vertex α is $\mathrm{cl}(\{\alpha\})$ and is denoted $\mathrm{cl}(\alpha)$. □

Definition 37 A joint distribution P over variables V obeys the *local Markov property* relative to an undirected graph $\mathcal{G} = (V, E)$, if for any vertex $\alpha \in V$, we have:

$$\alpha \perp V \setminus \mathrm{cl}(\alpha) | \mathrm{bd}(\alpha) \, [P]$$ □

It can be shown [9] that if a distribution obeys the global Markov property for some graph then it obeys the local Markov property for that graph (but not necessarily vice-versa). Also the local Markov property implies the pairwise one (but not vice-versa). Theorem 3, which is the Hammersley-Clifford theorem, states when the local pairwise Markov property implies factorisation.

Theorem 3 ***Hammersley-Clifford theorem.*** *A positive joint probability distribution P satisfies the pairwise Markov property with respect to an undirected graph $\mathcal{G}$ iff it factorises according to $\mathcal{G}$.* □

The proof of the Hammersley-Clifford theorem can be found in [9]. So as long as a distribution defines a non-zero probability to each joint instantiation then all three Markov properties and the factorisation property are equivalent.

`VisitAsia`	A
`Tuberculosis`	T
`XRay`	X
`Dyspnea`	D
`Bronchitis`	B
`Smoking`	S
`TbOrCa`	E
`Cancer`	L

Table 5.1: Abbreviations of 'Asia' example variables. (`Cancer` was 'lung cancer' and `TbOrCa` was 'either tuberculosis or lung cancer' in the original paper.)

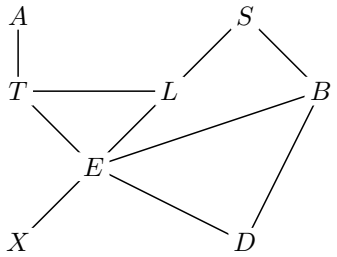

Figure 5.6: The interaction graph for the 'Asia' example distribution $P(A, B, D, E, L, S, T, X)$

5.4 Exploiting conditional independence

If a distribution has a known factored representation then its interaction graph can be used to read off conditional independence relations and thus speed up computations. This will be demonstrated using (as always) the 'Asia' example. Since the 'Asia' variable names are rather long they will be abbreviated as in the original paper [10], see Table 5.1.

A factored representation of the distribution is displayed in Fig 4.8. Using the abbreviations from Table 5.1 the hypergraph is:

$$\{\{A\}, \{S\}, \{A, T\}, \{S, L\}, \{S, B\}, \{T, E, L\}, \{E, X\}, \{B, D, E\}\}$$

and the interaction graph is given in Fig 5.6.

Consider now the task of computing the marginal conditional distribution $P(A|L = \texttt{absent}, E = \texttt{false})$. From the interaction graph in Fig 5.6 it is clear that $A, T \perp \{S, B, D, X\}|\{L, E\}$

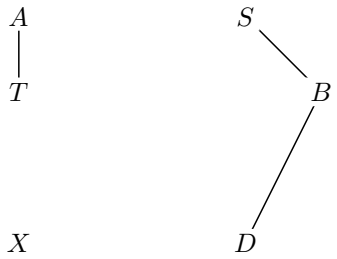

Figure 5.7: The interaction graph for $P(A, B, D, S, T, X|L = a, E = f)$

$$
\begin{aligned}
&P(A|L = a, E = f) \\
&\quad = \sum_{T,X,S,B,D} P(A, B, D, S, T, X|L = a, E = f) \\
&\quad = \sum_{T,X,S,B,D} P(A, T|L = a, E = f) P(B, D, S, X|L = a, E = f) \\
&\quad = \sum_{T} P(A, T|L = a, E = f) \sum_{B,D,S,X} P(B, D, S, X|L = a, E = f) \\
&\quad = \sum_{T} P(A, T|L = a, E = f)
\end{aligned}
\tag{5.4}
$$

This can be done using naïve variable elimination as detailed in Section 4.9 and in `gPy` Example 22.

5.5 Hierarchical models

The previous sections of this chapter have shown that if a probability distribution has a factored distribution then its associated interaction graph provides a useful picture of that distribution's conditional independence properties. The Hammersley-Clifford theorem provides an inference in the other direction: as long a distribution is positive then obeying even just the pairwise Markov property with respect to a graph is enough to guarantee that the distribution has some factored representation using the cliques of that graph. Now it is important to be a bit more precise about how a graph represents the conditional independence structure of a distribution.

Let P obey the global Markov property with respect to graph $\mathcal{G}$. Does it follow that $\mathcal{G}$ reveals *all* the conditional independence structure of P? In general, no: since P may have *additional* conditional independence relations not revealed by $\mathcal{G}$. (An appreciation of this point is crucial when we come to look at Bayesian networks.) To see this consider $\mathcal{G} \uparrow$ the set of all graphs formed by adding edges to $\mathcal{G}$. It is obvious that for any $\mathcal{G}' \in \mathcal{G} \uparrow$, P also obeys the

global Markov property with respect to $\mathcal{G}'$: adding edges never increases the number of separators in the graph and so no extra conditional independence relations will be imposed on P.

to do

Chapter 6

Decomposable models and join forests

The variable elimination algorithm takes a factored representation as input and produces a (factored representation of) a marginal distribution as output. As mentioned in Chapter 4 this approach is not efficient if the goal is to produce *several* different marginal distributions. In this chapter an alternative (albeit related) algorithm is given. The basic idea is simple: given a factored representation as input, compute a different factored representation of the same distribution from which the desired marginals can be easily extracted.

6.1 Join forest calibration by example

The algorithm we will use to achieve this task builds a data structure called a *join forest* and then runs an algorithm known as calibration to compute the desired output. To properly explain join forest calibration several new concepts are required, but since the basic idea is straightforward a brief example-driven overview of how it works will first be given. The 'Asia' factored representation will be used as an example and for convenience the representation itself, its hypergraph and interaction graph are reproduced here as Figs 6.1–6.3, respectively. (Note that the abbreviations introduced in Table 5.1 are being used.)

Our goal is to compute a new factored representation of the 'Asia' distribution from which marginals are easily extracted. As a first step towards this we first use the *chain rule* to rewrite the joint probability distribution as a product of conditional probability distributions. To see why the first equation in (6.1) holds, note that *by definition* the conditional distribution $P(A|T,X,D,B,S,E,L)$ equals $P(A,T,X,D,B,S,E,L)/P(T,X,D,B,S,E,L)$ from which the equation obviously follows. Having used the chain rule once to factorise $P(A,T,X,D,B,S,E,L)$ into the product of $P(A|T,X,D,B,S,E,L)$ and $P(T,X,D,B,S,E,L)$ it can be used again to break up $P(T,X,D,B,S,E,L)$ and so on until we have the final product on the last two lines of (6.1).

B	D	E	
absent	absent	false	0.90
absent	absent	true	0.30
absent	present	false	0.10
absent	present	true	0.70
present	absent	false	0.20
present	absent	true	0.10
present	present	false	0.80
present	present	true	0.90

B	S	
absent	nonsmoker	0.70
absent	smoker	0.40
present	nonsmoker	0.30
present	smoker	0.60

L	S	
absent	nonsmoker	0.99
absent	smoker	0.90
present	nonsmoker	0.01
present	smoker	0.10

L	E	T	
absent	false	absent	1.00
absent	false	present	0.00
absent	true	absent	0.00
absent	true	present	1.00
present	false	absent	0.00
present	false	present	0.00
present	true	absent	1.00
present	true	present	1.00

S	
nonsmoker	0.50
smoker	0.50

E	X	
false	abnormal	0.05
false	normal	0.95
true	abnormal	0.98
true	normal	0.02

T	A	
absent	no_visit	0.99
absent	visit	0.95
present	no_visit	0.01
present	visit	0.05

A	
no_visit	0.99
visit	0.01

Figure 6.1: A factored representation of the 'Asia' distribution containing 8 factors: $f_{B,D,E}, f_{B,S}, f_{L,S}, f_{E,L,T}, f_S, f_{E,X}, f_{A,T}$ and f_A.

$$\{\{A\},\{S\},\{A,T\},\{S,L\},\{S,B\},\{T,E,L\},\{E,X\},\{B,D,E\}\}$$

Figure 6.2: Hypergraph of the factored representation of the 'Asia' distribution given in Fig 6.1.

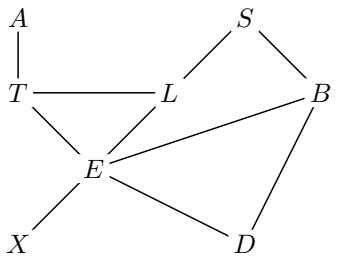

Figure 6.3: Interaction graph of the factored representation of the 'Asia' distribution given in Fig 6.1.

$$\begin{aligned}
&P(A,T,X,D,B,S,E,L)\\
&\quad = \quad P(A|T,X,D,B,S,E,L)P(T,X,D,B,S,E,L)\\
&\quad = \quad P(A|T,X,D,B,S,E,L)P(T|X,D,B,S,E,L)P(X,D,B,S,E,L)\\
&\quad = \quad P(A|T,X,D,B,S,E,L)P(T|X,D,B,S,E,L)P(X|D,B,S,E,L)P(D,B,S,E,L)\\
&\quad \dots\\
&\quad = \quad P(A|T,X,D,B,S,E,L)P(T|X,D,B,S,E,L)P(X|D,B,S,E,L)\\
&\qquad \times P(D|B,S,E,L)P(B|S,E,L)P(S|E,L)P(E|L)P(L)
\end{aligned} \tag{6.1}$$

It is not clear that this decomposition is getting us anywhere. Firstly, (6.1) contains *conditional* marginals when our goal is to get simply marginals, and secondly if each of these conditional distributions were represented by a single `Factor` object then some of them, having many variables, would be very big.

The latter of these two problems can be addressed by exploiting the conditional independence structure of the distribution. From the interaction graph in Fig 6.3 it is evident that, for example, $A \perp X,D,B,S,E,L|T\,[P]$. It follows that the $P(A|T,X,D,B,S,E,L)$ term in (6.1) can be replaced with $P(A|T)$. By inspecting all conditional independence properties for all the other variables we can produce the equation in (6.2) for the joint distribution.

$$\begin{aligned}
&P(A,T,X,D,B,S,E,L)\\
&\quad = \quad P(A|T)P(T|E,L)P(X|E)P(D|B,E)P(B|S,E)P(S|E,L)P(E|L)P(L)
\end{aligned} \tag{6.2}$$

So by exploiting conditional independence a more compact representation of the joint distribution is possible: if each of these conditional marginal distributions were represented by a single `Factor` object then none would be too big. But these marginals are still conditional. To get hold of unconditional marginals it is necessary to go beyond the factored representations that have been used so far and use *quotient factored representations*: representing distributions by multiplication *and division* of factors. In particular, from the definition of conditional probability (6.2) can be rewritten as (6.3).

$$\begin{aligned} & P(A,T,X,D,B,S,E,L) \\ &= \frac{P(A,T)P(T,E,L)P(X,E)P(D,B,E)P(B,S,E)P(S,E,L)P(E,L)P(L)}{P(T)P(E,L)P(E)P(B,E)P(S,E)P(E,L)P(L)} \end{aligned} \tag{6.3}$$

From the marginals in the RHS of (6.3) it is now easy to compute marginals for any given individual variable. Some, like $P(E)$ and $P(L)$ are immediately available; others can be computed by finding the smallest marginal containing the desired variable and summing out all other variables. For example, it makes sense to compute $P(S)$ from $P(S,E)$ rather than from $P(B,S,E)$.

The expression in (6.3) could evidently be replaced by a more compact representation by cancelling common terms in the numerator and denominator which produces (6.4). For expositional reasons the variables in each marginal have also been ordered alphabetically.

$$\begin{aligned} & P(A,T,X,D,B,S,E,L) \\ &= \frac{P(A,T)P(E,L,T)P(E,X)P(B,D,E)P(B,E,S)P(E,L,S)}{P(T)P(E,L)P(E)P(B,E)P(E,S)} \end{aligned} \tag{6.4}$$

The RHS of (6.4) is the representation which we aim to compute. We will see later that the structure of (6.4) is intimately connected to the previously mentioned join forest. The next issue is how to compute (6.4) from the 8 factors $f_{B,D,E}, f_{B,S}, f_{L,S}, f_{E,L,T}, f_S, f_{E,X}, f_{A,T}$ and f_A in Fig 6.1. A proper explanation of this computation (known as calibration) will be provided later in this chapter. For now just note that it is possible to immediately represent $P(A,T,X,D,B,S,E,L)$ as a quotient factored representation with the same structure as (6.4) using the input 8 factors and additional *identity factors* which are just factors all of whose values are 1. Denoting, for example, the factor that maps each joint instantiation of E and S to 1 by $1_{E,S}$, the equations in (6.5) follow.

$$
\begin{aligned}
&P(A,T,X,D,B,S,E,L)\\
&\quad = Z^{-1} f_A f_{A,T} f_{E,L,T} f_{E,X} f_{B,D,E} f_{B,S} f_S f_{L,S}\\
&\quad = Z^{-1} \frac{[f_A f_{A,T}][f_{E,L,T}][f_{E,X}][f_{B,D,E}][f_{B,S} 1_E][f_S f_{L,S} 1_E]}{[1_T][1_{E,L}][1_E][1_{B,E}][1_{E,S}]} \qquad (6.5)\\
&\quad = \frac{P(A,T)P(E,L,T)P(E,X)P(B,D,E)P(B,S,E)P(E,L,S)}{P(T)P(E,L)P(E)P(B,E)P(E,S)} \qquad (6.6)\\
& \qquad (6.7)
\end{aligned}
$$

Basically, calibration 'moves information around' between factors so that (6.5) is changed into (6.6). Now we dive into the necessary theory to understand how this is done.

6.2 Decomposable hypergraphs and triangulated graphs

Consider the 6 factors in the numerator of (6.5). This factored representation has been generated by multiplying together some factors of the original factored representation in Fig 6.1 and broadcasting some others by multiplying by identity factors. It is instructive to compare the hypergraph of the original representation:

$$\mathcal{H}_1 = \{\{A\},\{S\},\{A,T\},\{L,S\},\{B,S\},\{E,L,T\},\{E,X\},\{B,D,E\}\}$$

to that of the numerator of (6.5):

$$\mathcal{H}_2 = \{\{A,T\},\{E,L,S\},\{B,S,E\},\{E,L,T\},\{E,X\},\{B,D,E\}\}$$

Clearly, $\mathcal{H}_2$ is a cover of $\mathcal{H}_1$ (recall Definition 32 on page 89). $\mathcal{H}_2$ also has the desirable property of being a *decomposable* hypergraph, and hence is a *decomposable cover* of $\mathcal{H}_1$. We now address the following issues:

1. what makes a hypergraph decomposable
2. why decomposability matters
3. how to construct decomposable covers

6.2.1 Triangulated graphs

We will approach decomposable hypergraphs by way of triangulated graphs. This is just simpler and also more standard since many presentations of graphical models make far less use of hypergraphs than is done here.

It is instructive to compare the interaction graph $\mathcal{G}_1$ for the original factored representation which can be found in Fig 6.3 to the interaction graph $\mathcal{G}_2$ for the

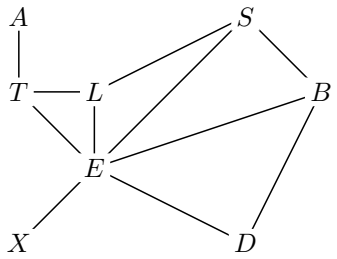

Figure 6.4: Interaction graph $\mathcal{G}_2$ for the numerator of (6.6).

numerator of (6.6) which is in Fig 6.4. (By definition, $\mathcal{G}_1$ is the 2-section of $\mathcal{H}_1$ and $\mathcal{G}_2$ the 2-section of $\mathcal{H}_2$.) The crucial difference between these two graphs is that $\mathcal{G}_2$ is *triangulated* whereas $\mathcal{G}_1$ is not. Definition 38 defines what this means.

Definition 38 Recall the definition of path from Definition 28 on page 87. A *cycle of length* n is a path $\alpha = \alpha_0, \alpha_1, \ldots, \alpha_n = \beta$ where $\alpha = \beta$. A cycle contains a *chord* if two non-consecutive vertices are neighbours. An undirected graph is *triangulated* if every cycle of length $n \geq 4$ has a chord. A synonym for 'triangulated graph' is *chordal graph*. □

$\mathcal{G}_1$ is not triangulated since it contains the cycle E, L, S, B, E which is of length 4 and contains no chord. $\mathcal{G}_2$, on the other hand, contains no such cycle; all its cycles are either of length 3 or have chords.

Triangulated graphs have a number of useful properties, the most important of which is that they can be *decomposed*, i.e. broken apart in a particular way. Definitions 39 and 40 (taken verbatim from [5]) provides the necessary definitions. The classic Theorem 4 states that decomposable and triangulated graphs are one and the same.

Definition 39 A triple (A, B, C) of disjoint subsets of the vertex set V of an undirected graph $\mathcal{G}$ is said to form a *decomposition* of $\mathcal{G}$, or to *decompose* $\mathcal{G}$, if $V = A \cup B \cup C$, and the following two conditions hold:

1. C separates A from B;
2. C is a complete subset of V. □

Note that we allow any of the sets A, B or C to be empty. If both A and B are non-empty, we say the decomposition is *proper*.

Definition 40 An undirected graph is *decomposable* if either (i) it is complete or (ii) it possesses a proper decomposition (A, B, C) such that both subgraphs $\mathcal{G}_{A\cup C}$ and $\mathcal{G}_{B\cup C}$ are decomposable. □

Theorem 4 *The following three conditions are equivalent for an undirected graph* $\mathcal{G}$*:*

1. $\mathcal{G}$ *is decomposable;*

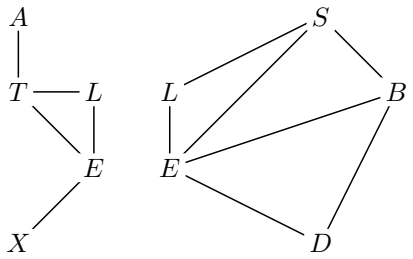

Figure 6.5: Decomposition $(\{A, T, X\}, \{E, L\}, \{B, D, S\})$ of graph $\mathcal{G}_2$

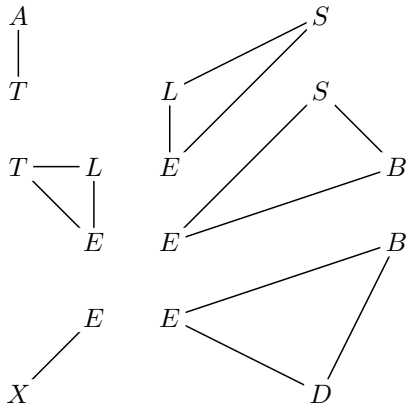

Figure 6.6: Complete decomposition of graph $\mathcal{G}_2$

2. $\mathcal{G}$ is triangulated;

3. Every minimal (α, β)-separator is complete □

PROOF See [5] or [9]. ■

Although the proof of Theorem 4 is not given we at least show that $\mathcal{G}_2$ is decomposable. Fig 6.5 shows a first decomposition of $\mathcal{G}_2$ and Fig 6.6 shows the result of further decompositions producing 6 complete graphs. In contrast, Fig 6.7 shows that $\mathcal{G}_2$ is not (fully) decomposable. The graph with vertices $\{B, E, L, S\}$ is not complete but cannot be decomposed.

6.2.2 Deciding whether a graph is triangulated

To decide whether an undirected graph is triangulated we first introduce the notion of a *fill-in* produced by an ordering of the vertices of a graph. The basic idea is this: eliminate (i.e. delete) the vertices in some order; when a vertex is deleted add edges, if necessary, so that all its neighbours are directly connected (i.e. adjacent). The fill-in is the set of added edges, if any. Definition 41 gives the formal definition, provides an alternative definition and introduces notation.

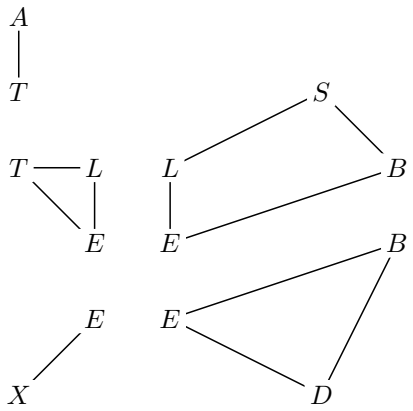

Figure 6.7: Decomposition of $\mathcal{G}_1$. Since the top-right graph is not complete this is not a full decomposition.

Definition 41 Let α be an ordering of the vertices V of an undirected graph $\mathcal{G} = (V, E)$. Consider deleting the vertices V in the order α. When a vertex is to be deleted add extra edges, if necessary, to ensure that all its neighbours are adjacent. The set of extra edges is the the *fill-in* produced by α and is denoted $F(\alpha)$. An alternative definition given by [14] is: $F(\alpha) = \{\{v, w\} : \{v, w\} \notin E$ and there is a path from v to w containing only v, w and vertices ordered before both v and $w\}$. The *elimination graph* of $\mathcal{G}$ with ordering α is the original graph with the fill-in added and is denoted $\mathcal{G}(\alpha)$: $\mathcal{G}(\alpha) = (V, E \cup F(\alpha))$. If $F(\alpha) = \emptyset$ then α is a *zero fill-in ordering* of $\mathcal{G}$. □

gPy **Example 26 (Triangulation demo)**
To see the fill-ins produced on an example graph by two different orderings do:

```
>>> from gPy.Examples import *
>>> triangulation_demo()
```

In the top frame the ordering `VisitAsia`, `Tuberculosis`, `XRay`, `Dyspnea`, `Bronchitis`, `Smoking`, `TbOrCa`, `Cancer` *is considered. Each click on 'Next' progresses the simple elimination algorithm for computing the fill-in. The vertex to be eliminated in in red, its neighbours are in green. Eliminated vertices are greyed out. Fill-in edges are blue. In the first ordering only one fill-in edge is needed. It's a different story for the second ordering in the bottom frame!*

Theorem 5 provides the central result linking fill-ins and triangulated graphs.

Theorem 5 *A graph is triangulated if and only if it has a zero fill-in ordering.*□

This theorem can apparently be traced back to the work of Dirac in the early 1960s [6]. Proofs can be found in many places, [14] cite [12]. Theorem 5 leads to an algorithm to check whether a graph is triangulated: search for a zero fill-in, the graph is triangulated if and only if one can be found.

Fortunately, it is not necessary to plod through all possible vertex orderings in search of a zero fill-in. If an ordering is constructed by *maximum cardinality search* then [14] have shown that the ordering is a zero fill-in if and only if the graph is triangulated. Maximum cardinality search is very simple: "Number the vertices from n to 1 in decreasing order. As the next vertex to number, select the vertex adjacent to the largest number of previously numbered vertices, breaking ties arbitrarily" [14]. The search gets its name by calling the number of previously numbered vertices adjacent to a given vertex the vertex's 'cardinality'.

`gPy` **Example 27 (Maximum cardinality search demo)**
To see maximum cardinality search in action on two different graphs just do:

```
>>> from gPy.Examples import *
>>> max_card_search_demo()
```

Click 'Next' to progress the demonstration. Note that the ordering is constructed backwards from the last vertex. The top graph is not triangulated but the bottom one (an example from [14]) is, and so the order found is a zero fill-in.

Algorithm 3 is the `gPy` implementation of maximum cardinality search. There are only two small differences from the algorithm presented in [14]: the numbering is from $n-1$ to 0, and the user can specify a tie-breaker (the `choose` function) if desired.

Algorithm 3 Maximum cardinality search on a graph

```
def maximum_cardinality_search(self,choose=None):
    if choose is None:
        def choose(set): return set.pop()

    vertices = self.vertices()    # 1
    sets = []                     # 2
    alpha = {}                    # 3
    alpha_inv = []                # 4
    cardinality = {}              # 5
    for vertex in vertices:       # 6
        cardinality[vertex] = 0   # 7
        sets.append(set())        # 8
        alpha_inv.append(None)    # 9
    sets.append(set())            # 9a
    sets[0] = vertices            #10
    j = 0                         #12

    for i in range(len(vertices)-1,-1,-1): # 13
        v = choose(sets[j])   # 14
        alpha[v] = i          # 15
        alpha_inv[i] = v      # 16
        for w in self._ne[v].difference(alpha): # 17
            card_w = cardinality[w]             # 18
            sets[card_w].remove(w)              # 19
            card_w += 1                         # 20
            sets[card_w].add(w)                 # 21
            cardinality[w] = card_w             # 22
        j += 1                                 # 23
        while j >= 0 and sets[j] == emptyset: # 24
            j -= 1                             # 25
    return alpha, alpha_inv     # 26
```

Lines 1–12 are initialisation. The key data structure is the dictionary `cardinality` which throughout maps each unnumbered vertex to the number of numbered vertices adjacent to it: its *cardinality*. The list `sets` is such that `sets[i]` is a set containing all unnumbered vertices with cardinality `i`. Initially no vertices are numbered so all vertices have cardinality zero (lines 7 and 10). `alpha` and `alpha_inv` will contain the vertex numbering on return. `alpha` maps each vertex to its number, and `alpha_inv` is just the inverse mapping (see lines 17 and 18). Eventually all the `None` values in `alpha_inv` (line 9) will be replaced by the appropriate integers. `j` is the largest value such that `sets[j]` is non-empty. (Line 9a adds a dummy empty set so that `j` in line 24 never causes an `IndexError`.)

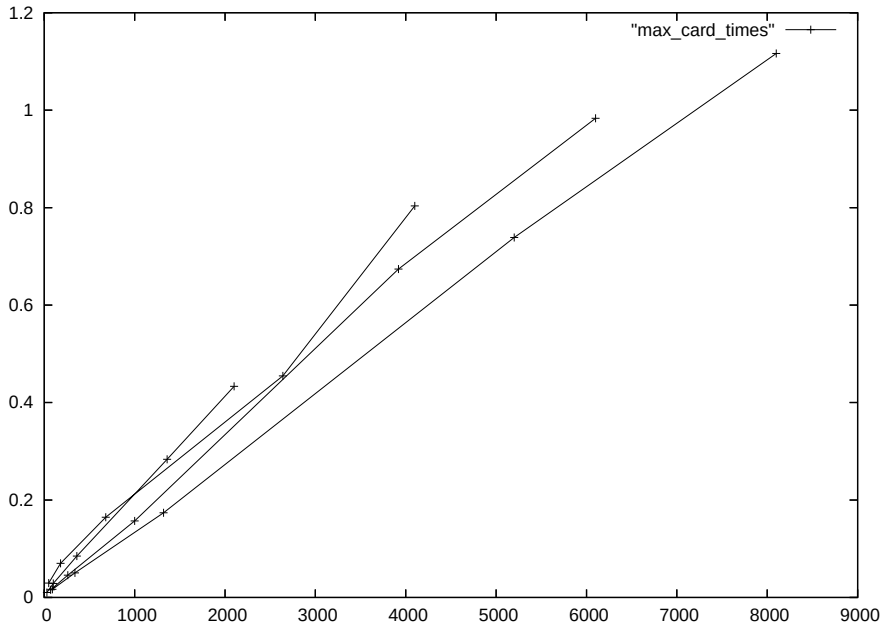

Figure 6.8: Empirical confirmation that maximum cardinality search is $O(n + m)$. x-axis is $n+m$ for randomly generated graphs, y-axis is time taken to do 100 maximum cardinality searches. The 4 lines correspond to different level of graph sparsity. To generate the leftmost, shortest line a parameter r was set to 0.2. Graphs with $n = 10, 20, 40, 80, 100$ vertices were created with $m = rn^2$ randomly placed edges. The other three lines correspond (left to right) to setting $r = 0.4, 0.6, 0.8$. (These timings were created by running `gPy.Examples.time_mc`.)

In the main loop (lines 13-26) we number vertices from `i=n-1` down to `i=0` (The builtin `range` function (13) generates a list with precisely this sequence of integers.) The loop invariant is that `cardinality`, `sets` and `j` maintain the properties described above. In particular, `sets[j]` contains all unnumbered vertices with maximum cardinality.

Line 14 assigns `v` to be a maximum cardinality vertex and removes it from `sets[j]`. This assignment is then stored (lines 15,16). Now that `v` has been numbered, the cardinalities of all its unnumbered neighbours (these neighbours are given by the set `self._ne[v].difference(alpha)` in line 17) need to be incremented. Lines 17-22 update `sets` and `cardinality` accordingly. Lines 23-35 maintain the loop invariant for `j`.

Once an ordering has been found using maximum cardinality search it remains to check whether this ordering is a zero fill-in. The temptation is to use a simple algorithm like Algorithm 4. This algorithm works its way through the `elimination_order`. For each `vertex` its '`live_neighbours`'—those neighbours which have yet to be eliminated—are considered. If any pair of these

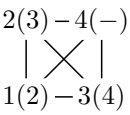

Figure 6.9: A complete graph. The number in parentheses is the vertex's follower, if any.

`live_neighbours` are not themselves neighbours then `elimination_order` is not a zero fill-in, and so if `elimination_order` has been found by maximum cardinality search it follows that the graph `self` is not triangulated.

Algorithm 4 Naïve check that an ordering is a zero fill-in

```
def zero_fillin_check(self,elimination_order):
    eliminated = set()
    for vertex in elimination_order:
        live_neighbours = self._ne[vertex] - eliminated
        while live_neighbours != emptyset:
            nbr = live_neighbour.pop()
            for other_nbr in live_neighbours:
                if other_nbr not in self._ne[nbr]:
                    return False
        eliminated.add(vertex)
    return True
```

The problem with Algorithm 4 is that it can do the same work over and over again. Suppose, for example, that `self` were a complete graph (i.e. all pairs of vertices are neighbours) such as that displayed in Fig 6.9 (ignore the numbers in parentheses for now). For such a graph any elimination order is a zero fill-in and so *a fortiori* the graph is triangulated. Consider running Algorithm 4 on a complete graph with many vertices. When eliminating the first vertex all other pairs of vertices are checked for adjacency. When eliminating the second vertex, most of these pairs are checked *again*, and this problem will occur for the rest of the vertices. The basic problem is that Algorithm 4 does not keep track of what work has already been done and thus is doomed to redo much of it.

A superior approach is provided, once again, by [14]. Their algorithm exploits the notion of a vertex's *follower*. The follower of vertex v is the earliest vertex in the elimination order coming after v which is adjacent to v in the elimination graph (i.e. the graph formed by adding the fill-in edges). A vertex need not have a follower—the last vertex never has one.

Define $f^*(x)$ to be $\{x, f(x), f(f(x)), ...\}$, i.e. the path of followers starting with x. Theorem 3 of [14] establishes the following. If v comes before w in the elimination ordering then there is a $v - w$ edge in the elimination graph if and only if there is a vertex x which is (i) adjacent to w in the original graph and (ii) where v is a member of $f^*(x)$.

This result leads to the following algorithm. The vertices are processed in elimination order. For each vertex w the neighbours which come before w are considered. For each of these neighbours x, the vertices in $f^*(x)$ coming before w are found and connected to w.

A variant of this is implemented as Algorithm5 follows. Dictionary `f` stores the 'follower' mapping and the list `fill_in` records fill-in edges. They are initialised in lines 1 and 2. Vertices `w` are considered according to the given order (line 3).

A dummy value of `w` for the follower of `w` is assigned (line 4). `self._ne[w]` on line 5 is the set of neighbours of `w`. By intersecting this with the dictionary `f` the variable `earlier_neighbours` contains only neighbours of `w` earlier in the elimination order. The set `done` (of which more later) is initialised to these neighbours together with `w`.

Each earlier neighbour `x` is considered in turn (line 7). If the follower of `x` has yet to be found then `f[x]` in line 8 is `x` otherwise it is the follower of `x`. In the latter case, the chain of followers of `x` are considered (lines 9-13) until we hit one that has already been 'done'. Any vertex in such a chain is a follower of an earlier neighbour of `w` which is not itself an existing neighbour of `w` and so a fill-in edge is needed (line 11). `x` at line 11 cannot be an existing neighbour since all earlier neighbours are in the set 'done' and x is not (line 9) and all later neighbours cannot be reached yet by a chain of followers. Note that the set 'done' also prevents revisiting sequences of followers due to line 10. Lines 9a and 9b provide an early exit if all that is required is a check for a zero fill-in.

If the `x` at the end of one of these chains is found to contain a dummy value for its follower this is replaced by the true value—`w`—in lines 13–14. Lines 16–18 add the fill-in edges to the graph (if that is required) and return the fill-in edges. Lines 14a and 14b return the appropriate boolean value if a check is being done.

At the cost of a small amount of extra storage (the `f` dictionary and the `done` set) Algorithm 5 is a linear algorithm compared to the quadratic Algorithm 4. Working through how Algorithm 5 operates on the graph in Fig 6.9 is a useful way of understanding how the algorithm works. Vertex followers are given in parentheses. Fig 6.10 displays an empirical comparison of the two algorithms.

6.2.3 Decomposable hypergraphs

Having introduced triangulated graphs we now define the closely related notion of *decomposable hypergraph*. There are a number of equivalent characterisations of what it means for a hypergraph to be decomposable—[1] provide no fewer than 12 equivalent conditions—and different authors make different choices about which is *the* definition. Here we follow [14] and use Definition 42.

Definition 42 A hypergraph $\mathcal{H}$ is *decomposable* if it is graphical and its 2-section $\mathcal{H}_{[2]}$ is decomposable. A synonym for 'decomposable hypergraph' is *acyclic hypergraph*.

A key property of decomposable hypergraphs is that each decomposable hypergraph has an associated *join forest*. To define a join forest we first define a

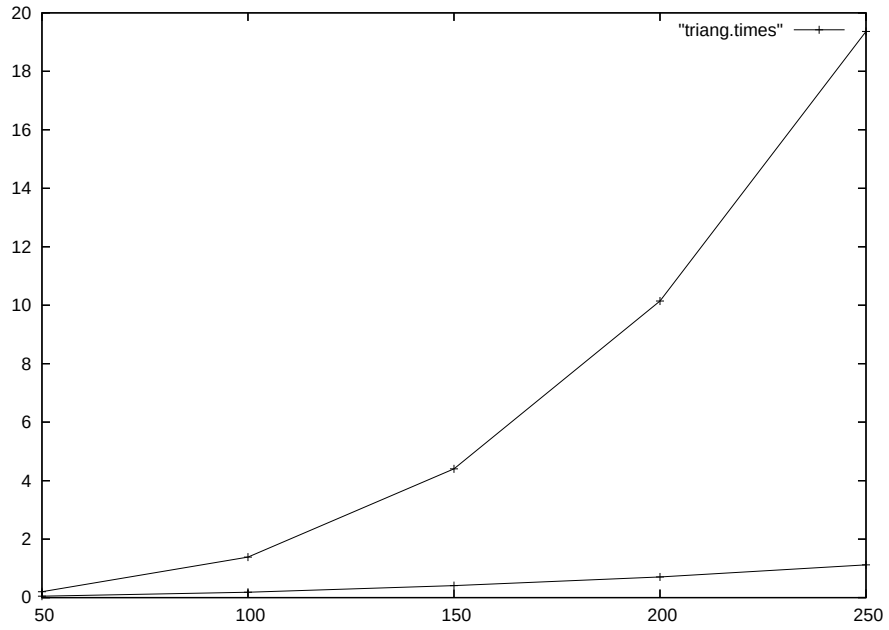

Figure 6.10: Empirical confirmation that naive triangulation takes quadratic time but Tarjan and Yannakakis's algorithm is linear. x-axis is the number of nodes in a complete graph of n nodes with datapoints for $n = 50, 100, 150, 200, 250$. y-axis is time taken to check for triangulation 10 times. Lower line gives times using Tarjan and Yannakakis's algorithm, upper curve gives times using a naive approach. (These timings were created by running `gPy.Examples.time_chk`.)

Algorithm 5 Tarjan and Yannakakis's fill-in algorithm

```
def triangulate(self,elimination_order,
                zero_fillin_check=False,modify=True):
    f = {}          # 1
    fill_in = []    # 2
    for w in elimination_order:                          # 3
        f[w] = w                                         # 4
        earlier_neighbours = self._ne[w].intersection(f) # 5
        done = set([w]) | earlier_neighbours             # 6
        for x in earlier_neighbours:           #  7
            x = f[x]                           #  8
            while x not in done:               #  9
                if zero_fillin_check:            #  9a
                    return False                 #  9b
                done.add(x)                    # 10
                fill_in.append((x,w))          # 11
                x = f[x]                       # 12
            if f[x] == x:                      # 13
                f[x] = w                       # 14
    if zero_fillin_check:                        # 14a
        return True                              # 14b
    if modify:                  # 15
        self.put_lines(fill_in) # 16
    return fill_in              # 17
```

related notion: the *representative graph* of a hypergraph. Definition 43 defines this graph formally and Figs 6.11 and 6.12 display the representative graphs for the two hypergraphs under discussion. Although the formal definition (see [7, p. 387]) defines a representative graph for any hypergraph, note that Definition 43 only considers representative graphs for simple hypergraphs. This avoids complications arising from repeated vertices in the representative graph.

Definition 43 The *representative graph* $L(\mathcal{H})$ of a simple hypergraph $\mathcal{H}$ is the undirected graph whose vertices are the hyperedges of $\mathcal{H}$ and where two vertices are connected if the two corresponding hyperedges have a non-empty intersection. Synonyms for 'representative graph' include: *line-graph* and *intersection graph*

Next we define some graph-theoretical notions which are subsidiary to the definition of a join forest in Definition 44.

Definition 44 For any undirected graph, define the relation $\sim^*$ so that $v \sim^* w$ if there is a path from v to w in the graph. $\sim^*$ is clearly an equivalence relation and call the corresponding equivalence classes the *connectivity components* of the graph. A *tree* is a connected, undirected graph without cycles. A *forest* is

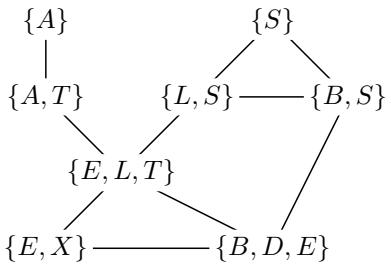

Figure 6.11: The representative graph $L(\mathcal{H}_1)$ for the hypergraph $\mathcal{H}_1 = \{\{A\}, \{S\}, \{A, T\}, \{L, S\}, \{B, S\}, \{E, L, T\}, \{E, X\}, \{B, D, E\}\}$

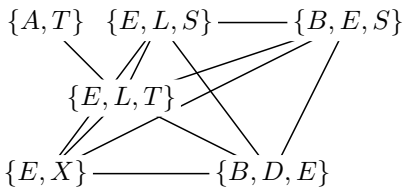

Figure 6.12: The representative graph $L(\mathcal{H}_2)$ for the hypergraph $\mathcal{H}_2 = \{\{A, T\}, \{E, L, S\}, \{B, E, S\}, \{E, L, T\}, \{E, X\}, \{B, D, E\}\}$

an undirected graph whose connectivity components are all trees. A tree which connects all the vertices of a connected undirected graph is a *spanning tree* for that graph. Given a graph, not necessarily connected, a forest is a *spanning forest* for the graph if it contains exactly one spanning tree for each connectivity component of the graph. □

Definition 45 finally gives the definition of a join forest.

Definition 45 Let $\mathcal{F}$ be a spanning forest for the representative graph $L(\mathcal{H})$ of a hypergraph $\mathcal{H}$. Note that, for any two hyperedges $h_1, h_2 \in \mathcal{H}$ in the same connectivity component of $\mathcal{F}$, there is a unique path between h_1 and h_2 in $\mathcal{F}$. $\mathcal{F}$ is a *join forest* if each hyperedge in the path between h_1 and h_2 contains the set $h_1 \cap h_2$ for any choice of $h_1, h_2 \in \mathcal{H}$ where h_1 and h_2 are in the same connectivity component. A synonym for 'join forest' is *junction forest*

Theorem 6 provides a useful equivalent characterisation of join forests.

Theorem 6 *Let $\mathcal{F}$ be a spanning forest for the representative graph $L(\mathcal{H})$ of a hypergraph $\mathcal{H}$. $\mathcal{F}$ is a join forest if and only if the star of any vertex in $\mathcal{H}$ induces a connected subtree of $\mathcal{F}$.* □

The key result is stated in Theorem 7.

Theorem 7 *A hypergraph is decomposable if and only if it has a join forest.* □

Proof See [9]. ■

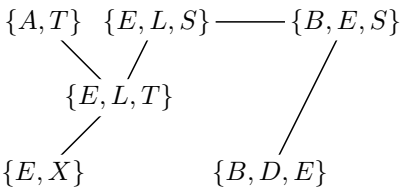

Figure 6.13: A join tree for $\mathcal{H}_2 = \{\{A,T\},\{E,L,S\},\{B,E,S\},\{E,L,T\},\{E,X\},\{B,D,E\}\}$

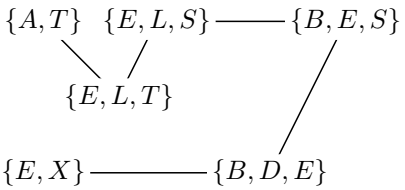

Figure 6.14: An alternative join tree for $\mathcal{H}_2 = \{\{A,T\},\{E,L,S\},\{B,E,S\},\{E,L,T\},\{E,X\},\{B,D,E\}\}$

Although Definition 45 and Theorem 7 are given in terms of join *forests*, a join forest for a decomposable hypergraph will contain only one tree if its representative graph is connected. Since this is commonly the case most presentations of the calibration algorithm ([9] is an exception) are given in terms of *join trees* (or synonymously *junction trees*). When dealing with connected hypergraphs we too will talk about join trees rather than pedantically insisting that we are dealing with a different object: a join forest containing one tree. (In the `gPy` implementation such informality cannot be permitted and everything is done with join forests.)

Fig 6.13 shows a join tree for the decomposable hypergraph $\mathcal{H}_2$. Note that it can be produced by deleting edges from the representative graph for that hypergraph. It is simple to check that this is a join tree, just take any two hyperedge-vertices of the tree, find their intersection and check that this intersection is contained in all hyperedge-vertices in the path between the original two hyperedge-vertices. For example, consider the hyperedge-vertices $\{E,X\}$ and $\{B,D,E\}$. Their intersection is $\{E\}$ and the path between them is

$$\{E,X\},\{E,L,T\},\{E,L,S\},\{B,E,S\},\{B,D,E\}$$

. $\{E\}$ is a subset of each hyperedge on this path as expected. We can also check that it is a join tree by using stars. The star of, for example, L is $\{\{E,L,T\},\{E,L,S\}\}$ which is indeed a connected subtree. The same holds for all other vertex stars. Note that a decomposable hypergraph generally has more than one join tree; Fig 6.14 shows an alternative join tree for $\mathcal{H}_2$.

6.2.4 Deciding whether a hypergraph is decomposable

Recall from Definition 40 (page 100) that a hypergraph is decomposable iff it is graphical and its 2-section is triangulated. It follows that it is possible to check a hypergraph's decomposability by constructing its 2-section and then (i) checking that all of the 2-section's cliques are partial hyperedges of the hypergraph (so that the hypergraph is graphical) and (ii) checking that the 2-section is triangulated using the algorithms given in Section 6.2.2.

This is an inefficient approach. To take an example flagged up by [14], suppose we have a hypergraph with n vertices and a single hyperedge. The 2-section is a complete graph with $n(n-1)/2$ edges. So even to construct the 2-section is $O(n^2)$. It turns out that by conducting the decomposability check directly on the hypergraph, without constructing the 2-section, it is possible to check for decomposability in $O(n+m)$ time where m is the number of hyperedges in the hypergraph.

Graham's algorithm

Before presenting this linear algorithm, it is useful to consider a less efficient, but simpler way of checking for decomposability: *Graham's algorithm* [8]. Graham's algorithm carries out the following two operations until neither applies:

1. Delete a vertex that it is in only one hyperedge
2. Delete a redundant hyperedge

If all vertices are deleted then we say that Graham's algorithm *succeeds*. Theorem 8 spells out why this matters.

Theorem 8 *A hypergraph $\mathcal{H}$ is decomposable if and only if Graham's algorithm succeeds with $\mathcal{H}$ as input.* □

Proof See [1]. ■

Algorithm 6 provides the `gPy` implementation of Graham's algorithm. The implementation is in 4 stages:

1. The special case of an empty hypergraph is dispensed with.
2. The hypergraph is reduced.
3. For each vertex, its 'cardinality'—the number of hyperedges of which it is a member—is found and stored. The list of sets `sets` is used to map each cardinality to the set of hyperedges with that cardinality.
4. The main loop is performed. As long as there are vertices in only one hyperedge, i.e. as long as `sets[0]` is non-empty, two things are done: all these vertices are first removed and then redundant hyperedges are removed. Note that the hypergraph is always reduced at the top of the loop. It follows that only hyperedges generated by removing vertices can

possibly be redundant. These are collected in the list `new_hs` and each one is checked to see if it is a redundant hyperedge contained in the hypergraph. If so it is removed and the dictionary `cardinality` and the list `sets` are updated appropriately.

If Graham's algorithm succeeds then the resulting hypergraph will be $\{\emptyset\}$. (An exception is if the hypergraph is intially empty in which case the algorithm succeeds and returns $\emptyset$.) This provides a straightforward test for decomposability.

Algorithm 6 Graham's algorithm

```
    def grahams(self):

        if self.is_empty():
            return self

        self.red()

        cardinality = {}
        sets = [set()]
        max = 0
        for vertex, star in self._star.items():
            card = len(star) - 1
            cardinality[vertex] = card
            if card > max:
                for i in range(card-max):
                    sets.append(set())
                max = card
            sets[card].add(vertex)

        while sets[0] != set():
            new_hs = = self.remove_vertices(sets[0])
            sets[0] = set()
            for hyperedge in new_hs:
                if self.is_redundant(hyperedge):
                    self.remove_hyperedge_once(hyperedge)
                    for vertex in hyperedge:
                        card = cardinality[vertex]
                        sets[card].remove(vertex)
                        card -= 1
                        cardinality[vertex] = card
                        sets[card].add(vertex)
        return self
```

As well as testing for decomposability, Graham's algorithm can be used to construct a join forest if its input is decomposable. When a hyperedge is

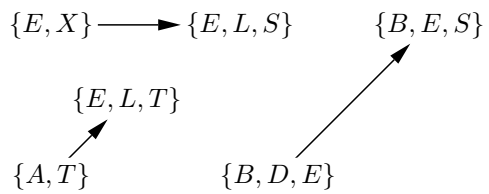

Figure 6.15: Partially construed join tree for $\mathcal{H}_2 = \{\{A,T\},\{E,L,S\},\{B,E,S\},\{E,L,T\},\{E,X\},\{B,D,E\}\}$ after eliminating vertices A, X and D. Edges have been directed to indicate how it was constructing by a run of Graham's algorithm.

removed because it is redundant draw a link between it and the hyperedge which contains it (if there are several which contain it just choose one arbitrarily). If the hypergraph is decomposable then this procedure will produce a join forest for it.

Let's see how Graham's algorithm works on some example hypergraphs. Consider first

$$\mathcal{H}_2 = \{\{A,T\},\{E,L,S\},\{B,S,E\},\{E,L,T\},\{E,X\},\{B,D,E\}\}$$

This is already reduced so after initialising `cardinality` and `sets` the hypergraph is unaltered when the main loop is entered. The vertices A, X and D occur in only one hyperedge ($\{A,T\}$, $\{E,X\}$ and $\{B,D,E\}$ respectively) so they are removed creating the new hyperedges: $\{T\},\{E\},\{B,E\}$. All of these are redundant and so will be removed. It is useful to think of redundant hyperedges being 'absorbed' into containing hyperedges. Assuming we wish to create a join forest a record is kept of which hyperedges absorb them:

$$\begin{array}{lcl} \{A,T\} & \rightarrow & \{E,L,T\} \\ \{E,X\} & \rightarrow & \{E,L,S\} \\ \{B,D,E\} & \rightarrow & \{B,S,E\} \end{array}$$

(For $\{E\}$ there is a choice of absorbing hyperedges.) Note that a record of a link between the *original* hyperedge and the hyperedge which ends up absorbing it is recorded. There is no need to go into the mechanics of how to do this: it is enough to maintain a mapping between each original hyperedge and its 'state' at any point in the algorithm. At this point of the algorithm the join forest is as shown in Fig 6.15.

Having deleted redundant hyperedges the following hypergraph re-enters the top of the loop:

$$\{\{E,L,S\},\{B,E,S\},\{E,L,T\}\}$$

Vertices B and T are now in only one hyperedge. Removing them produces hyperedges $\{S,E\}$ and $\{E,L\}$ both of which are redundant. The following links

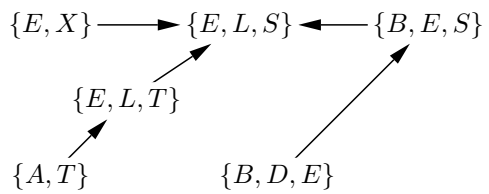

Figure 6.16: A join tree of $\mathcal{H}_2 = \{\{A,T\},\{E,L,S\},\{B,E,S\},\{E,L,T\},\{E,X\},\{B,D,E\}\}$. Edges have been directed to indicate how it was constructing by a run of Graham's algorithm.

can be added to the join forest:

$$\begin{array}{rcl} \{B,E,S\} & \rightarrow & \{E,L,S\} \\ \{E,L,T\} & \rightarrow & \{E,L,S\} \end{array}$$

The hypergraph is now just $\{\{E,L,S\}\}$. E, L and S can now be removed. No further join forest links are created. Graham's algorithm terminates with the hypergraph $\{\emptyset\}$ and the hypergraph is proved decomposable. The join forest is a single join tree and is shown in Fig 6.16. Edges have been drawn to reflect how this run of Graham's algorithm produced it. Formally join trees are undirected, the undirected version of Fig 6.16 is the join tree in Fig 6.13 on page 111.

Consider next this hypergraph: $\{\{1,2,3\},\{2,4\},\{3,4\},\{2,3,4\}\}$ which is used an example in [14]. $\{2,4\}$ and $\{3,4\}$ are redundant producing the following links in the join forest:

$$\begin{array}{rcl} \{2,4\} & \rightarrow & \{2,3,4\} \\ \{3,4\} & \rightarrow & \{2,3,4\} \end{array}$$

and leaving us with the hypergraph $\{\{1,2,3\},\{2,3,4\}\}$. Removing the vertices 1 and 4 produces new hyperedges $\{2,3\}$ and $\{2,3\}$ again (so the hypergraph is no longer *simple* at this point). Clearly one (but not both!) of this pair of repeated hyperedges is redundant and so one can be absorbed into the other. It does not matter which way round this is done; one choice produces the following join forest link:

$$\{2,3,4\} \rightarrow \{1,2,3\}$$

The algorithm terminates by removing vertices from the single surviving hyperedge $\{2,3\}$ and decomposability is established. The corresponding join tree is given in Fig 6.17.

Finally, consider running Graham's algorithm on the non-decomposable hypergraph:

$$\mathcal{H}_1 = \{\{A\},\{S\},\{A,T\},\{L,S\},\{B,S\},\{E,L,T\},\{E,X\},\{B,D,E\}\}$$

Reducing the hypergraph gives

$$\{\{A,T\},\{L,S\},\{B,S\},\{E,L,T\},\{E,X\},\{B,D,E\}\}$$

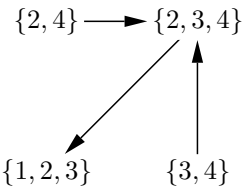

Figure 6.17: A join tree of $\{\{1,2,3\},\{2,4\},\{3,4\},\{2,3,4\}\}$. Edges have been directed to indicate how it was constructing by a run of Graham's algorithm.

Removing A, X and D makes new hyperedges $\{T\}$, $\{E\}$ and $\{B,E\}$. The first two are redundant, removing them leaves:

$$\{\{L,S\},\{B,S\},\{E,L,T\},\{B,E\}\}$$

Now only T can be removed, producing the non-redundant $\{E,L\}$ leaving us with the hypergraph:

$$\{\{L,S\},\{B,S\},\{E,L\},\{B,E\}\}$$

At this point Graham's algorithm stops since each vertex is in more than one hyperedge and there are no redundant hyperedges. Since the hypergraph has not been reduced to a single empty hyperedge the hypergraph is not decomposable.

gPy **Example 28 (Visualising Graham's algorithm)**
There are 4 demos in `gPy.Examples` *showing runs, both successful and unsuccessful, on 4 different hypergraphs. To run the first of them do:*

```
>>> from gPy.Examples import *
>>> grahams_demo()
```

This will generate a window with 3 frames. In the top-left is the hypergraph input to Graham's algorithm, both in its original state and in the state it is currently in at any particular point during the execution of the algorithm. In the top-right is the 2-section of the input hypergraph. `gPy` *makes no effort to place the vertices of this graph nicely, so move them around dragging them with the left mouse button. Since the input hypergraph is both reduced and graphical, the hyperedges are exactly the cliques of this 2-section. At the bottom is a graph whose vertices are the hyperedges and which initially has no edges. Again re-arrange the vertices to be aesthetically pleasing. This graph is a forest (containing 6 single-vertex trees), but it is not yet a* join *forest.*

On standard output will be a message informing us that `VisitAsia` *is about to be eliminated since it exists only in one hyperedge:* {`Tuberculosis`, `VisitAsia`}. *Click on* `Next` *to perform this first step of Graham's algorithm.* `VisitAsia` *is duly eliminated: it is greyed out in the*

2-section and the state of the hyperedge containing it changes and is also greyed out. This is to represent that the hyperedge has gone since it is redundant being contained in {`Cancer`, `TbOrCa`, `Tuberculosis`}. *An undirected edge reflecting this 'absorption' is drawn on the join forest. The edge is undirected since the official representation of a join forest is being used.*

A message about removing **`XRay`** *is now on standard output. Click* **`Next`** *to continue the algorithm and just keep going. Eventually all vertices are eliminated since the input hypergraph is indeed decomposable. Note that the last 3 vertices are removed in one go. Just kill the window to finish the demo.*

The other demos to run are called: `grahams_demo2`, `grahams_demo3` *and* `grahams_demo4`. `grahams_demo2` *has a nondecomposable (albeit graphical) input hypergraph so Graham's algorithm fails.* `grahams_demo3` *uses a decomposable example from [14] and* `grahams_demo4` *a non-decomposable (indeed not even graphical) example from the same paper.*

It is not too difficult to see why the structure which can be produced by Graham's algorithm is a join forest. A link is drawn from a hyperedge when it has 'become' redundant. Since this happens at most once there is at most one arrow from any hyperedge. This suffices to establish that what is constructed is a forest. It remains to establish that the join property obtains. To see this note that (when it succeeds) Graham's algorithm eliminates each vertex in a particular hyperedge. In the case of $\mathcal{H}_2$, whose join tree (directed version) is shown in Fig 6.16, the vertices are eliminated as follows:

A	eliminated in	$\{A, T\}$
B	eliminated in	$\{B, E, S\}$
D	eliminated in	$\{B, D, E\}$
E	eliminated in	$\{E, L, S\}$
L	eliminated in	$\{E, L, S\}$
S	eliminated in	$\{E, L, S\}$
T	eliminated in	$\{E, L, T\}$
X	eliminated in	$\{E, X\}$

If a vertex v is contained in a hyperedge h other than the one in which it is eliminated (call this hyperedge $h(v)$), then h must become redundant at some point and then be absorbed into another hyperedge h'. There is thus an arrow from h to h' in the directed version of the join forest. This h' must also contain v since v is not eliminated in h and h' contains all 'surviving' vertices in h. If $h' \neq h(v)$ then, by the same argument, there is some h'' containing v such that there is an arrow from h' to h'' in the directed version of the join forest. So, from any hyperedge containing a vertex v there is a directed path of hyperedges which, since the graph is finite and a forest, must terminate at $h(v)$ the hyperedge in which v is eliminated. So all hyperedges containing v are connected to $h(v)$

and thus are all connected to each other. By Theorem 6 this proves that the structure constructed by a successful run of Graham's algorithm is a join forest. (Trace the 'trajectory' of the various vertices in Fig 6.16 to convince yourself of this argument.)

Using Graham's algorithm to check for decomposability and build join forests provides a good way to understand what decomposability means for a factored representation of a probability distribution. If the hypergraph associated with a factored representation is decomposable then Graham's algorithm succeeds on it. So all the variables of the distribution can be summed out without creating 'new' intermediate factors: at each point there is always at least one variable which is only contained in one factor, so it can be summed out just by being marginalised away from this factor. The only factor multiplication that is carried out occurs when a redundant factor is 'absorbed' into another.

Maximum cardinality search on a hypergraph

Despite its admirable ability to bring out the properties of decomposable hypergraphs, Graham's algorithm is not the most efficient way of checking for decomposability and building join forests. The basic problem is that redundancy checking leads to a quadratic algorithm. A more efficient approach is to use *maximum cardinality search on a hypergraph* as described by [14]. Graham's algorithm eliminates vertices in a particular order: if the hypergraph is decomposable this order is a zero fill-in on its 2-section. Maximum cardinality search on a hypergraph finds such an ordering, *but in reverse* and also builds any associated join forest also in reverse. The basic idea is simple and has an obvious analogy to maximum cardinality search on a graph:

> Number the vertices n to 1 in decreasing order. As the next vertex to number, select any unnumbered vertex in a[] [hyper]edge of [$\mathcal{H}$] containing as many numbered vertices as possible, breaking ties arbitrarily. [14]

[14] prove that if $\mathcal{H}$ is decomposable then any ordering produced by maximum cardinality search of $\mathcal{H}$ can also be generated by maximum cardinality search of $\mathcal{H}_{[2]}$ and is thus a zero fill-in of $\mathcal{H}_{[2]}$.

Algorithm 7 shows a version of maximum cardinality search on a hypergraph which generates information sufficient to create a join forest if the hypergraph is decomposable, and optionally checks that the hypergraph is decomposable as it progresses. Although it does effectively number vertices from n to 1 it does not bother to record this numbering since (as pointed out by [14]) this is not needed for decomposability checking and join forest construction.

If a hyperedge has had all its vertices numbered call it *exhausted*, otherwise it is *nonexhausted*. A key observation is this: If a nonexhausted hyperedge h is a maximum cardinality hyperedge (i.e. contains as many numbered vertices as possible) and one of these vertices is numbered, then, if it remains nonexhausted, all its remaining unnumbered vertices are eligible for numbering—the number of numbered vertices in h has just gone up by one—so it is still a maximum

cardinality hyperedge. In short the algorithm can be run by selecting maximum cardinality *hyperedges* and numbering all unnumbered vertices in such a hyperedge in one go. (The cardinality of a hyperedge is the number of numbered vertices it contains.) Dictating that the vertex numbering must be done in this way restricts the search so [14] call it *restricted maximum cardinality search on hypergraphs*

In detail, Algorithm 7 operates as follows. Lines 1–2 define the tie breaker function for choosing between hyperedges with maximum cardinality. Lines 3–7 are initialisation. `sets` is such that `sets[i]` contains all hyperedges of cardinality `i` so initially all hyperedges are in `sets[0]`. `cardinality` maps each hyperedge to its cardinality as long as it is still eligible for selection. Initially no hyperedge has been selected and all have cardinality 0; line 4 constructs the appropriate dictionary. `eliminated_in` will map each selected hyperedge to the set of vertices eliminated in it. `eliminated_vertices` is just the set of vertices which have been eliminated. These are the vertices which have been 'numbered', but we record simply whether a vertex has been numbered, not the number itself, so calling such vertices 'eliminated' rather than 'numbered' seems a little more intuitive. `receiver` will provide a mapping between hyperedges which is used to construct a join forest (if that is possible). Lines 8–9 define the internal `ok` function which will be explained later.

Lines 10–13 ensure that the body of the loop is only processed when the last element of `sets`—`sets[-1]`—is non-empty. `sets[-1]` will then contain at least one hyperedge of maximum cardinality. One of these is selected at line 14. All vertices in the selected hyperedge which have not already been eliminated are then eliminated in this hyperedge (lines 15–17). At line 18 a check is (optionally) done to see whether non-decomposability can be established and the functions aborts returning `False` if so. More on this later.

Assuming no early exit has occurred the selected hyperedge is removed from the `cardinality` dictionary to flag that it has been selected (line 21). The main task now is to find each not-already-selected hyperedge containing a vertex which has been just been eliminated (i.e. those in `eliminated_here`) and set it to 'point to' the selected hyperedge using the dictionary `receiver`. This is done by considering each eliminated vertex in turn (line 22). The pointing is effected at line 24. Only hyperedges in the vertex's star are worth looking at; line 23 restricts attention to the intersection of these with the not-already-selected hyperedges. Line 28 checks to see if each such hyperedges has become exhausted, that is, whether all its vertices have been numbered. Leaving aside the decomposability check (of which more later) exhausted hyperedges are removed from the `cardinality` dictionary in order to flag that they are no longer eligible for selection. The rest of lines 25–38 do bookkeeping work to ensure that `sets` and `cardinality` are kept up to date.

Now let's see how maximum cardinality search works using the same example hypergraphs that were used for Graham's algorithm. Consider

$$\mathcal{H}_2 = \{\{A,T\},\{E,L,S\},\{B,E,S\},\{E,L,T\},\{E,X\},\{B,D,E\}\}$$

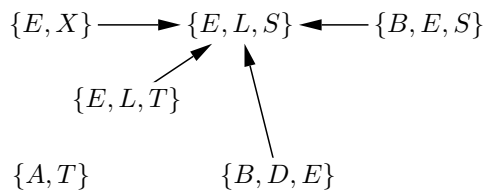

Figure 6.18: Partially constructed join forest corresponding to 1st iteration of maximum cardinality search on the hypergraph $\mathcal{H}_2 = \{\{A,T\},\{E,L,S\},\{B,E,S\},\{E,L,T\},\{E,X\},\{B,D,E\}\}$. (Edges have been directed to reflect how the algorithm works.)

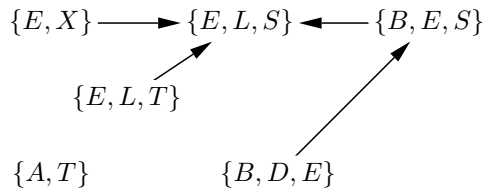

Figure 6.19: Partially constructed join forest corresponding to 2nd and 3rd iteration of maximum cardinality search on the hypergraph $\mathcal{H}_2 = \{\{A,T\},\{E,L,S\},\{B,E,S\},\{E,L,T\},\{E,X\},\{B,D,E\}\}$. (Edges have been directed to reflect how the algorithm works.)

Initially, all hyperedges have cardinality zero since no vertices have been eliminated ('numbered') and thus all are maximum cardinality hyperedges. Suppose that $\{E,L,S\}$ is the first selected hyperedge so we have

$$\begin{array}{lll} E & \text{eliminated in} & \{E,L,S\} \\ L & \text{eliminated in} & \{E,L,S\} \\ S & \text{eliminated in} & \{E,L,S\} \end{array}$$

Hyperedges $\{B,E,S\},\{E,L,T\},\{E,X\},\{B,D,E\}$ all contain just-eliminated vertices so the each is set to 'point to' $\{E,L,S\}$ giving rise to the situation depicted in Fig 6.18

For the next iteration, either $\{B,E,S\}$ or $\{E,L,T\}$ will be selected since they are the only hyperedges having 2 already-eliminated vertices (E and S). Suppose $\{B,E,S\}$ is selected. After its selection we have:

$$B \quad \text{eliminated in} \quad \{B,E,S\}$$

and now $\{B,D,E\}$ must be set to point to $\{B,E,S\}$ since it contains the just-eliminated B.

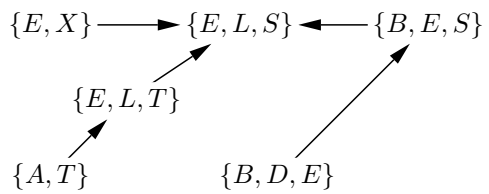

Figure 6.20: Join forest corresponding to 4th and later iterations of maximum cardinality search on the hypergraph $\mathcal{H}_2 = \{\{A,T\},\{E,L,S\},\{B,E,S\},\{E,L,T\},\{E,X\},\{B,D,E\}\}$. This is also the final join forest. (Edges have been directed to reflect how the algorithm works.)

In the 3rd iteration, both $\{E,L,T\}$ and $\{B,D,E\}$ are maximum cardinality hyperedges, both containing 2 already-eliminated vertices. Suppose that $\{B,D,E\}$ is selected, then:

$$D \quad \text{eliminated in} \quad \{B,D,E\}$$

but no pointers need updating since no other hyperedge contains D.

In the 4th iteration $\{E,L,T\}$ will be selected since it is the only hyperedge to have as many as 2 eliminated vertices. This leads to:

$$T \quad \text{eliminated in} \quad \{E,L,T\}$$

and $\{A,T\}$ must be set to point to $\{E,L,T\}$ since it contains T. Fig 6.20 shows the new situation.

The final two iterations will select $\{A,T\}$ and $\{E,X\}$ in some order. No pointers need be updated we just have:

$$\begin{array}{ll} A & \text{eliminated in} \quad \{A,T\} \\ X & \text{eliminated in} \quad \{E,X\} \end{array}$$

It is obvious that maximum cardinality search on hypergraphs always produces a *forest* as represented by the `receiver` dictionary. This dictionary provides a mapping between hyperedges which we visually represent by arrows between hyperedges as in Figs 6.18–6.20. Since each hyperedge is mapped to at most one other hyperedge the resulting structure is a 'directed' forest. To get an (undirected) forest its suffices to replace each directed edge with an undirected one. We now turn to the question of whether this forest is a *join* forest. One easy of deciding this is to say that if it is a join forest then it could have been built by running Graham's algorithm. Recall that Graham's algorithm adds an edge to the join forest if one hyperedge gets absorbed by another. This generally happens due to vertices being eliminated in the absorbed hyperedge (although if the input hypergraph contains redundant hyperedges they can be

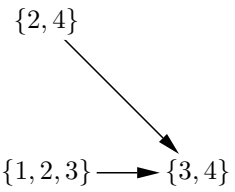

Figure 6.21: Forest for $\mathcal{H}_1^t = \{\{3,4\},\{2,4\},\{1,2,3\}\}$ after 1st iteration of maximum cardinality search.

absorbed right away without any vertex elimination). It follows that we need to check that the vertices contained in a hyperedge which are not eliminated in that hyperedge are all contained in the hyperedge which receives (i.e. 'absorbs') it.

This check can be carried out at the end of the algorithm once the forest has been created, but Algorithm 7 allows the user to do this check *en passant*. Once a hyperedge has become exhausted its receiver (if any) and the vertices eliminated in it (if any) are determined and we can do the check. The check for selected hyperedges occurs at line 18 and for non-selected (but nonetheless exhausted) hyperedges at line 29. Note that the function `ok` which actually does the check has to allow for a hyperedge to have no vertices eliminated in it and also to have no receiver. In either of these cases there will be no entry for the hyperedge in the relevant dictionary and so the rather ugly-looking `get` dictionary method is used to return a value of the emptyset. It is instructive to manually check that all hyperedges in Fig 6.20 pass this check.

To reinforce ideas two more runs of maximum cardinality search will be examined using example hypergraphs from [14]. Consider first

$$\mathcal{H}_1^t = \{\{3,4\},\{2,4\},\{1,2,3\}\}$$

Suppose that $\{3,4\}$ is the first selected hyperedge so that

$$\begin{array}{lll} 3 & \text{eliminated in} & \{3,4\} \\ 4 & \text{eliminated in} & \{3,4\} \end{array}$$

and the associated forest is that in Fig 6.21. Evidently, $\{3,4\}$ passes the decomposability test since all its vertices are eliminated in it.

Next suppose that $\{1,2,3\}$ is selected so that

$$\begin{array}{lll} 1 & \text{eliminated in} & \{1,2,3\} \\ 2 & \text{eliminated in} & \{1,2,3\} \end{array}$$

and the associated forest is that in Fig 6.22.

$\{1,2,3\}$ passes the decomposability test. Now $\{2,4\}$ has now become exhausted (without ever being selected) and so it is ready to be checked. No vertices were eliminated in it, and we see that it contains the vertex 4 which is not present in its receiver $\{1,2,3\}$. This hyperedge hence fails the test and

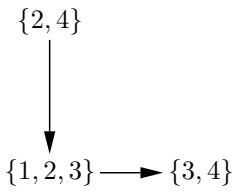

Figure 6.22: Forest for $\mathcal{H}_1^t = \{\{3,4\},\{2,4\},\{1,2,3\}\}$ after 2nd (and final) iteration of maximum cardinality search.

we conclude that the hypergraph $\mathcal{H}_1^t$is not decomposable (in fact, it is not even graphical).

Consider now

$$\mathcal{H}_2^t = \{\{3,4\},\{2,4\},\{1,2,3\},\{2,3,4\}\}$$

which is $\mathcal{H}_1^t$ with the hyperedge $\{2,3,4\}$ added. By selecting $\{1,2,3\}$ and then $\{2,3,4\}\}$ the join forest of Fig 6.17 on page 116 is produced with

1	eliminated in	$\{1,2,3\}$
2	eliminated in	$\{1,2,3\}$
3	eliminated in	$\{1,2,3\}$
4	eliminated in	$\{2,3,4\}$

6.2.5 Finding a decomposable cover for a hypergraph

If the hypergraph associated with a factored representation is decomposable then, as will be shown in Section 6.3, join forest calibration can be used to efficiently compute marginal distributions for all variables. But what to do if the hypergraph is not decomposable? The answer is to construct a different factored representation (of the same distribution) whose associated hypergraph is decomposable. An extreme example of this would be to multiply all factors together to create a factored representation with only a single factor; the associated hypergraph has only one hyperedge containing all variables and is thus decomposable. The problem with this is that calibration would be very inefficient. Basically we want a decomposable hypergraph whose biggest hyperedge is as small as possible.

It is easiest to address this problem by focusing exclusively on the hypergraph and put aside the fact that it represents the structure of some factored representation. Given a hypergraph what we are looking for is a *decomposable cover* for that hypergraph. Definition 46 gives the relevant definition.

Definition 46 A hypergraph $\mathcal{H}_2$ is a *cover* for another hypergraph $\mathcal{H}_1$ if it has the same vertex set and every hyperedge of $\mathcal{H}_1$ is contained in some hyperedge of $\mathcal{H}_2$. A *decomposable cover* is just a cover which is decomposable. □

For any hypergraph we want a decomposable cover whose largest hyperedge is as small as possible. How to find one? Unfortunately, this problem is NP-hard because it is the problem of finding the best variable elimination ordering

for variable elimination which in turn is essentially the same problem as finding a minimal fill-in for a graph. Recall that a hypergraph is decomposable if its reduction is the clique hypergraph of some decomposable graph. So one option for finding a decomposable cover is the following: (1) construct the hypergraph's 2-section, (2) triangulate it using some vertex elimination ordering to compute the fill-in and (3) yank out the cliques of the resulting graph. This clique hypergraph will be a decomposable cover for the original hypergraph. In general, extracting cliques is an expensive business since there can be so many cliques in a graph: up to $3^{n/3}$ for an n-vertex graph. ([15] provide an $O(3^{n/3})$ clique finding algorithm.) However, as noted by [5], finding the cliques of a *triangulated* graph, as done at step (2) is a simpler business. Constructing 2-sections, as in step (1), is also (at worst) quadratic. The real problem is that finding an optimal ordering in step 2 is an NP-hard problem. Since step (2) is the real issue we can view finding an optimal decomposable cover for a hypergraph and finding an optimal triangulation of a graph as interchangeable problems.

Since the problem is NP-hard, it follows that some sort of compromise must be made. If the join forest we are about to produce is to be used many times then it might just be worth the (exponential) time necessary to ensure it is optimal. At the other extreme we could use a fast approach to constructing the join forest, knowing that the resulting forest might well contain a hyperedge considerably bigger than necessary. Generally, a position somewhere between these two extremes is taken: a reasonably fast algorithm is used to produce what is likely to be a near-optimal join forest.

There is a large literature on this problem, mostly focusing on heuristic approaches which are not guaranteed to find optimal join forests. Almost always, the problem is presented in terms of triangulating a graph, rather than working on hypergraphs. Here we will just dip into this literature and address a small number of the issues. (For a good survey see [4].)

Maximum cardinality search

Recall that maximum cardinality search (MCS), whether on graphs or hypergraphs, provides a linear time check for decomposability. If this check shows that the graph/hypergraph is *not* decomposable the question then arises of whether the vertex ordering found by MCS provides a good triangulation/cover nonetheless. Unfortunately, as shown experimentally by [13], MCS (at least with an arbitrary approach to tie-breaking) generally does a poor job. As an example of this phenomenon, consider the non-decomposable graph in Fig 6.23 which is taken from [3]. The MCS ordering clearly generates far more fill-in edges than necessary.

gPy **Example 29 (Elimination ordering from MCS)**
If you need convincing of the problems with MCS just do the following. You will need to re-arrange the vertices in the GUI to look nice.

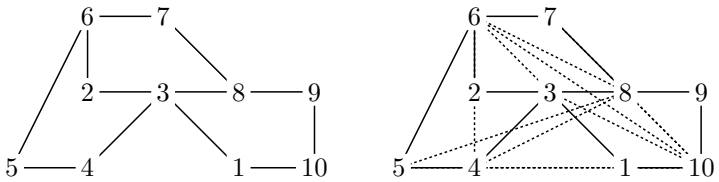

Figure 6.23: Non-decomposable graph with vertices ordered by a maximum cardinality search.

```
>>> from Tkinter import Tk
>>> from gPy.Examples import berryfig6
>>> root=Tk()
>>> berryfig6.gui_display(root)
<gPy.Structures.GraphCanvas instance at 0xb773d72c>
>>> def tmp(s):
...  x=max(s)
...  s.remove(x)
...  return x
...
>>> berryfig6.maximum_cardinality_search(tmp)
({1: 0, 2: 1, 3: 2, 4: 3, 5: 4, 6: 5, 7: 6, 8: 7,
9: 8, 10: 9}, [1, 2, 3, 4, 5, 6, 7, 8, 9, 10])
>>> berryfig6.triangulate(range(1,11))
[(3, 6), (4, 6), (4, 8), (5, 8), (6, 8), (3, 10),
(4, 10), (5, 10), (6, 10), (7, 10), (8, 10)]
>>> berryfig6.gui_display(root)
<gPy.Structures.GraphCanvas instance at 0xb76c192c>
>>>
```

6.3 Calibration

Algorithm 7 Maximum cardinality search on a hypergraph

```
def maximum_cardinality_search(self,
                               choose=None,decomp_check=False):
    if choose is None:                                          # 1
        def choose(s): return s.pop()                           # 2

    sets = [self._hyperedges.copy()]                            # 3
    cardinality = dict(                                         # 4
        zip(self._hyperedges,[0] * len(self._hyperedges)))      # "
    eliminated_in = {}                                          # 5
    eliminated_vertices = set()                                 # 6
    receiver = {}                                               # 7

    def ok(h):                                                  # 8
        return (h - eliminated_in.get(h,frozenset()) <=         # 9
                receiver.get(h,frozenset()))                    # "

    while sets:                                                 # 10
        if not sets[-1]:                                        # 11
            sets.pop()                                          # 12
            continue                                            # 13
        selected_hyperedge = choose(sets[-1])                   # 14
        eliminated_here = (                                     # 15
            selected_hyperedge - eliminated_vertices)           # ""
        eliminated_vertices.update(eliminated_here)             # 16
        eliminated_in[selected_hyperedge] = eliminated_here     # 17
        if decomp_check and not ok(selected_hyperedge):         # 18
            return False                                        # 19
        else:                                                   # 20
            del cardinality[selected_hyperedge]                 # 21
        for vertex in eliminated_here:                          # 22
            for hyperedge in (                                  # 23
                self._star[vertex].intersection(cardinality)):# ""
                receiver[hyperedge] = selected_hyperedge        # 24
                card_h = cardinality[hyperedge]                 # 25
                sets[card_h].remove(hyperedge)                  # 26
                card_h += 1                                     # 27
                if card_h == len(hyperedge):                    # 28
                    if decomp_check and not ok(hyperedge):      # 29
                        return False                            # 30
                    else:                                       # 31
                        del cardinality[hyperedge]              # 32
                else:                                           # 33
                    cardinality[hyperedge] = card_h             # 34
                    try:                                        # 35
                        sets[card_h].add(hyperedge)             # 36
                    except IndexError:                          # 37
                        sets.append(set([hyperedge]))           # 38
    return eliminated_in, receiver                      # 39
```

Chapter 7

Bayesian networks

Chapter 8

Approximate inference

Chapter 9

Parameter estimation

Chapter 10

Introduction to structure learning

List of Symbols

$\mathcal{G}$	Graph
$\alpha, \beta, \ldots$	Vertices of a graph or hypergraph
$\mathcal{H}$	Hypergraph
$\mathcal{H}(\{f_1, f_2, \ldots f_n\})$	Hypergraph of a factored representation
h	Hyperedge
H	Base set of a hypergraph
E_i	Simple event
$A, B, C, \ldots$	Events
Ω	Sample space
$\mathcal{H}_{[2]}$	2-section of hypergraph $\mathcal{H}$
f	Factor
$f(X_1, X_2, \ldots, X_n)$	Factor using variables $X_1, X_2, \ldots X_n$
$f_1 f_2 \ldots f_n$	Product of the factors $f_1, f_2, \ldots f_n$
$\{f_1, f_2, \ldots, f_n\}$	Set of factors; a factored representation
P	Probability distribution
$P(X_1, \ldots, X_n)$	Joint probability distribution over random variables $X_1, \ldots X_n$
$(X_1 = x_1, \ldots, X_n = x_n)$	Joint instantiation; an event
$\mathcal{R}$	Set of real numbers
$\mathcal{R}_{\geq 0}$	Set of non-negative real numbers
Δ	Set of variables
$\Delta(f)$	Set of variables used by factor f
δ	Variable
$\mathcal{I}$	Table
$\mathcal{I}_\delta$	Values of variable δ
$\mathcal{I}(f)$	Table for factor f
i	Cell in a table
Z	Normalising constant; partition function
V	Vertices of a graph
E	Edges of a graph

Bibliography

[1] Catriel Beeri, Ronald Fagin, David Maier, and Mihalis Yannakakis. On the desirability of acyclic database schemes. *Journal of the ACM*, 30(3):479–513, 1983.

[2] Claude Berge. *Graphs and hypergraphs*. North-Holland, Amsterdam, 1973.

[3] Anne Berry, Jean Blair, Pinar Heggernes, and Barry Peytton. Maximum cardinality search for computing minimal triangulations of graphs. *Algorithmica*, 39(4):287–298, 2004.

[4] Hans L. Bodlaender. Treewidth: Characterizations, applications, and computations. Technical Report UU-CS-2006-041, University of Utrecht, 2006.

[5] Robert G. Cowell, A. Philip Dawid, Steffen L. Lauritzen, and David J. Spiegelhalter. *Probabilistic Networks and Expert Systems*. Springer, New York, 1999.

[6] G. A. Dirac. On rigid circuits. *Abhandlungen Mathematisches Seminar Hmaburg*, 25:71–76, 1961.

[7] Pierre Duchet. Hypergraphs. In Ronald Graham, Martin Grötschel, and László Lovász, editors, *Handbook of Combinatorics*, volume 1, pages 381–432. North-Holland, Amsterdam, 1995.

[8] M. H. Graham. On the universal relation. Technical report, University of Toronto, Toronto, Canada, 1979.

[9] Steffen L. Lauritzen. *Graphical Models*. Oxford University Press, Oxford, 1996.

[10] Steffen L. Lauritzen and David J. Spiegelhalter. Local computations with probabilities on graphical structures and their applications to expert systems. *Journal of the Royal Statistical Society A*, 50(2):157–224, 1988.

[11] Andrew Moore and Mary Soon Lee. Cached sufficient statistics for efficient machine learning with large datasets. *Journal of Artificial Intelligence Research*, 8:67–91, 1998.

[12] D. J. Rose. Triangulated graphs and the elimination process. *Journal of Mathematical Analysis and Applications*, 32:597–609, 1970.

[13] Uffe Kjærulff. *Aspects of Efficiency Improvement in Bayesian Networks*. PhD thesis, Dept. of Mathematics and Computer Science, Aalborg University, 1993.

[14] Robert E. Tarjan and Mihalis Yannakakis. Simple linear-time algorithms to test chordality of graphs, test acyclicity of hypergraphs, and selectively reduce acyclic hypergraphs. *SIAM Journal of Computing*, 13(3):566–579, August 1984.

[15] Etsuji Tomita, Akira Tanaka, and Haruhisa Takahashi. The worst-case time complexity for generating all maximal cliques and computational experiments. *Theoretical Computer Science*, 363(1):28–42, 2006.

Index