A Python implementation of graphical models

by
Almero Gouws

Thesis presented in partial fulfilment of the requirements for the degree of Master of Science in Engineering at Stellenbosch University

Supervisor: Professor Ben Herbst
Department of Applied Mathematics, Faculty of Engineering

March 2010

Declaration

By submitting this thesis electronically, I declare that the entirety of the work contained herein is my own, original work, that I am the owner of the copyright thereof (unless to the extent explicitly otherwise stated) and that I have not previously in its entirety or in part submitted it for obtaining any qualification.

March 2010

Abstract

In this thesis we present GrMPy, a library of classes and functions implemented in Python, designed for implementing graphical models. GrMPy supports both undirected and directed models, exact and approximate probabilistic inference, and parameter estimation from complete and incomplete data. In this thesis we outline the necessary theory required to understand the tools implemented within GrMPy as well as provide pseudo-code algorithms that illustrate how GrMPy is implemented.

Opsomming

In hierdie verhandeling bied ons GrMPy aan,'n biblioteek van klasse en funksies wat Python geimplimenteer word en ontwerp is vir die implimentering van grafiese modelle. GrMPy ondersteun beide gerigte en ongerigte modelle, presiese en benaderde moontlike gevolgtrekkings en parameter skattings van volledige en onvolledige inligting. In hierdie verhandeling beskryf ons die nodige teorie wat benodig word om die hulpmiddels wat binne GrMPy geimplimenteer word te verstaan so wel as die pseudo-kode algoritmes wat illustreer hoe GrMPy geimplimenteer is.

Acknowledgments

I would like to thank my supervisor, Prof. Ben Herbst, for all he did to aid me in my research and guide me in writing this thesis, and also for urging me to continue studying after I completed my undergraduate degree.

I would like to thank Stefan van der Walt for his advice in the design and implementation of the project.

I would like to acknowledge Tiberio Caetano for the insight he gave me on his short trip to South Africa, without his work I would never have fully understood graphical models.

I would like to acknowledge the small community of people who contributed to the project, those who submitted code, those who helped me with the design, and those who supported me in the long hours behind my computer.

I would like to thank Fauve, Bjorn and Jar for offering me a much needed distraction from work for the last two years.

Finally, I would like to thank my family for the support they gave me during my studies.

Contents

Contents **i**

List of Figures **iii**

List of Tables **v**

1 Introduction **1**

1.1 Related software . . . 1

2 Models **3**

2.1 Bayesian Networks . . . 3

2.1.1 Derivation . . . 3

2.1.2 Conditional Probability Distributions . . . 4

2.2 Markov Random Fields . . . 5

2.2.1 Derivation . . . 6

2.2.2 Potentials . . . 7

2.3 Factor Graphs . . . 9

3 Inference **12**

3.1 Moralization . . . 12

3.2 The elimination algorithm . . . 14

3.2.1 Derivation . . . 14

3.2.2 Limitations . . . 17

3.3 Belief propagation . . . 19

3.3.1 Elimination to propagation . . . 19

3.3.2 Trees . . . 21

3.3.3 Belief propagation in factor graphs . . . 23

3.4 The sum-product and max-sum algorithms . . . 25

3.4.1 Sum-product derivation . . . 26

3.4.2 Max-sum derivation . . . 28

3.5 The junction tree algorithm . . . 34

3.5.1 Derivation . . . 34

3.5.2 Max-sum in junction trees . . . 41

3.6 Loopy belief propagation . . . 43

3.6.1 Derivation . . . 44

4 Learning **49**
4.1 Maximum-likelihood-estimation . . . 49
4.2 Expectation-maximization . . . 51

5 Open-source software **57**
5.1 Overview . . . 57
5.2 New features . . . 57
5.2.1 Gaussian conditional probability distributions . . . 57
5.2.2 Dynamic Bayesian networks . . . 57

6 Implementation **59**
6.1 The MODEL class . . . 60
6.2 The BNET class . . . 62
6.2.1 Attributes . . . 62
6.2.2 Methods . . . 63
6.3 The MRF class . . . 69
6.3.1 Attributes . . . 69
6.3.2 Methods . . . 70
6.4 The CPD class . . . 75
6.4.1 Attributes . . . 75
6.4.2 Methods . . . 76
6.5 The CLIQUE class . . . 79
6.5.1 Attributes . . . 79
6.5.2 Methods . . . 80
6.6 The POTENTIAL class . . . 84
6.6.1 Attributes . . . 84
6.6.2 Methods . . . 85
6.7 The INF_ENGINE class . . . 91
6.8 The JTREE class . . . 91
6.8.1 Attributes . . . 91
6.8.2 Methods . . . 92
6.9 The LOOPY class . . . 104
6.9.1 Attributes . . . 104
6.9.2 Methods . . . 104
6.10 Supporting functions . . . 117

7 Applications **121**
7.1 Image denoising . . . 121
7.2 Audio-visual speech recognition . . . 123

8 Future work and conclusion **125**

Bibliography **127**

List of Figures

2.1 An example of a Bayesian network. 4
2.2 A simple Bayesian network with the dimensions of each node's conditional probability distribution shown. 5
2.3 An example of a Markov random field. 6
2.4 An example Markov random field with 2 maximal cliques. 7
2.5 An undirected tree-structured graph. 10
2.6 The factor graph representation of Figure 2.5. 10
2.7 An undirected graph with a loop. 11
2.8 (a) Factor graph for Figure 2.7 with arbitrary clique potentials. (b) Factor graph for Figure 2.7 with clique potentials factorised over maximal cliques. 11

3.1 (a) A directed graph. (b) An undirected graph graph, created by moralizing (a). 13
3.2 The graph from Figure 2.3 after eliminating node x_6. 15
3.3 The graph from Figure 3.2 after eliminating node x_5. 15
3.4 (a) The graph from Figure 3.3 after eliminating the node x_4. (b) The graph from (a) after eliminating the node x_3. (c) The graph from (b) after eliminating the node x_2 16
3.5 The elimination of nodes in calculating $p(x_1)$ 18
3.6 The elimination of nodes in calculating $p(x_2)$ 18
3.7 A chain structured Markov random field 19
3.8 The elimination algorithm in a chain graph. 20
3.9 Belief propagation in a chain graph. 21
3.10 An example of message passing in a tree. The nodes are numbered in the order which they calculated and sent their messages. The edges are labeled with the directions the messages were passed over them. 22
3.11 (a) Depth first search of the graph in Figure 2.5. (b) Reversed depth first search of the graph in Figure 2.5 23
3.12 A tree structured Markov random field. 23
3.13 Belief propagation on Figure 3.12 23
3.14 The factor graph representation of Figure 3.12 24
3.15 (a) A tree structured Markov random field. (b) The corresponding factor graph for the Markov random field shown in (a) 26
3.16 (a) The result of applying the depth-first search algorithm to Figure 3.15(a). (b) The result of reversing the traversal order and direction from (a). 26
3.17 (a) The messages produced to calculate $p(x_3)$. (b) The messages produced to calculate $p(x_2)$. 27

3.18 (a) The forward pass of the sum-product algorithm. (b) The backward pass of the sum-product algorithm. 28
3.19 A simple single-clique Markov random field. 28
3.20 A simple Markov random field. 29
3.21 The operations performed to calculate the messages at different nodes during belief propagation in (a) the sum-product algorithm and (b) the max-sum algorithm. 31
3.22 Each column represents a variable and the states that the variable can realize. The lines linking the states represent the backtracking paths of the two possible MAP configurations. 33
3.23 The triangulated representation of the graph in Figure 2.3. 35
3.24 (a) A graph containing a non-chordal cycle. (b) A triangulated graph. 35
3.25 (a) A clique tree with the junction tree property. (b) A clique tree which does not exhibit the junction tree property. 36
3.26 The clique trees from Figure 3.25 with the separator sets drawn in. 37
3.27 The resulting message passing order after applying the reverse depth first search algorithm to Figure 3.26. The edges are labeled with the directions in which the messages are to be passed as well as the order they will be passed in. 38
3.28 A two clique Markov random field 42
3.29 The junction tree for the Figure 3.28 43
3.30 An example of a lattice structured graph. 44
3.31 (a) A simple lattice graph. (b) The clique graph representation of (a). 44
3.32 The four subtrees extracted from Figure 3.31(b). 45
3.33 A subtree extracted from a clique graph with one clique node and its n separator node neighbours. 46
3.34 The graph Figure 3.31(b) showing the potentials assigned to each node. 47
3.35 One iteration of the loopy belief propagation algorithm on Figure 3.34. The nodes marked gray form the subtree that the sum-product algorithm is being applied to in a specific step. The number of superscripts * a potential has indicates how many times it has been updated. 48

4.1 A simple Bayesian network with two nodes. 49
4.2 A simple Markov random field with two nodes. 51

5.1 A first order Hidden Markov model. 58

7.1 The Markov random field created from a 3x3 image. The shaded nodes indicate the observed nodes. 122
7.2 An uncorrupted image. 122
7.3 (a) Figure 7.2 with 10% of the pixels corrupted. (b) The image from (a) after being denoised. (c) Figure 7.2 with 6% of the pixels corrupted. (d) The image from (c) after being denoised 123

List of Tables

1.1 A comparison of GrMPy with other freely available software packages which also support both directed and undirected graphical models. In the Supported Variables field: C - Continuous and D - Discrete; Inference field: E - Exact and A - Approximate; Platform field: P.I. - Platform independent. 2

2.1 An example of the conditional probability distribution $p(x_1)$. 5
2.2 An example of the conditional probability distribution $p(x_2|x_1)$. 5
2.3 An example of the potential $\psi_1(x_1, x_2)$. 8
2.4 An example of the potential $\psi_2(x_2, x_3)$. 8
2.5 The result of expanding the potential $\psi_1(x_1, x_2)$ into $\psi_1(x_1, x_2, x_3)$. 8
2.6 The result of expanding the potential $\psi_2(x_2, x_3)$ into $\psi_2(x_1, x_2, x_3)$. 9
2.7 The value of $\psi_3(x_1, x_2, x_3)$ obtained by adding the potentials $\psi_1(x_1, x_2, x_3)$ and $\psi_2(x_1, x_2, x_3)$. 9
2.8 The value of $\psi_3^E(x_1, x_2)$ obtained from $\psi_3(x_1, x_2, x_3)$ after observing $x_3 = 0$ 9
2.9 The value of $\psi_3^E(x_1, x_3)$ obtained from $\psi_3(x_1, x_2, x_3)$ after observing $x_2 = 1$ 9

3.1 The joint distribution ψ_1 over variables x_1 and x_2. 29
3.2 The marginal probability distributions $p(x_1)$ and $p(x_2)$. 29
3.3 The probability distributions ψ_1 and ψ_2. 32
3.4 The numerical value of $\mu_{\psi_2 \to x_2}(x_2)$. 32
3.5 The numerical value of $[\psi_1(x_1, x_2) + \mu_{x_2 \to \psi_1}(x_2)]$. 33
3.6 The numerical value of $\mu_{\psi_1 \to x_1}(x_1)$. 33

4.1 An example of a training set for the graph shown in Figure 4.1. 50
4.2 The maximum-likelihood estimate of the conditional probability distribution $p(x_1)$, estimated from the sample data in Table 4.1. 50
4.3 The maximum-likelihood estimate of the conditional probability distribution $p(x_2|x_1)$, estimated from the sample data in Table 4.1. 50
4.4 An example of a partially observed training set for the graph shown in Figure 4.2. 51
4.5 The potential $\psi(x_1, x_2)$ and its expected sufficient statistics $ESS(x_1, x_2)$ initialized for the EM algorithm. 52
4.6 The expected sufficient statistics $ESS(x_1, x_1)$ after incorporating the fully observed samples. 52
4.7 The partially observed potential $\psi^E(x_1|x_2 = 1)$. 52
4.8 The marginal distribution $\psi^M(x_1|x_2 = 1)$. 53
4.9 The resized potential $\psi^R(x_1, x_2)$ which will be added directly to the expected sufficient statistics. 53

4.10 The expected sufficient statistics for the potential $\psi(x_1, x_2)$ once all the samples have been processed. 53
4.11 $\psi(x_1, x_2)$ and $ESS(x_1, x_2)$ after the first iteration of the EM algorithm. 53

6.1 The adjacency matrix for the Bayesian network in Figure 2.1. 63
6.2 The adjacency matrix for the Markov random field in Figure 2.3. 69

Chapter 1 - Introduction

Graphical models are widely used for solving problems relating to machine learning, though there are few freely available software packages for implementing them. We present GrMPy, an open-source library of classes and functions implemented in Python. The aim of this project is to develop a comprehensive library of tools for handling graphical models that is open-source, implemented in a freely available language, and platform independent. GrMPy currently supports discrete directed and undirected graphical models, exact and approximate probabilistic inference and parameter estimation from complete and incomplete data. In this thesis we present the theory behind these features, illustrate how the theory translates to the design of the library, and detail how the algorithms have been implemented.

The GrMPy package, and more information on it, is freely available at http://dip.sun.ac.za/vision/trac-git/agouws-GrMPy.bzr.

1.1 Related software

Before detailing the theory and implementation of GrMPy, we compare its features to those of related software packages. There are several existing software packages for implementing graphical models, though few of them are both freely available and can support both directed and undirected graphs. The most prominent software packages that share these traits with GrMPy are BNT, MIM, WinMine, LibDAI. BNT [Murphy, 2001] was originally written by Dr. K. P. Murphy, and later released as an open-source project, and is the original inspiration for GrMPy. Though the development of BNT was abandoned in October 2007, it is still available at http://www.cs.ubc.ca/ murphyk/-Software/BNT/bnt.html. MIM is an application aimed at modeling data with graphical models, and is freely available at http://www.hypergraph.dk/. WinMine is an application aimed at determining statistical models from data, and is freely available at http://research.microsoft.com/en-us/um/people/dmax/winmine/tooldoc.htm. libDAI [Mooij, 2009] is an open source C++ library that provides implementations of various inference methods for discrete graphical models. A tabular comparison of GrMPy and these packages is shown in Table 1.

As shown in Table 1, GrMPy and LibDAI are the only two packages that are both open-source and written in freely available languages. They are also the only packages that are available on an open-source platform, such as Linux. The main difference between GrMPy and LibDAI is that LibDAI is designed to support only discrete variables, whereas GrMPy currently supports continuous Gaussian nodes, and is designed so that other types of distributions can be added with ease. Since LibDAI has been written in C++, its main advantage over GrMPy is speed. As a comparison, the image denoising application explained in Section 7.1, consisting of 20000 variables and 29800 potentials, executes in 26 minutes using GrMPy and less than a second using LibDAI. As explained in Section 8,

	GrMPy	**BNT**	**MIM**	**WinMine**	**LibDAI**
Source available	Yes	Yes	No	No	Yes
Language	Python	Matlab/C	N/A	N/A	C++
Supported Variables	D/C	D/C	D/C	D/C	D
Platform	P.I.	Windows	Windows	Windows	P.I.
Inference	E/A	E/A	E/A	None	E/A
Parameter Learning	Yes	Yes	Yes	Yes	Yes
Structure Learning	No	Yes	Yes	Yes	No

Table 1.1: A comparison of GrMPy with other freely available software packages which also support both directed and undirected graphical models. In the Supported Variables field: C - Continuous and D - Discrete; Inference field: E - Exact and A - Approximate; Platform field: P.I. - Platform independent.

GrMPy's performance issues are being addressed. Though GrMPy will never match the performance of LibDAI, its support of continuous nodes and the ease in which it can be extended make it a viable choice when it comes to the implementation of graphical models for research purposes.

Chapter 2 - Models

Graphical models [Bishop, 2006] are diagrammatic representations of probability distributions. Other than visually representing many properties of a joint probability distribution, such as the conditional independence statements associated with the distribution, the computationally complex problems of inference and learning can be expressed as graphical manipulations which seamlessly follow the underlying mathematics.

A graphical model is made up of nodes which are connected by either directed or undirected edges. Each node represents a random variable, and the edges represent the probabilistic relationships between the variables. Graphs that are comprised of only directed edges are known as directed graphs or *Bayesian networks*, whilst graphs that are comprised of only undirected edges are known as undirected graphs, or *Markov random fields*. Both types of models are covered in this thesis, as well as another model know as a *factor graph*.

2.1 Bayesian Networks

A Bayesian network (BNET), or directed graph [Bishop, 2006], is a graphical model that contains only directed edges and no directed cycles. The directed nature of Bayesian networks makes them suitable for modeling the dependency relationships between random variables.

2.1.1 Derivation

The purpose of a Bayesian network is to visually represent a set of N variables $\mathbf{x} = \{x_1, ..., x_N\}$, and the conditional independencies between the variables in $\mathbf{x}$. From the conditional independencies implied by the Bayesian network we extract the form of the joint probability distribution over the variables in $\mathbf{x}$. A Bayesian network for a set of N random variables $\mathbf{x} = \{x_1, ..., x_N\}$ consists of two things. Firstly, it contains a set of N nodes, with a one-to-one relationship with the variables in $\mathbf{x}$. Secondly, it contains a set of directed edges between the nodes that encode the conditional independence statements associated with the variables in $\mathbf{x}$. Throughout this thesis, we use the notation x_i to represent both the variable and its corresponding node, and we use the words 'variable' and 'node' interchangeably. Figure 2.1 shows an example of a Bayesian network for the random variables $\{x_1, x_2, x_3, x_4, x_5, x_6\}$.

To illustrate the connection between a Bayesian network and the joint probability distribution it represents, we use the Bayesian network shown in Figure 2.1 as an example. In a Bayesian network a node is conditioned only on its parents, for example the node x_2 in Figure 2.1 is conditioned only on the node x_1. Thus the local conditional probability distribution for the node x_2 is $p(x_2|x_1)$. The details of defining and manipulating conditional probability distributions can be found in Section

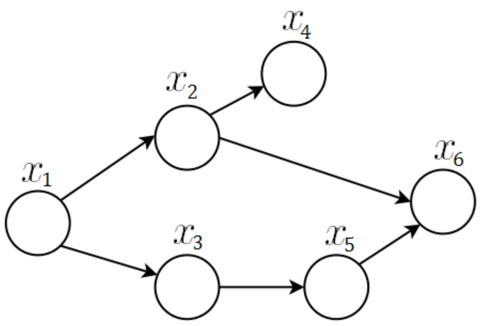

Figure 2.1: An example of a Bayesian network.

2.1.2. In general the local conditional probability distribution of node x_i in a Bayesian network has the form $p(x_i|\pi_i)$, where π_i is the set nodes of parents of the node x_i. The joint probability distribution of any Bayesian network is simply the product of all the local conditional probability distributions associated with the network. Therefore the general factorised equation for a Bayesian network with N nodes is

$$p(\mathbf{x}) = \prod_{i=1}^{N} p(x_i|\pi_i), \tag{2.1}$$

where π_i is the set of parent nodes of the node x_i. As an example, we use (2.1) to determine the joint probability distribution of the Bayesian network shown in Figure 2.1,

$$p(\mathbf{x}) = p(x_1)p(x_2|x_1)p(x_3|x_1)p(x_4|x_2)p(x_5|x_3)p(x_6|x_2, x_5). \tag{2.2}$$

In later chapters we see how the connection between the factorised algebraic representation of the joint probability distribution in (2.1), and the graphical representation of the distribution as a Bayesian network allow us to solve several computationally complex problems efficiently.

2.1.2 Conditional Probability Distributions

In this section we cover the representation of the local conditional probability distributions defining the joint probability distribution of a Bayesian network, given by (2.1). In this thesis we assume all random variables to be discrete.

We represent the conditional probability distribution for a discrete node x_i as a table containing a probability for each value that x_i can take on for every combination of values its parents can take on. Therefore, a node with no parents will have a one-dimensional conditional probability distribution, and a node with two parents will have a three-dimensional conditional probability distribution. The table for each node can be filled with arbitrary non-negative values with the constraint that they sum to one for given fixed values of the parents of the node. This is to ensure that the joint probability distribution obtained by taking the product of the local distributions is properly normalized. Consider the Bayesian network shown in Figure 2.2, where all the nodes can only assume binary values.

The node x_1 in Figure 2.2 has no parents, and is therefore not conditioned on any other nodes. The local conditional probability distribution $p(x_1)$ for this node is a one-dimensional table containing

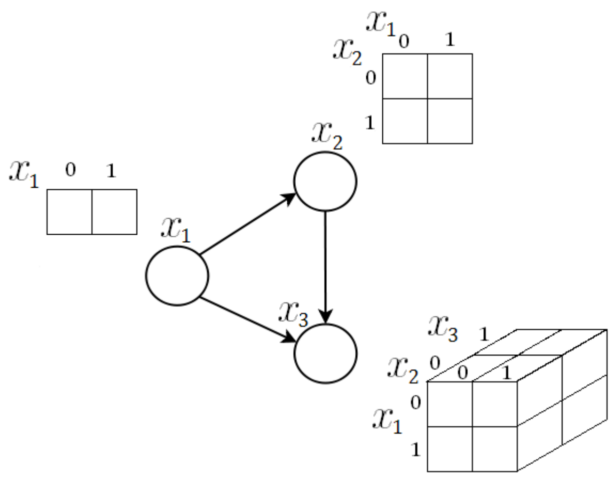

Figure 2.2: A simple Bayesian network with the dimensions of each node's conditional probability distribution shown.

a probability for each possible value that x_1 can take on. An example distribution for $p(x_1)$ is shown in Table 2.1.

$x_1 = 0$	$x_1 = 1$
0.4	0.6

Table 2.1: An example of the conditional probability distribution $p(x_1)$.

The node x_2 has one parent, the node x_1. The local conditional probability distribution $p(x_2|x_1)$ is a two-dimensional table consisting of a probability for each possible value of x_2, given each possible value of x_1. An example distribution of $p(x_2|x_1)$ is shown in Table 2.2.

	$x_1 = 0$	$x_1 = 1$
$x_2 = 0$	0.7	0.6
$x_2 = 1$	0.3	0.4

Table 2.2: An example of the conditional probability distribution $p(x_2|x_1)$.

To apply (2.1), we need to be able to multiply conditional probability distributions together. In this thesis however we never work directly with these distributions. Instead we convert these distributions into the potential distributions as described in Section 2.2.2 before manipulating them in any way. For now, we simply need to know how to construct a lookup table for any given node in a Bayesian network.

2.2 Markov Random Fields

A Markov random field (MRF), or undirected graph [Bishop, 2006], is a graphical model which contains only undirected edges. The family of distributions that can be represented by a Markov random field overlap with the family distributions that can be represented by a Bayesian network. This means there are distributions that can be represented by either of the models, or by only one of the models.

2.2.1 Derivation

As with Bayesian networks, the purpose of a Markov random field is to visually represent a set of N variables $\mathbf{x} = \{x_1, ..., x_N\}$, and the conditional independencies between the variables in $\mathbf{x}$. A Markov random field for a set of N random variables, $\mathbf{x} = \{x_1, ..., x_N\}$, consists of N nodes, with a one-to-one correspondence with the variables in $\mathbf{x}$, and a set of undirected edges connecting the nodes. Figure 2.3 shows a Markov random field for the random variables $\{x_1, x_2, x_3, x_4, x_5, x_6\}$.

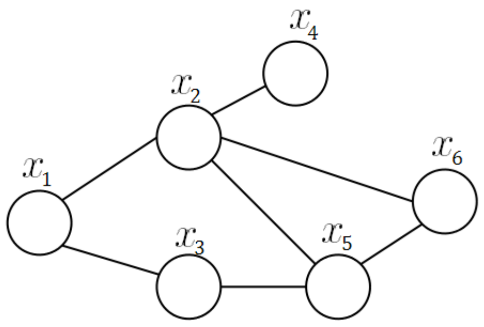

Figure 2.3: An example of a Markov random field.

Since there are no directed edges in a Markov random field, there are no parent and child nodes, so we can not define local conditional probability distributions over nodes and their parents. Instead we define functions of the variables over the *cliques* in the graph. A clique is a set of nodes in a graph that are fully connected, meaning that every node in the set is connected via an edge to every other node in the set. For example, the nodes x_2, x_5 and x_6 in Figure 2.3 form a clique. These three nodes also form *maximal clique* in the graph. A maximal clique is a clique to which no other nodes can be added without it no longer being a clique. For example, the nodes x_2 and x_5 in Figure 2.3 form a clique, but the clique is not maximal because we still add the node x_6 to it. The nodes x_2, x_5 and x_6 form a maximal clique because adding any other node to the clique stops it from being a clique, since the set is no longer fully connected.

Grouping the nodes in a graph into maximal cliques allows us to factorize the joint probability distribution into functions over those maximal cliques, and these functions are known as *potentials*. A potential is a non-negative function defined over a clique and it represents the probability distribution between the nodes in the clique. Note that potentials are not required to be normalized. The details of defining and manipulating potentials are covered in Section 2.2.2, but for now we denote the potential over a clique encompassing the nodes in the set $\mathbf{x_s}$ as $\psi(\mathbf{x_s})$.

The joint probability distribution of a Markov random field with a set of maximal cliques $\mathbf{C}$ is simply the normalized product of the potentials over all the cliques in $\mathbf{C}$, and is given by

$$p(\mathbf{x}) = \frac{1}{Z} \prod_{c \in \mathbf{C}} \psi_c(\mathbf{x_c}), \tag{2.3}$$

where $\mathbf{x_c}$ is the set of nodes belonging to the clique indexed by c, and $\frac{1}{Z}$ is the normalization factor where

$$Z = \sum_{\mathbf{x}} \prod_{c \in \mathbf{C}} \psi_c(\mathbf{x_c}). \tag{2.4}$$

As an example, we determine the joint probability distribution of the Markov random field in Figure 2.3 using (2.3) to be

$$p(\mathbf{x}) = \frac{1}{Z} \psi(x_1, x_2) \psi(x_1, x_3) \psi(x_2, x_4) \psi(x_2, x_5, x_6). \tag{2.5}$$

The factorization of the joint probability distribution of a Markov random field is a very powerful tool, as we shall see in Chapter 3.

2.2.2 Potentials

Potentials are non-negative functions defined over the cliques, usually the maximal cliques, in a Markov random field, and are the factors that make up the joint probability distribution of a Markov random field, as shown in (2.3).

We represent a potential $\psi(\mathbf{x_s})$ as a multi-dimensional table, with entries for each possible combination that the nodes in the clique $\mathbf{x_s}$ can take on. The table representing a potential will have one dimension for each node in the clique $\mathbf{x_s}$. Therefore, a potential for a clique containing 2 nodes, will be a two-dimensional table, and a potential for a clique containing 3 nodes, will be a three-dimensional table, and so on. Consider the Markov random field in Figure 2.4, where all the nodes can assume one of two values.

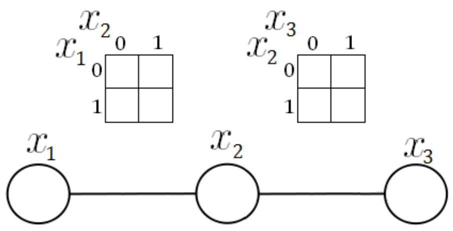

Figure 2.4: An example Markov random field with 2 maximal cliques.

This Markov random field has the two maximal cliques $\{x_1, x_2\}$ and $\{x_2, x_3\}$. The potentials we assign to each of these cliques are $\psi_1(x_1, x_2)$ and $\psi_2(x_2, x_3)$ respectively. Since both potentials involve two variables, each taking on one of two values, both potentials are represented as two-dimensional tables containing an entry for each of the four possible combinations of values the nodes in the potentials can take on. An example of $\psi_1(x_1, x_2)$ is shown in Table 2.3, and an example of $\psi_2(x_2, x_3)$ is shown in Table 2.4. As noted earlier, potentials do not need to be normalized.

Using (2.3) we determine the joint probability distribution of Figure 2.4 to be

$$p(\mathbf{x}) = \frac{1}{Z} \psi(x_1, x_2) \psi(x_2, x_3). \tag{2.6}$$

So, to evaluate (2.3) we need to be able to multiply potential functions together, and in later parts of this thesis we need to add, subtract and divide potentials. The two main problems with manipulating

	$x_2 = 0$	$x_2 = 1$
$x_1 = 0$	0.8	0.1
$x_1 = 1$	0.4	0.9

Table 2.3: An example of the potential $\psi_1(x_1, x_2)$.

	$x_3 = 0$	$x_3 = 1$
$x_2 = 0$	0.7	0.5
$x_2 = 1$	0.5	0.3

Table 2.4: An example of the potential $\psi_2(x_2, x_3)$.

potential functions are that they are often defined over different variables, and often have different dimensions. As an example we find the sum of the two potentials $\psi_1(x_1, x_2)$ and $\psi_2(x_2, x_3)$.

The first step is to identify the dimensions of the resulting potential after adding $\psi_1(x_1, x_2)$ and $\psi_2(x_2, x_3)$. Obviously, the resulting potential will be a function over the variables in both $\psi_1(x_1, x_2)$ and $\psi_2(x_2, x_3)$, therefore we are attempting to find the potential $\psi_3(x_1, x_2, x_3)$ such that

$$\psi_3(x_1, x_2, x_3) = \psi(x_1, x_2) + \psi(x_2, x_3). \tag{2.7}$$

Since $\psi_3(x_1, x_2, x_3)$ involves three variables, it is represented as a three-dimensional table. Therefore, we need to find three-dimensional representations of both $\psi_1(x_1, x_2)$ and $\psi_2(x_2, x_3)$ over all three variables x_1, x_2 and x_3. The values of $\psi_1(x_1, x_2)$ are fixed for whatever value the variable x_3 can realize, therefore we simply make $\psi_1(x_1, x_2)$ into a new potential $\psi_1(x_1, x_2, x_3)$ by adding a third dimension such that

$$\psi_1(x_1, x_2, x_3 = 0) = \psi_1(x_1, x_2, x_3 = 1) = \psi_1(x_1, x_2). \tag{2.8}$$

We apply the same idea to $\psi_2(x_2, x_3)$, since the values of this potential are fixed for all realizations of x_1 can realize. We define the potential $\psi_2(x_1, x_2, x_3)$ such that

$$\psi_2(x_2, x_3, x_1 = 0) = \psi_2(x_2, x_3, x_1 = 1) = \psi_2(x_2, x_3). \tag{2.9}$$

Table 2.5 shows the result of expanding $\psi_1(x_1, x_2)$ into $\psi_1(x_1, x_2, x_3)$ and Table 2.6 shows the result of expanding $\psi_2(x_2, x_3)$ into $\psi_2(x_1, x_2, x_3)$.

$x_3 = 0$	$x_2 = 0$	$x_2 = 1$
$x_1 = 0$	0.8	0.1
$x_1 = 1$	0.4	0.9

$x_3 = 1$	$x_2 = 0$	$x_2 = 1$
$x_1 = 0$	0.8	0.1
$x_1 = 1$	0.4	0.9

Table 2.5: The result of expanding the potential $\psi_1(x_1, x_2)$ into $\psi_1(x_1, x_2, x_3)$.

In order to obtain $\psi_3(x_1, x_2, x_3)$ we simply add $\psi_1(x_1, x_2, x_3)$ and $\psi_2(x_1, x_2, x_3)$ element-wise, the result of which is shown in Table 2.7.

Using $\psi_1(x_1, x_2, x_3)$ and $\psi_2(x_1, x_2, x_3)$, we also element-wise multiply, divide, add and subtract them. The key to evaluating an equation involving potentials is to find representations of all the

$x_3 = 0$	$x_2 = 0$	$x_2 = 1$
$x_1 = 0$	0.7	0.5
$x_1 = 1$	0.7	0.5

$x_3 = 1$	$x_2 = 0$	$x_2 = 1$
$x_1 = 0$	0.5	0.3
$x_1 = 1$	0.5	0.3

Table 2.6: The result of expanding the potential $\psi_2(x_2, x_3)$ into $\psi_2(x_1, x_2, x_3)$.

$x_3 = 0$	$x_2 = 0$	$x_2 = 1$
$x_1 = 0$	1.5	0.6
$x_1 = 1$	1.1	1.4

$x_3 = 1$	$x_2 = 0$	$x_2 = 1$
$x_1 = 0$	1.3	0.4
$x_1 = 1$	0.9	1.2

Table 2.7: The value of $\psi_3(x_1, x_2, x_3)$ obtained by adding the potentials $\psi_1(x_1, x_2, x_3)$ and $\psi_2(x_1, x_2, x_3)$.

potentials that include the variables of all the potentials. Once all the potentials in an equation are converted to the same global representation, we perform all the basic arithmetic functions on them.

Finally, it is necessary to be able to introduce evidence into a potential. As an example, consider the potential $\psi_3(x_1, x_2, x_3)$ shown in Table 2.7. Assuming we observe that $x_3 = 0$, we wish to enter this evidence into the potential $\psi_3(x_1, x_2, x_3)$. We take the appropriate 'slice' of the potential table where $x_3 = 0$. In other words we build a new potential $\psi_3^E(x_1, x_2)$ using only the entries from $\psi_3(x_1, x_2, x_3)$ where $x_3 = 0$. The value of $\psi_3^E(x_1, x_2)$ is shown in Table 2.8. As another example Table 2.9 shows the potential $\psi_3^E(x_1, x_3)$ obtained from the potential $\psi_3(x_1, x_2, x_3)$ after observing $x_2 = 1$.

	$x_2 = 0$	$x_2 = 1$
$x_1 = 0$	1.5	0.6
$x_1 = 1$	1.1	1.4

Table 2.8: The value of $\psi_3^E(x_1, x_2)$ obtained from $\psi_3(x_1, x_2, x_3)$ after observing $x_3 = 0$

	$x_3 = 0$	$x_3 = 1$
$x_1 = 0$	0.6	0.4
$x_1 = 1$	1.4	1.2

Table 2.9: The value of $\psi_3^E(x_1, x_3)$ obtained from $\psi_3(x_1, x_2, x_3)$ after observing $x_2 = 1$

2.3 Factor Graphs

In this section we introduce the concept of factor graphs [Bishop, 2006]. Factor graphs are bipartite graphs, consisting of two different types of nodes, that explicitly represent the factorization of a function. Recall that the joint probability distribution for a Markov random field can be expressed in factorised form, as in (2.3). Therefore $p(\mathbf{x})$ for any graph can be expressed as a product of the factors over subsets of the variables in the graph. Factor graphs make this factorization explicit by introducing additional nodes, known as factor nodes, into the standard graph structure we have used so far. Therefore factor graphs contain both factor nodes and variable nodes. We shall construct a factor graph for the example graph in Figure 2.5.

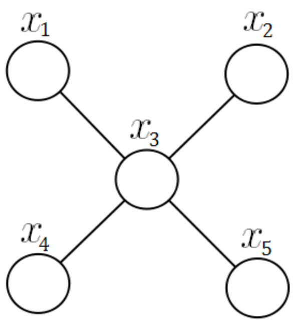

Figure 2.5: An undirected tree-structured graph.

The graph in Figure 2.5 has the following joint probability distribution

$$p(\mathbf{x}) = \frac{1}{Z}\psi_1(x_1, x_3)\psi_2(x_2, x_3)\psi_3(x_3, x_4)\psi_4(x_3, x_5). \tag{2.10}$$

The clique potentials are the factors in the equation, and are therefore represented as factor nodes in the factor graph. To construct a factor graph, we place all the variable nodes, denoted by circles, in the graph as before, as well as a node for each factor, denoted by squares. We then add undirected edges from each factor node to the variable nodes that the factor involves. For instance, the factor $\psi_1(x_1, x_3)$ will be connected to the variable nodes x_1 and x_3. The factor graph representation of Figure 2.5 is shown Figure 2.6.

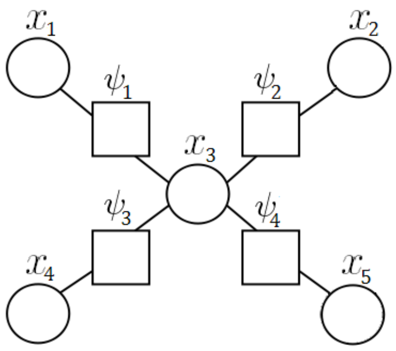

Figure 2.6: The factor graph representation of Figure 2.5.

From this we see that converting an undirected tree into a factor graph results in another tree. However, consider the graph shown in Figure 2.7.

This graph is not a tree, but depending on how we factorize the graph's clique potentials, the corresponding factor graph may be a tree. Two possible factor graphs corresponding to the graph in Figure 2.7 are shown in Figure 2.8. Note that Figure 2.8(a) is not a tree, but Figure 2.8(b) is. This means that by using different factorizations we convert non-tree undirected graphs into tree structured factor graphs.

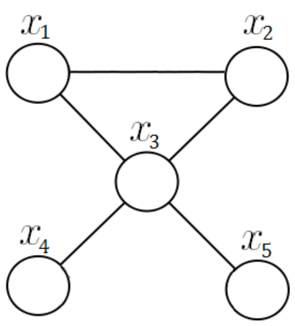

Figure 2.7: An undirected graph with a loop.

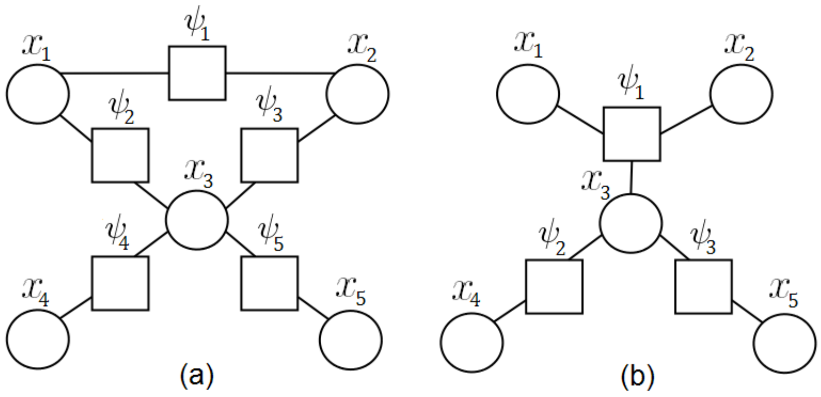

Figure 2.8: (a) Factor graph for Figure 2.7 with arbitrary clique potentials. (b) Factor graph for Figure 2.7 with clique potentials factorised over maximal cliques.

Chapter 3 - Inference

Probabilistic inference is the process of determining the posterior distribution of variables after observing evidence. This chapter covers various algorithms for performing probabilistic inference in graphical models [Bishop, 2006].

3.1 Moralization

Moralization is the process of converting a directed graphical model into an undirected graphical model. This is a useful (and necessary) tool when it comes to performing probabilistic inference on a directed graph. All the inference algorithms covered in this thesis are for undirected models. If we ever encounter a directed model we simply moralize it, which converts it into an undirected graph, and then perform inference.

In a Bayesian network, the joint probability distribution is factorised into several local probability distributions, whereas in a Markov random field, the joint probability distribution is factorised into potentials defined over the cliques in the graph. Therefore, to covert a directed model into an undirected model we need to find a way to convert local probability distributions into clique potentials.

In a directed model there are 3 types of nodes. Firstly there are nodes that have a single parent, such as the nodes x_2, x_3, x_4 and x_5 in Figure 3.1(a). These nodes have the conditional local probability distributions $p(x_2|x_1)$, $p(x_3|x_1)$, $p(x_4|x_2)$ and $p(x_5|x_3)$ respectively, and all of these distributions are two dimensional tables involving two nodes, and all the nodes in each distribution are linked. For example, consider $p(x_2|x_1)$, which is a function of x_1 and x_2. Since the nodes x_1 and x_2 are linked, they form a clique. Therefore, $p(x_2|x_1)$ is already a clique potential, which is a non-negative function over a clique in graph, and this is true for the local probability distribution for any node in directed graph which has only one parent. Therefore, we simply convert the local probability distribution of any node in a directed graph with one parent into a clique potential for an undirected graph, using the equation

$$\psi(x_c, x_p) = p(x_c|x_p). \tag{3.1}$$

The next type of node in a directed graph is a node which has more than one parent, such as node x_6 in Figure 3.1(a). The node x_6 is conditioned on its parents x_2 and x_5, and the local probability distribution for this node is $p(x_6|x_2, x_5)$, which is a 3-dimensional table involving the nodes x_2, x_5 and x_6. Unlike the single-parent nodes, these three nodes do not form a clique, since the parent nodes x_2 and x_5 are not linked. The solution to this problem is simple, we just connect the nodes x_2 and x_5 in the undirected graph, this is called moralization. Now $p(x_6|x_2, x_5)$ is now a non-negative function over a clique in the undirected graph, so

$$\psi(x_c, \mathbf{x}_p) = p(x_c|\mathbf{x}_p). \tag{3.2}$$

where x_c is a child node in the directed graph and $\mathbf{x}_p$ is the set of parent nodes for the node x_c. But what about the extra edges we add in to moralize the directed graph? How do they effect the model? Recall, that a graphical model represents a family of probability distributions, and this family can be defined by the list of conditional independence statements inferred from the graph. In a graphical model, conditional independence statements are implied by the *lack* of edges. Therefore, adding edges decreases the number of conditional independence statements we are making about the variables, and this makes the family of distributions that satisfy the graph larger. So the family of distributions that satisfy a directed graph is a subset of the family of distributions that satisfy its moralized graph, so solving an inference problem for the moralized graph, solves the problem for the original directed graph as well.

The last type of node in a directed graph is a node which has no parents, such as the node x_1 in Figure 3.1(a). This node has the local probability distribution $p(x_1)$, and since $p(x_1)$ is a non-negative function over one node, and one node can be seen as a clique, it qualifies as a clique potential. Therefore,

$$\psi(x_c) = p(x_c). \tag{3.3}$$

Let us now focus on the graph-theoretic process of moralization. Basically, we visit every node in the graph, and if that node has any parents that are not connected by an edge, we connect them. Once we have performed this process at every node we simply drop the directionality of all the edges in the graph. Figure 3.1(a) is an example directed graph and Figure 3.1(b) is the undirected graph created by moralizing Figure 3.1(a).

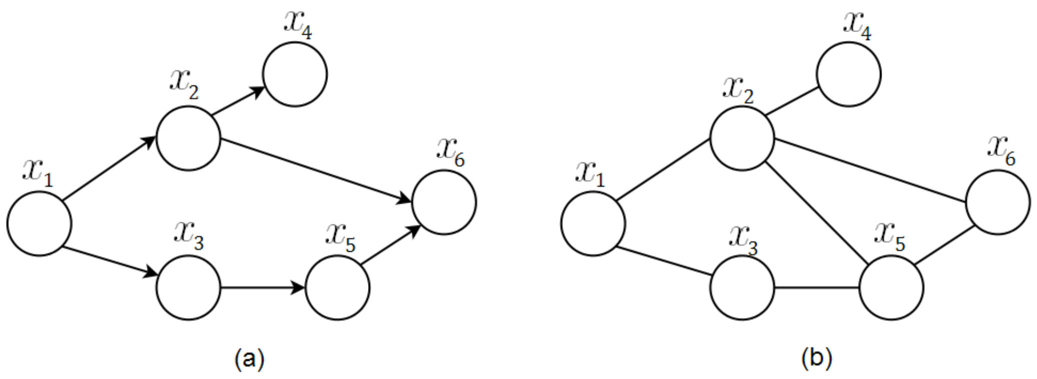

Figure 3.1: (a) A directed graph. (b) An undirected graph graph, created by moralizing (a).

Once we have created the moralized graph for this example, we initialize its clique potentials using the equations (3.1), (3.2) and (3.3). For the rest of this chapter on inference we focus only on undirected models, since any directed model is moralized into a undirected model before inference.

3.2 The elimination algorithm

The elimination algorithm is used to perform exact probabilistic inference on general graphical models. The algorithm is suitable for determining the conditional and marginal probabilities of a single node in a model, and is also the cornerstone in understanding the more advanced inference algorithms.

3.2.1 Derivation

Let us assume that we have a Markov random field with an arbitrary graph structure containing N nodes, as well as defined clique potentials, and we wish to obtain the marginal probability distribution $p(x_n)$ for a single node x_n. We refer to this node as the 'query node'. For simplicity, let us initially assume that all the nodes in the graph are unobserved. We know that the model's joint probability distribution is defined, in factor form, as the product of the potentials defined over the maximal cliques in the graph, given by (2.3). To clarify the connection between the algebra and the graphical representation of the problem, we use the example shown in Figure 2.3. From (2.3) the joint probability distribution for the graph in Figure 2.3 is given by

$$p(\mathbf{x}) = \frac{1}{Z}\psi(x_1, x_2)\psi(x_1, x_3)\psi(x_2, x_4)\psi(x_3, x_5)\psi(x_2, x_5, x_6). \tag{3.4}$$

Say we want to determine $p(x_1, x_2, x_3, x_4, x_5)$, then we simply marginalize (3.4),

$$p(x_1, x_2, x_3, x_4, x_5) = \sum_{x_6} \frac{1}{Z}\psi(x_1, x_2)\psi(x_1, x_3)\psi(x_2, x_4)\psi(x_3, x_5)\psi(x_2, x_5, x_6). \tag{3.5}$$

Assuming that each variable in the graph can take on k values, then the joint probability distribution can be represented by a table with k^6 elements. Therefore, the complexity of summation as it stands is $O(k^6)$, since summing over a table with k^6 elements requires accessing each of those elements. To improve computational efficiency we apply the well known distributive law. Observe that the first four potentials are constants in the summation, as those cliques do not involve the node x_6, therefore, using the distributive law, we 'push' the summation along the product of the potentials until we encounter a potential that does contain the node x_6, yielding the equation

$$p(\mathbf{x}) = \frac{1}{Z}\psi(x_1, x_2)\psi(x_1, x_3)\psi(x_2, x_4)\psi(x_3, x_5)\sum_{x_6}\psi(x_2, x_5, x_6). \tag{3.6}$$

The potential $\psi(x_2, x_5, x_6)$ can be represented by a table with only k^3 entries. Therefore, we have significantly reduced the complexity of the summation to $O(k^3)$ from $O(k^6)$. Let us now determine the marginal probability distribution $p(x_1)$ using this method. The naive attempt at marginalization looks like this

$$p(x_1) = \frac{1}{Z}\sum_{x_2, x_3, x_4, x_5, x_6}\psi(x_1, x_2)\psi(x_1, x_3)\psi(x_2, x_4)\psi(x_3, x_5)\psi(x_2, x_5, x_6). \tag{3.7}$$

By applying the distributive law as we did before, and 'pushing' the summations along the equation, we obtain

$$p(x_1) = \frac{1}{Z}\sum_{x_2}\psi(x_1,x_2)\sum_{x_3}\psi(x_1,x_3)\sum_{x_4}\psi(x_2,x_4)\sum_{x_5}\psi(x_3,x_5)\sum_{x_6}\psi(x_2,x_5,x_6). \quad (3.8)$$

We now evaluate the sums from right to left. Evaluating the left most summation, $m_6(x_2,x_5) = \sum_{x_6}\psi(x_2,x_4,x_6)$, we obtain

$$p(x_1) = \frac{1}{Z}\sum_{x_2}\psi(x_1,x_2)\sum_{x_3}\psi(x_1,x_3)\sum_{x_4}\psi(x_2,x_4)\sum_{x_5}\psi(x_3,x_5)m_6(x_2,x_5), \quad (3.9)$$

where $m_6(x_2,x_5)$ is the intermediate factor left after summing the potential $\psi(x_2,x_5,x_6)$ over x_6. It is obvious that the remaining intermediate factor depends on the nodes x_2 and x_5, after x_6 has been marginalized out of the potential. Note that the node x_6 has been completely eliminated from the equation. As every summation is evaluated, a variable is eliminated from the equation, and we view this as the corresponding node being eliminated from the graph. Therefore we have eliminated the node x_6 from the graph in Figure 2.3, resulting in the graph shown in Figure 3.2.

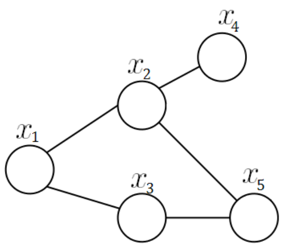

Figure 3.2: The graph from Figure 2.3 after eliminating node x_6.

We continue by evaluating the sum $m_5(x_2,x_3) = \sum_{x_5}\psi(x_3,x_5)m_6(x_2,x_5)$, resulting in

$$p(x_1) = \frac{1}{Z}\sum_{x_2}\psi(x_1,x_2)\sum_{x_3}\psi(x_1,x_3)m_5(x_2,x_3)\sum_{x_4}\psi(x_2,x_4). \quad (3.10)$$

Note that after evaluating the summation over the variable x_5, we are left with the two dimensional intermediate factor $m_5(x_2,x_3)$. In general, this factor does not factorize with respect to x_2 and x_3, which means that the summation has induced a dependency between the variables x_2 and x_3. Since we know that dependencies in graphical models are expressed as edges, graphically eliminating the node x_5 has connected the nodes x_2 and x_3 with an edge, resulting in the graph shown in Figure 3.

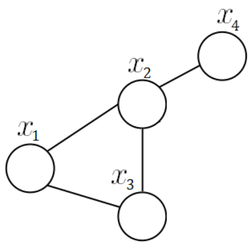

Figure 3.3: The graph from Figure 3.2 after eliminating node x_5.

Continuing with the evaluation of the sums, we get the equations

$$
\begin{aligned}
p(x_1) &= \frac{1}{Z}\sum_{x_2}\psi(x_1,x_2)m_4(x_2)\sum_{x_3}\psi(x_1,x_3)m_5(x_2,x_3), && (3.11)\\
&= \frac{1}{Z}\sum_{x_2}\psi(x_1,x_2)m_4(x_2)m_3(x_1,x_2), && (3.12)\\
&= \frac{1}{Z}m_2(x_1), && (3.13)
\end{aligned}
$$

where (3.11), (3.12) and (3.13) correspond to the elimination of the nodes x_4, x_3 and x_2 respectively, until only the query node, x_1, is left. The corresponding graphical manipulations are shown in Figure 3.4.

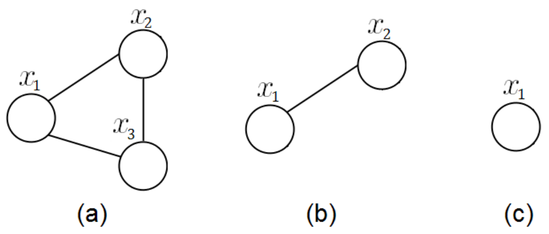

Figure 3.4: (a) The graph from Figure 3.3 after eliminating the node x_4. (b) The graph from (a) after eliminating the node x_3. (c) The graph from (b) after eliminating the node x_2

We now have an algorithm for finding the marginal probability distribution of a single query node in a graph, as long as none of the nodes in the graph are observed.

Since this thesis is focused on the implementation of graphical models, the formal algebraic explanation of the role of conditioning in probabilistic inference is beyond its scope. Instead we present a functional look at the use of observed evidence in calculating conditional probability distributions. Assume we wish to calculate the conditional probability $p(x_1|x_6)$, meaning we have observed the value of the node x_6 and wish to find the posterior distribution of node x_1. Looking at the factorised joint probability distribution of the graph in Figure 2.3, there is only one clique that contains the node x_6. The potential for this clique is 3-dimensional, and to enter the evidence we simply take the appropriate 2-dimensional slice of the potential, as explained in Section 2.2.2, resulting in a new potential which we shall indicate as $\psi^E(x_2,x_5)$, where the superscript E indicates that we have observed evidence for the potential. If we now apply the elimination algorithm as before, we get

$$p(x_1|x_6) = \frac{1}{Z}\sum_{x_2}\psi(x_1,x_2)\sum_{x_3}\psi(x_1,x_3)\sum_{x_4}\psi(x_2,x_4)\sum_{x_5}\psi(x_3,x_5)\psi^E(x_2,x_5). \tag{3.14}$$

Note that the summation over the node x_6 is not present. Since the node x_6 is observed, its value is fixed it can no longer be marginalized. The elimination algorithm now continues as before.

One thing that we have not touched on is the choice of the elimination order. It should be obvious that as long as we choose an elimination order that ends with the query node, the algorithm is applicable, but not necessarily efficient. For example, in calculating $p(x_1)$ for the graph from

Figure 2.3, we used the obvious elimination order 6,5,4,3,2,1. Assume we chose the elimination order 5,6,4,3,2,1. The marginalization equations would be

$$\begin{aligned}
p(x_1) &= \frac{1}{Z}\sum_{x_2}\psi(x_1,x_2)\sum_{x_3}\psi(x_1,x_3)\sum_{x_4}\psi(x_2,x_4)\sum_{x_6}\sum_{x_5}\psi(x_3,x_5)\psi(x_2,x_5,x_6), && (3.15)\\
&= \frac{1}{Z}\sum_{x_2}\psi(x_1,x_2)\sum_{x_3}\psi(x_1,x_3)\sum_{x_4}\psi(x_2,x_4)\sum_{x_6}m_5(x_2,x_3,x_6), && (3.16)\\
&= \frac{1}{Z}\sum_{x_2}\psi(x_1,x_2)m_4(x_2)\sum_{x_3}\psi(x_1,x_3)m_6(x_2,x_3), && (3.17)\\
&= \frac{1}{Z}\sum_{x_2}\psi(x_1,x_2)m_4(x_2)m_3(x_1,x_2), && (3.18)\\
&= \frac{1}{Z}m_2(x_1). && (3.19)\\
& && (3.20)
\end{aligned}$$

Comparing (3.20) with (3.13), we see that even though the elimination order is different, the result is the same. However the computation is more expensive. The first summation in the original elimination order is $\sum_{x_6}\psi(x_2,x_5,x_6)$, and this involved summing over a table with k^3 elements, with a computational complexity of $O(k^3)$. However, using the new elimination order, the first summation is $\sum_{x_5}\psi(x_2,x_5,x_6)\psi(x_3,x_5)$, and to do so we first have to multiply the two potentials together, resulting in a table with k^4 elements, meaning that the summation has a computational complexity of $O(k^4)$. By further comparing the algebra arising from the two different elimination orders, we see that the rest of the corresponding summations in calculating $p(x_1)$ have the same complexities. Choosing a good elimination order can definitely improve the efficiency of the elimination algorithm, however it is beyond the scope of this thesis. For the rest of the thesis, we assume the order to be arbitrary, with the query node as the last node in the order.

3.2.2 Limitations

The elimination algorithm's major drawback is that it is inefficient when executing multiple queries on the same graph. For example, let us compare the calculation of the marginal probability distribution $p(x_1)$ for the graph in Figure 2.3 with the calculation of the marginal probability distribution $p(x_2)$ for the same graph. The elimination of the nodes during the calculation of $p(x_1)$ and $p(x_2)$ are shown in Figures 3.5 and 3.6 respectively.

Note that all the elimination steps (and therefore all the algebraic steps) in calculating $p(x_2)$, except for the final step, are exactly the same as the steps in calculating $p(x_1)$. The only difference being the elimination of the final node. This means that successive calls to the elimination algorithm for different query nodes on the same graph repeats calculations, and it is therefore computationally inefficient for problems where multiple queries are executed, such as finding the marginal probability of every unobserved node in a graph. The belief propagation algorithm presented in Section 3.3 overcomes this limitation on tree-structured graphs, and is extended into the junction tree algorithm for general graphs in Section 3.5.

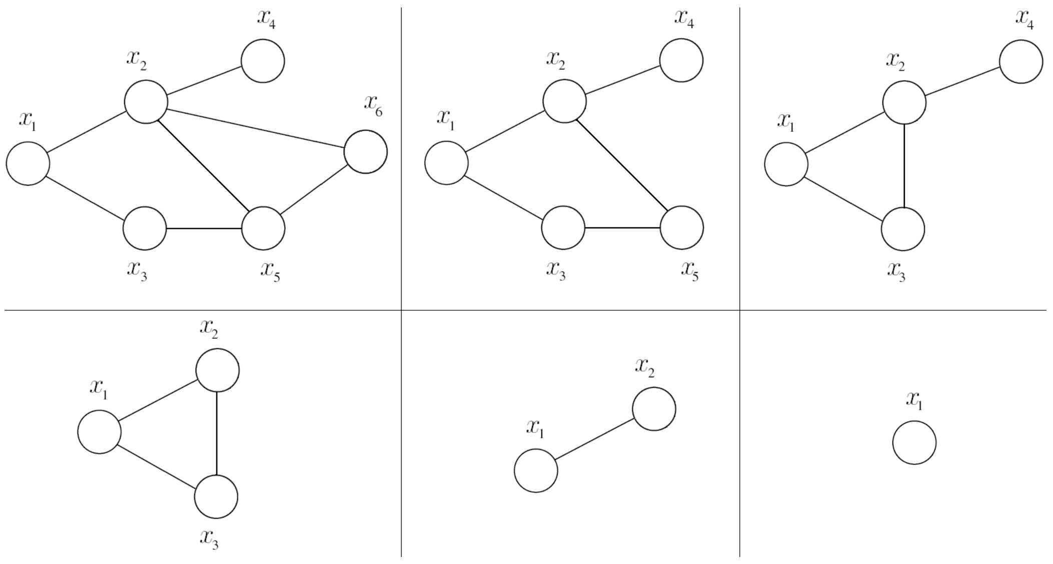

Figure 3.5: The elimination of nodes in calculating $p(x_1)$

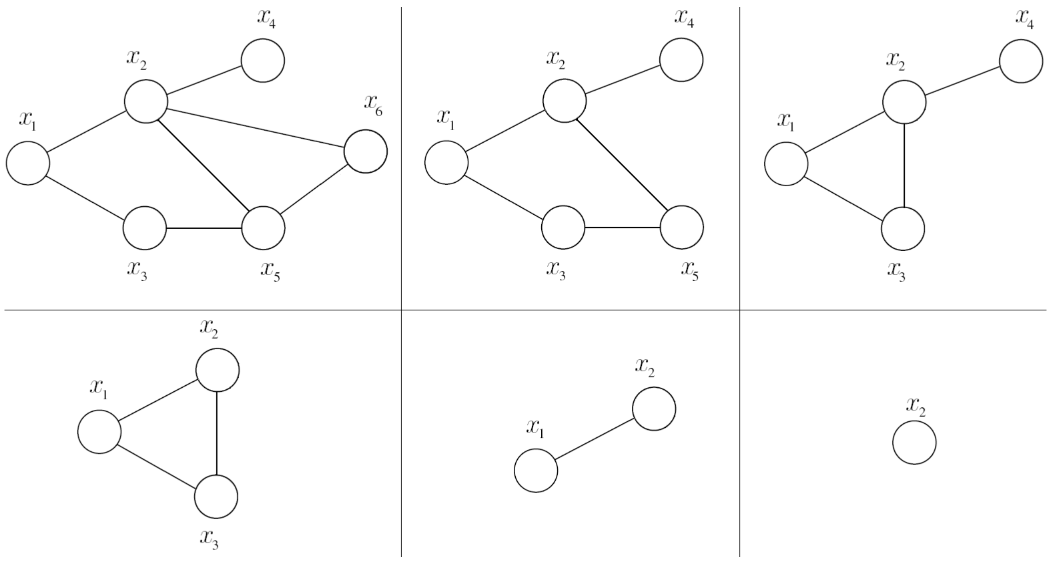

Figure 3.6: The elimination of nodes in calculating $p(x_2)$

3.3 Belief propagation

Belief propagation is used to perform exact probabilistic inference on graphical models that are tree structured or can be represented as tree-structured factor graphs. It is essentially an alternative view of the elimination algorithm, but is also the key to the efficient sum-product algorithm.

3.3.1 Elimination to propagation

To unify the connection between the algebra and the graphical representation of the algorithm we use the chain structured Markov random field shown in Figure 3.7 as an example.

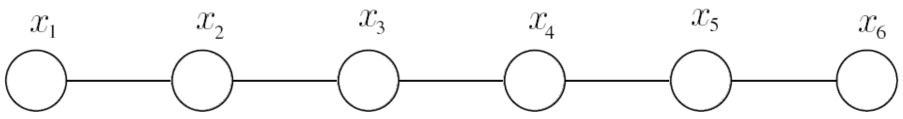

Figure 3.7: A chain structured Markov random field

We know from (2.3) that this graph's joint probability distribution is given by

$$p(\mathbf{x}) = \frac{1}{Z}\psi(x_1,x_2)\psi(x_2,x_3)\psi(x_3,x_4)\psi(x_4,x_5)\psi(x_5,x_6), \tag{3.21}$$

$$= \frac{1}{Z}\sum_{i=1}^{5}\psi(x_i,x_{i+1}), \tag{3.22}$$

where $\frac{1}{Z}$ is the normalization factor. Note that we write the joint probability distribution of a chain of any length in this way. The property we wish to exploit is that every chain has an easily determined optimal elimination order by nature of its structure. For example, if we wanted to calculate $p(x_1)$ using the elimination algorithm, the choice of the elimination order is trivial, being $(6,5,4,3,2,1)$, which would eliminate nodes from the right to the left until only the query node x_1 is left. However, if we now want to calculate the marginal probability distribution $p(x_3)$, what would the elimination order be? We start by simply moving node x_3 to the end of the previous elimination order, resulting in the order $(6,5,4,2,1,3)$. Applying the elimination algorithm using this order results in

$$p(x_3) = \frac{1}{Z}\sum_{x_1,x_2,x_4,x_5,x_6}\psi(x_1,x_2)\psi(x_2,x_3)\psi(x_3,x_4)\psi(x_4,x_5)\psi(x_5,x_6), \tag{3.23}$$

$$= \frac{1}{Z}\sum_{x_1,x_2}\psi(x_1,x_2)\psi(x_2,x_3)\sum_{x_4}\psi(x_3,x_4)\sum_{x_5}\psi(x_4,x_5)\sum_{x_6}\psi(x_5,x_6). \tag{3.24}$$

However, the evaluation of the sum $\sum_{x_2}\psi(x_1,x_2)\psi(x_2,x_3)$ has the computational complexity $O(k^3)$ since it involves three variables. However by simply swapping the nodes x_1 and x_2 in the elimination order, resulting in $(6,5,4,1,2,3)$, we get

$$p(x_3) = \frac{1}{Z}\sum_{x_2}\psi(x_2,x_3)\sum_{x_1}\psi(x_1,x_2)\sum_{x_4}\psi(x_3,x_4)\sum_{x_5}\psi(x_4,x_5)\sum_{x_6}\psi(x_5,x_6), \tag{3.25}$$

where all the summations have the computational complexity $O(k^2)$. By grouping the summations of all the nodes that are to the left of the query node together, and by grouping the summations of all the nodes that are to the right of the query node together, we factorize the equation to get

$$p(x_3) = \frac{1}{Z}\left[\sum_{x_2,x_1}\prod_{i=2}^{1}\psi(x_i,x_{i+1})\right]\left[\sum_{x_4,x_5,x_6}\prod_{i=3}^{5}\psi(x_i,x_{i+1})\right]. \tag{3.26}$$

We view this equation as eliminating nodes from the right and then from the left until we reach the query node x_3. The elimination process, and the consequential intermediate factors that are produced, are shown in Figure 3.8. The shaded node is the node we are calculating the marginal probability for.

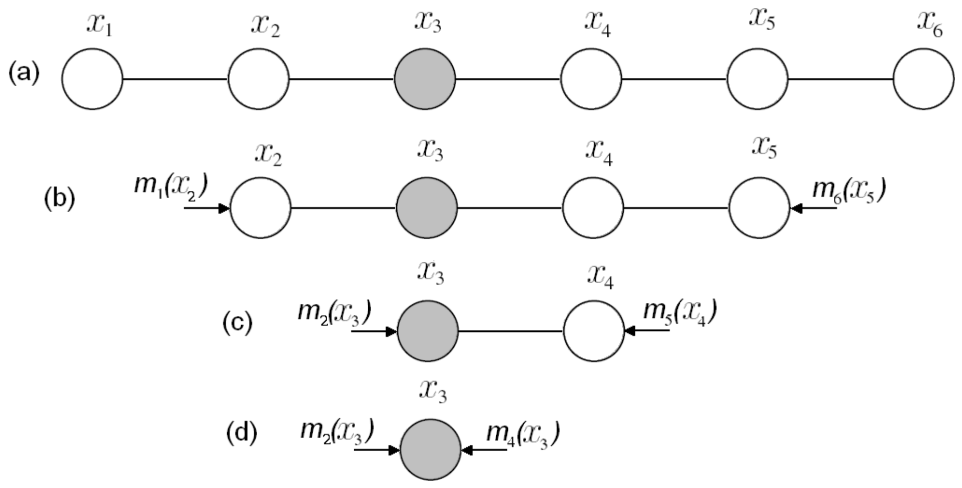

Figure 3.8: The elimination algorithm in a chain graph.

Figure 3.8 shows how eliminating a node in the chain produces the intermediate factor required to eliminate the next node in the chain. An alternative way of viewing these intermediate factors is as local messages, being passed from the leaf nodes along the chain until they reach the query node. Let us introduce some notation by rewriting (3.26) as

$$p(x_3) = \frac{1}{Z}\mu_\alpha(x_3)\mu_\beta(x_3), \tag{3.27}$$

where

$$\mu_\alpha(x_3) = \sum_{x_2,x_1}\prod_{i=2}^{1}\psi(x_i,x_{i+1}) \quad \text{and} \quad \mu_\beta(x_3) = \sum_{x_4,x_5,x_6}\prod_{i=3}^{5}\psi(x_i,x_{i+1}). \tag{3.28}$$

Graphically, $\mu_\alpha(x_3)$ is a message passed forward from node x_2 to node x_3, and $\mu_\beta(x_3)$ is a message passed backward from node x_4 to node x_3. These messages are recursively calculated, for instance, expanding $\mu_\alpha(x_3)$ results in

$$\mu_\alpha(x_3) = \sum_{x_2} \psi(x_2, x_3) \sum_{x_1} \psi(x_1, x_2), \quad (3.29)$$

$$= \sum_{x_2} \psi(x_2, x_3)\mu_\alpha(x_2). \quad (3.30)$$

Based on this, we say that any outgoing message $\mu_\alpha(x_n)$, is calculated by multiplying the incoming message $\mu_\alpha(x_{n-1})$ with the potential involving the nodes x_n and x_{n-1} and then summing the result over x_{n-1}. Since the node x_1 has no incoming forward messages, the first forward message is $\mu_\alpha(x_2) = \sum_{x_1} \psi(x_1, x_2)$. By expanding $\mu_\beta(x_n)$ below, we see that the backward messages are calculated in a similar way.

$$\mu_\beta(x_3) = \sum_{x_4,x_5,x_6} \prod_{i=3}^{5} \psi(x_i, x_{i+1}), \quad (3.31)$$

$$= \sum_{x_4} \psi(x_3, x_4) \sum_{x_5,x_6} \prod_{i=4}^{5} \psi(x_i, x_{i+1}), \quad (3.32)$$

$$= \sum_{x_4} \psi(x_3, x_4)\mu_\beta(x_4). \quad (3.33)$$

Therefore, any outgoing message $\mu_\beta(x_n)$, is calculated by multiplying the incoming message $\mu_\beta(x_{n+1})$ with the potential involving the nodes x_n and x_{n+1} and then summing the result over x_{n+1}. Since the node x_6 has no incoming backward messages, the first backward message is $\mu_\beta(x_5) = \sum_{x_6} \psi(x_5, x_6)$. This process of message passing is called *belief propagation*, and it is illustrated in Figure 3.9.

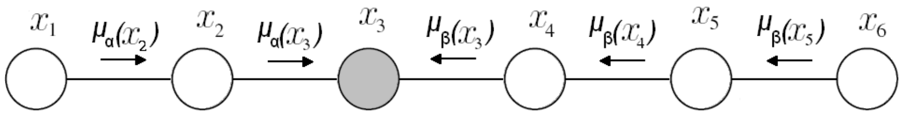

Figure 3.9: Belief propagation in a chain graph.

3.3.2 Trees

In the previous section, we derived the belief propagation algorithm for chain structured graphs. In this section we show that belief propagation can be applied, almost without alteration, to the broader class of graphs known as trees. An undirected graph is classified as a tree when there is only one path between any two nodes, in other words, it contains no loops. An example of a tree structured Markov random field is shown in Figure 2.5.

Assume we want to calculate $p(x_1)$ for the Markov random field shown in Figure 2.5. For a chain graph we would simply propagate the messages in from the ends of the chain, known as the leaf nodes, to the query node x_1, and we do the exact same in this situation, but we need to determine the message passing order. In a chain, a node can only send a message to one of its neighbours, once it has already received a message from its other neighbour. Remember that a chain is a special case of a tree, where each node can have at most two neighbours. We simply need to generalize the message passing rule for a tree, where a node can have more than two neighbours. We say that for an arbitrary tree, a node x_n may only send a message to one of its neighbouring nodes x_m, once it has received a message from each one of its neighbouring nodes other than node x_m. For instance, node x_3 in Figure 2.5 can only send a message to node x_1, once it has received messages from nodes x_2, x_4 and x_5. This process is shown in Figure 3.10.

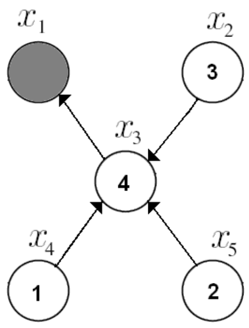

Figure 3.10: An example of message passing in a tree. The nodes are numbered in the order which they calculated and sent their messages. The edges are labeled with the directions the messages were passed over them.

The question is how to determine this passing order easily for an arbitrary tree? The answer is the well know depth-first search algorithm. Applying the depth-first search algorithm to an arbitrary tree, with the query node being used as the root node, has the effect of traversing the entire tree from the query node outwards toward the leaf nodes. For instance, applying the depth-first search algorithm to the tree from Figure 2.5, using node x_1 as the root node results in the traversal shown in Figure 3.11(a). The numbers within the nodes indicate the order in which they were visited, and the arrows on the edges indicate the direction of each traversal. Comparing Figure 3.11(a) with Figure 3.10, we see that the traversals are all in the opposite direction, and that the visiting order is also in reverse. So, by simply reversing the traversal order, and the traversal directions of the result given by the depth first search algorithm, we obtain the required message passing order for the tree, which is shown in Figure 3.11(b).

To confirm our results, consider Figure 3.11(b) and note that every node, other than the root node x_1, has one, and only one, outgoing edge. Now assume that as a node is visited, it calculates and sends its message along its only outgoing edge to its neighbour. For instance, node x_4 is visited first, and it sends a message to node x_3. If we follow this protocol, we see that the messages produced, and the order they are passed in, are exactly the messages required to calculate $p(x_1)$. From this we conclude that belief propagation can also be applied to any tree-structured graph, by simply determining the optimal message passing order via the depth-first search algorithm.

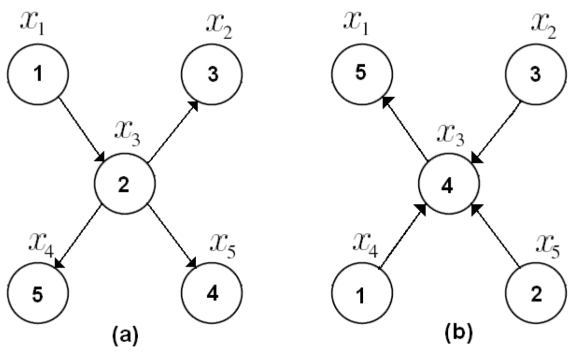

Figure 3.11: (a) Depth first search of the graph in Figure 2.5. (b) Reversed depth first search of the graph in Figure 2.5

3.3.3 Belief propagation in factor graphs

Belief propagation can be applied to any tree structured factor graph, and to derive the process we shall make use of a the graph shown in Figure 3.12 as an example.

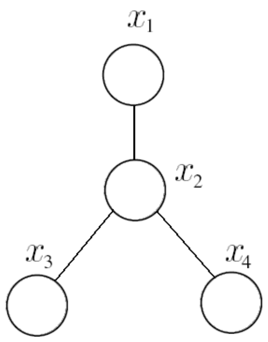

Figure 3.12: A tree structured Markov random field.

Calculating the marginal probability distribution $p(x_1)$ using standard belief propagation on the graph in Figure 3.12, we obtain the message passing shown in Figure 3.13.

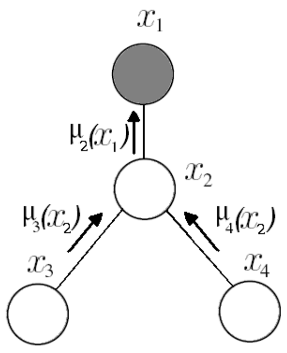

Figure 3.13: Belief propagation on Figure 3.12

The messages are

$$\mu_3(x_2) = \sum_{x_3} \psi_2(x_2, x_3), \tag{3.34}$$

$$\mu_4(x_2) = \sum_{x_4} \psi_3(x_2, x_4), \tag{3.35}$$

$$\mu_2(x_1) = \sum_{x_2} \psi_1(x_1, x_2)\mu_3(x_2)\mu_4(x_2). \tag{3.36}$$

For message passing in a factor graph to be correct, the messages received by the variable nodes need to be identical in both the factor graph belief propagation and standard belief propagation. Let us now attempt to apply belief propagation to the factor graph representation of the graph in Figure 3.12. The graph contains three cliques, and therefore three factors, and the factor graph equivalent is shown in Figure 3.14.

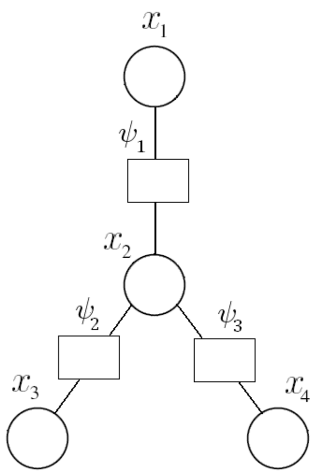

Figure 3.14: The factor graph representation of Figure 3.12

Each of the variable-to-variable messages passed in the standard graph is now broken into two messages in the factor graph, one from a variable node to a factor node, and one from factor node to a variable node. Remember that each of the standard messages involved two operations, multiplication and summation and we separate these operations to form two separate messages. As an example let us consider (3.36), the message from node x_2 to node x_1 in the standard graph. We separate this message into two messages, starting with the message from the variable node x_2 to the factor node ψ_1, denoted as $\mu_{x_2 \to \psi_1}(x_2)$. Since we send a message to the factor, the factor itself cannot exist in the message, and since the factor is not in the message, we cannot perform the marginalization yet either. Therefore, all that is left for the message $\mu_{x_2 \to \psi_1}(x_2)$ is the product of the messages incoming to the node x_2, namely $\mu_3(x_2)$ and $\mu_4(x_2)$, yielding the equation

$$\mu_{x_2 \to \psi_1}(x_2) = \mu_3(x_2)\mu_4(x_2). \tag{3.37}$$

Now we need to determine the message from the factor node ψ_1 to the variable node x_1, after ψ_1 has received all incoming messages, namely $\mu_{x_2 \to \psi_1}(x_2)$, denoted as $\mu_{\psi_1 \to x_1}(x_1)$. We know that it should equal the message received by the variable node x_1 during standard belief propagation, meaning that the factor graph message $\mu_{\psi_1 \to x_1}(x_1)$ must be the standard graph message $\mu_2(x_1)$. Therefore,

$$\begin{aligned} \mu_{\psi_1 \to x_1}(x_1) &= \mu_2(x_1), & (3.38)\\ &= \sum_{x_2} \psi_1(x_1, x_2)\mu_3(x_2)\mu_4(x_2), & (3.39) \end{aligned}$$

and by substituting (3.37) into (3.39) we get

$$\mu_{\psi_1 \to x_1}(x_1) = \sum_{x_2} \psi_1(x_1, x_2)\mu_{\psi_1 \to x_1}(x_1). \tag{3.40}$$

Therefore, to calculate the message from the factor ψ_1 to the node x_1 we calculate the product of the factor and the incoming messages, namely $\mu_{\psi_1 \to x_1}(x_1)$, and then marginalized out all variables other than x_1 from the product, namely x_2. We now write the general equations for the two types of messages. The message from a variable node x_m to a factor node ψ_s is given as

$$\mu_{x_m \to \psi_s}(x_m) = \prod_{r \in ne(x_m) \backslash \psi_s} \mu_{\psi_r \to x_m}(x_m), \tag{3.41}$$

where $r \in ne(x_m) \backslash \psi_s$ is the set of all the neighbours of x_m other than ψ_s, which is the neighbour we are sending the message to. The message from a factor node ψ_s to a variable node x_m is given as

$$\mu_{\psi_s \to x_m}(x_m) = \sum_{\mathbf{x}_s \backslash x_m} \psi_s(\mathbf{x}_s) \prod_{r \in \mathbf{x}_s \backslash x_m} \mu_{x_r \to \psi_s}(x_r), \tag{3.42}$$

where $\mathbf{x}_s \backslash x_m$ is the set of variable nodes connected to the factor node ψ_s, other than x_m, the variable node we are sending the message to. Finally, we need to define the starting messages from the leaf nodes which have only one neighbour. The starting message from a leaf variable node a factor node is

$$\mu_{x \to \psi}(x) = 1, \tag{3.43}$$

and the message from a leaf factor node to a variable node is simply the factor itself,

$$\mu_{\psi \to x}(x) = \psi(x). \tag{3.44}$$

3.4 The sum-product and max-sum algorithms

The sum-product algorithm [Bishop, 2006] is applicable to tree-structured graphs, and is suited for problems where multiple queries will be executed on the same graph, such as finding the marginal probability distribution for every unobserved node in a graph. The max-sum algorithm is a variant of the sum-product algorithm and is used to find the most likely configuration of unobserved nodes in a model.

3.4.1 Sum-product derivation

Assume we are given the tree structured Markov random field in Figure 3.15(a), and we wish to calculate the marginal probability distribution $p(x_3)$. We start by converting the graph to its factor graph representation, shown in Figure 3.15(b), and then apply the belief propagation algorithm for factor graphs.

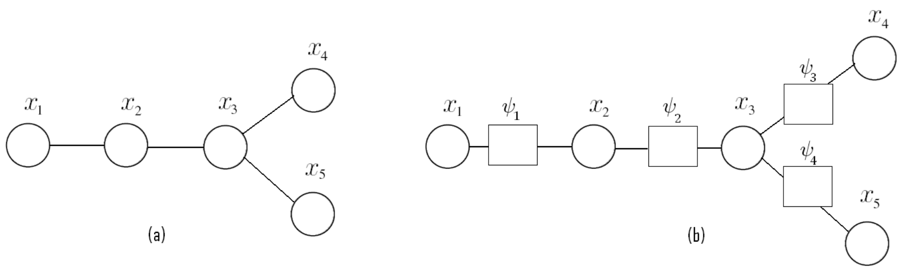

Figure 3.15: (a) A tree structured Markov random field. (b) The corresponding factor graph for the Markov random field shown in (a)

Before performing belief propagation, we need to determine the message passing order. As described in Section 3.3.2, to achieve this we simply apply the depth-first search algorithm to the graph using the query node x_3 as the root node, the result of which is shown in Figure 3.16(a). The nodes in Figure 3.16(a) are numbered in the order they were visited and the edges are marked with the directions in which they were traversed. We then reverse the resulting traversal order and direction to get the message passing order, shown in Figure 3.16(b).

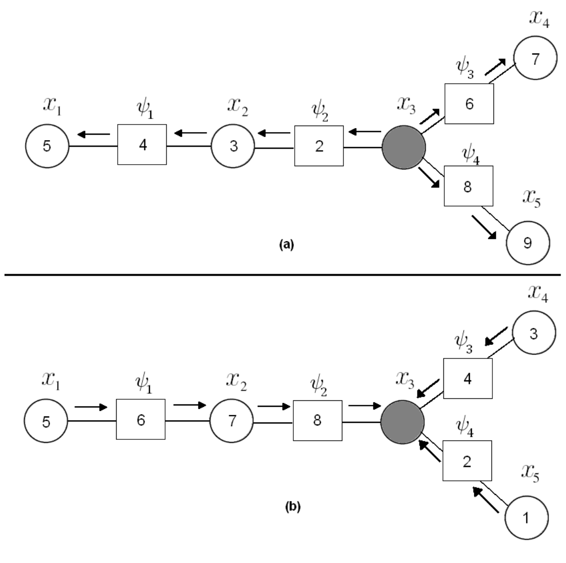

Figure 3.16: (a) The result of applying the depth-first search algorithm to Figure 3.15(a). (b) The result of reversing the traversal order and direction from (a).

In Figure 3.16(b), the nodes are numbered in the order they calculate and send their messages, and the edges are marked with the directions the messages are sent. Using the message passing order, we now perform factor graph belief propagation as described in Section 3.3.3, and the messages produced are illustrated in Figure 3.17(a). If we now wanted to calculate $p(x_2)$ for the same graph, we repeat the process described so far and simply use node x_2 as the query node, and the resulting message passing is shown in Figure 3.17(b).

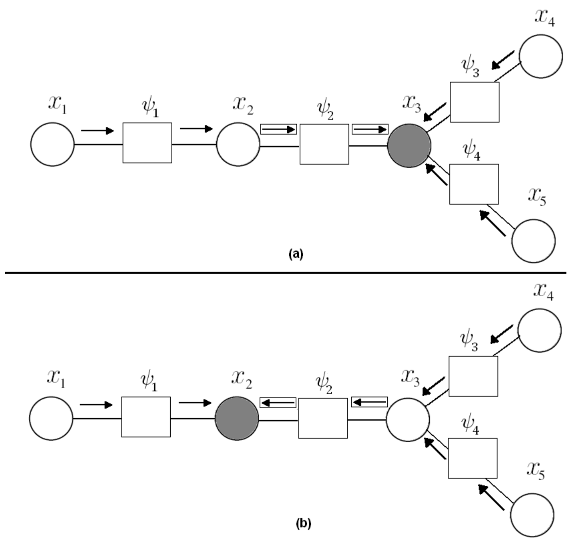

Figure 3.17: (a) The messages produced to calculate $p(x_3)$. (b) The messages produced to calculate $p(x_2)$.

The differences between the two sets of messages are marked with boxes in Figure 3.17, we see that belief propagation repeats calculations in the same way as the elimination algorithm. Remember that to calculate the marginal probability distribution for an arbitrary query node in a tree structured graph, we need to send messages from the leaf nodes toward the query node, so that the query node receives a message from each of its neighbours. During this process, all the non-query nodes in the graph also receive messages propagated from the leaves from all of their neighbours except for one, the one which they sent a message to. For instance, once $p(x_3)$ has been calculated, to calculate $p(x_2)$ we only need the messages $\mu_{x_3 \to \psi_2}$ and $\mu_{\psi_2 \to x_2}$, messages passed in the opposite direction to the ones sent from x_2 and ψ_2 while calculating $p(x_3)$. It stands to reason that if, after calculating $p(x_3)$, we simply propagate backwards from x_3 to the leaf nodes as shown in Figure 3.18(b), each node in the graph receives messages from all its neighbours. This means we are able to calculate the marginal probability distribution $p(x_n)$ for any variable node x_n in the graph by simply multiplying the stored messages from each of its neighbours together. This process is known as the sum-product algorithm, and is illustrated in Figure 3.18.

In Figure 3.18(a), the messages are sent from the leaf nodes to the root node during the *forward pass*. The messages in Figure 3.18(b) are sent from the root node to the leaf nodes during the *backward pass*. Note that the choice of query node is now immaterial, since the use of any query node results in the exact same messages being generated, just in a different order. The sum-product algorithm can be applied to any tree structured graph, or tree structured factor graph. The sum-product algorithm does not repeat calculations, making it significantly more efficient than the elimination algorithm when multiple queries are required in a network. For instance, imagine we are given a chain structured MRF of length N. No matter how many marginal probability distributions we wish

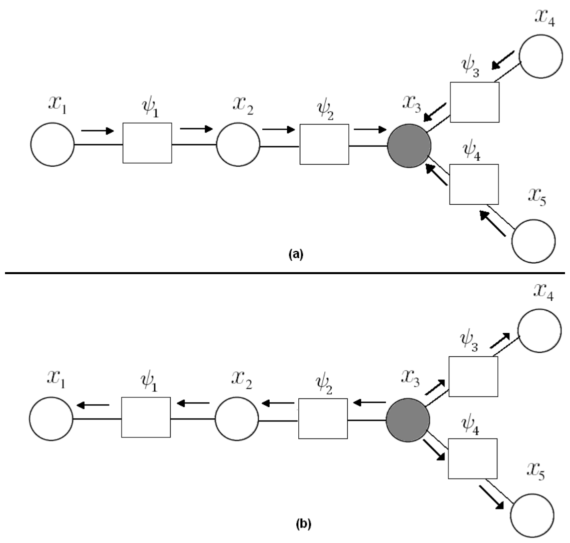

Figure 3.18: (a) The forward pass of the sum-product algorithm. (b) The backward pass of the sum-product algorithm.

to calculate, the sum-product algorithm only needs to produce and store $2(N-1)$ messages, whilst the elimination algorithm produces $(N-1)$ messages for every marginal probability distribution we wish to calculate. Therefore, in the worst case when we wish to calculate the marginal probability distribution for every node in the chain, the sum-product algorithm still only produces $2(N-1)$ messages, whilst the N calls to the elimination algorithm will produce $N(N-1)$ messages. Thus the sum-product algorithm is significantly more efficient when multiple queries are required in a network. The major downfall of the sum-product algorithm is that it can only be applied to graphs that have a tree structure, or an equivalent tree structured factor graph. This limitation is overcome using the junction tree algorithm.

3.4.2 Max-sum derivation

The sum-product algorithm allows us to efficiently determine the marginal probability distribution $p(x)$ for any variable x in an arbitrary tree-structured graph. Another common problem is to find the maximum joint probability of a model and the configuration of all the variables in the model that achieves that probability, known as the maximum-a-posteriori (MAP) configuration. This can be done using a modified version of the sum-product algorithm that incorporates dynamic programming. To begin with, assume we wish to find the MAP configuration of the simple Markov random field shown in Figure 3.19, where the potential ψ_1 has the distribution shown in Table 3.1.

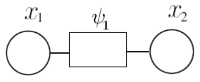

Figure 3.19: A simple single-clique Markov random field.

	$x_2 = 0$	$x_2 = 1$
$x_1 = 0$	0.4	0.6
$x_1 = 1$	0.4	0.1

Table 3.1: The joint distribution ψ_1 over variables x_1 and x_2.

A naive approach would be to execute the sum-product algorithm (trivial in this case), and find $p(x_n)$ for every node x_n, and then set each variable to the value with the highest probability in its marginal probability distribution. The resulting values for $p(x_1)$ and $p(x_2)$ achieved through this approach are shown in Table 3.2.

$p(x_1 = 0)$	$p(x_1 = 1)$
1	0.5

$p(x_2 = 0)$	$p(x_2 = 1)$
0.8	0.7

Table 3.2: The marginal probability distributions $p(x_1)$ and $p(x_2)$.

Based on the information in Table 3.2, we see that the configuration that sets each variable to the state with the maximum probability is $x_1 = 0$ and $x_2 = 0$. However, referring back to Table 3.1, this configuration corresponds to the joint probability of only 0.4, not the maximum value of 0.6 when $x_1 = 0$ and $x_2 = 1$. Therefore, this approach individually maximizes each variable, whereas we seek to find the configuration that maximizes the joint probability of the entire model. In short, we wish to calculate

$$\mathbf{x}^{max} = \underset{\mathbf{x}}{\operatorname{argmax}}\, p(\mathbf{x}), \tag{3.45}$$

and the maximum value of the joint probability distribution is given by

$$p(\mathbf{x}^{max}) = \max_{\mathbf{x}} p(\mathbf{x}). \tag{3.46}$$

Assume we wish to calculate $p(\mathbf{x}^{max})$ for the tree-structured Markov random field shown in Figure 3.20.

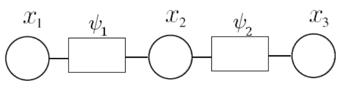

Figure 3.20: A simple Markov random field.

We write can the joint probability distribution of the graph in Figure 3.20 as

$$p(\mathbf{x}) = \frac{1}{Z}\psi_1(x_1, x_2)\psi_2(x_2, x_3), \tag{3.47}$$

and substituting (3.47) into (3.46) we get

$$p(\mathbf{x}^{max}) = \frac{1}{Z}\max_{\mathbf{x}} \psi_1(x_1, x_2)\psi_2(x_2, x_3). \tag{3.48}$$

If we now expand the vector maximization into separate maximizations over each variable we get

$$p(\mathbf{x}^{max}) = \frac{1}{Z} \max_{x_1} \max_{x_2} \max_{x_3} \psi_1(x_1, x_2)\psi_2(x_2, x_3). \tag{3.49}$$

It was at this point during the sum-product algorithm that we used the distributive law to 'push' the summations along the equation to increase computational efficiency, and fortunately the distributive law also applies to the maximization of products, since

$$\max(ab, ac) = a \max(b, c), a > 0. \tag{3.50}$$

Therefore, in the same way we push the summations in the sum-product algorithm, we push the maximizations here. Since the potential $\psi(x_1, x_2)$ is constant when maximizing over the variable x_3, we push the maximization as follows,

$$p(\mathbf{x}^{max}) = \frac{1}{Z} \max_{x_1} \max_{x_2} \psi_1(x_1, x_2) \max_{x_3} \psi_2(x_2, x_3). \tag{3.51}$$

Just like the summations in the sum-product algorithm, we interpret these maximizations as recursively generated messages. However, since we are now calculating the product of small probability values, instead of the product of summations, we may run into numerical underflow problems. An easy way to get around this is to take logarithms of both sides of (3.46),

$$\log(p(\mathbf{x}^{max})) = \max_{\mathbf{x}} \log(p(\mathbf{x})). \tag{3.52}$$

From (3.47) follows

$$\log(p(\mathbf{x})) = \log(\frac{1}{Z}) + \log(\psi_1(x_1, x_2)) + \log(\psi_2(x_2, x_3)). \tag{3.53}$$

Since the distributive law still applies,

$$\max(a + b, a + c) = a + max(b, c), \tag{3.54}$$

the maximizations can still be 'pushed' along the equation to increase computational efficiency, and the result of applying the logarithm and the distributive law to (3.48) is

$$\log(p(\mathbf{x}^{max})) = \log(\frac{1}{Z}) + \max_{x_1} \max_{x_2} \log \psi_1(x_1, x_2) + \max_{x_3} \log \psi_2(x_2, x_3). \tag{3.55}$$

We still view the maximizations as messages, and therefore still perform belief propagation; instead of multiplying messages together at a node, we add them. The difference between belief propagation in the sum-product and max-sum algorithms is shown in Figure 3.21.

We now define the general messages for belief propagation in the max-sum algorithm, starting with the message from a variable node x_m to a factor node ψ_s,

$$\mu_{x_m \to \psi_s}(x_m) = \sum_{r \in ne(x_m) \backslash \psi_s} \mu_{\psi_r \to x_m}(x_m), \tag{3.56}$$

where $r \in ne(x_m)\backslash\psi_s$ is the set of all the neighbours of x_m other than ψ_s, which is the neighbour we are sending the message to. The message from a factor node ψ_s to a variable node x_m is given as

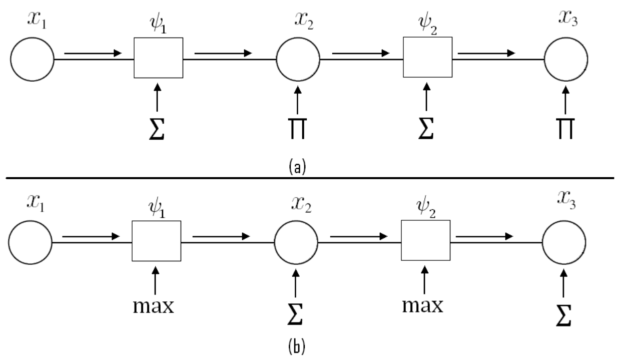

Figure 3.21: The operations performed to calculate the messages at different nodes during belief propagation in (a) the sum-product algorithm and (b) the max-sum algorithm.

$$\mu_{\psi_s \to x_m}(x_m) = \max_{\mathbf{x}_s \backslash x_m} [\log(\psi_s(\mathbf{x}_s)) + \sum_{r \in \mathbf{x}_s \backslash x_m} \mu_{x_r \to \psi_s}(x_r)], \tag{3.57}$$

where $\mathbf{x}_s \backslash x_m$ is the set of variable nodes connected to the factor node ψ_s, other than x_m, the variable node we are sending the message to. Finally, we need to define the starting messages from the leaf nodes, which have only one neighbour. The starting message from a leaf variable node to a factor node is

$$\mu_{x \to \psi}(x) = 0, \tag{3.58}$$

and the message from a leaf factor node to a variable node is simply the log of the factor,

$$\mu_{\psi \to x}(x) = \log(\psi(x)). \tag{3.59}$$

So belief propagation in the max-sum algorithm differs from belief propagation in the sum-product algorithm in the following ways:

1 The logarithm has been applied to all the factors.

2 The summations used to calculate the messages are replaced with maximizations.

3 The products used to calculate the messages are replaced with summations.

There is one more vital difference between the sum-product and max-sum algorithms. In sum-product we propagate messages from the leaf nodes to the root node in the forward pass, and then back from the root node to the leaf nodes in the backward pass. In max-sum we also start with the forward pass, however if we then continue on and run the backward pass, we are not guaranteed to find the MAP configuration. The problem is that there may be several configurations that correspond to the maximum probability of $p(x)$, and it is possible that during the backward pass the resulting MAP values at individual nodes may be set to values from different maximizing configurations, resulting in an overall configuration that does not correspond to the maximum value of $p(x)$. Therefore, we need to make sure all nodes correspond to the same MAP configuration. Once the forward pass is complete, we are able to determine the MAP value of the root node, in this example the node x_1, as shown below

$$x_1{}^{max} = \underset{x_1}{\operatorname{argmax}}\, \mu_{\psi_1 \to x_1}(x_1). \tag{3.60}$$

Once we have set the root node to its MAP value from a certain MAP configuration, we must now set all the other variable nodes to their MAP values from the same configuration. We do this by keeping track, for each variable during the forward pass, the values of its neighbouring variables that gave rise to its maximum state. Then, once we have set the MAP value for the root node we *backtrack* to the leaf nodes, setting them to MAP values from the same maximizing configuration as the root node. Since this is an initially tricky concept, we refer to a fully worked out example to clarify it. Once again, consider the simple chain structured Markov random field shown in Figure 3.20. Let the log values of the potentials be equal to those in Table 3.3.

ψ_1	$x_2 = 0$	$x_2 = 1$
$x_1 = 0$	0.4	0.1
$x_1 = 1$	0.5	0

ψ_2	$x_3 = 0$	$x_3 = 1$
$x_2 = 0$	0.5	0.3
$x_2 = 1$	0.3	0.9

Table 3.3: The probability distributions ψ_1 and ψ_2.

We assign the node x_1 as the root node, and begin propagation from the leaf node x_3. We define an initial message from variable node to a factor node as 0, therefore

$$\mu_{x_3 \to \psi_2}(x_3) = 0. \tag{3.61}$$

Next, we determine the message from the factor ψ_2 to the variable node x_2 using the following equation

$$\begin{aligned} \mu_{\psi_2 \to x_2}(x_2) &= \max_{x_3}[\psi_2(x_2, x_3) + \mu_{x_3 \to \psi_2}(x_3)], & (3.62)\\ &= \max_{x_3} \psi_2(x_2, x_3). & (3.63) \end{aligned}$$

From Table 3.3 we see that (3.63) evaluates to Table 3.4

$x_2 = 0$	$x_2 = 1$
0.5	0.9

Table 3.4: The numerical value of $\mu_{\psi_2 \to x_2}(x_2)$.

Before we continue with the belief propagation, we need to store the values of the node x_3 that corresponds to the value 0.5 if $x_2 = 0$, and 0.9 if $x_2 = 1$. Referring back to ψ_2 in Table 3.3, we see that the value 0.5 arises when $x_2 = 0$ and $x_3 = 0$. Therefore, when we are backtracking from the root, if the node x_2 gets set to the value 0, then the node x_3 must be set to the value 0. This relationship is depicted in Figure 3.22 by the edge marked as (a). Once again referring back to ψ_2 in Table 3.3, we see that the value 0.9 arises when $x_2 = 1$ and $x_3 = 1$, meaning that if the node

x_2 gets set to the value 1 during backtracking, then the node x_3 must be set to the value 1. This relationship is depicted in Figure 3.22 by the edge marked as (b). We now continue with the belief propagation from the variable node x_2. Since node x_2 has only one other neighbour other than the factor node ψ_1, the message it sends to node ψ_1 is equal to the message it received from the factor node ψ_2. Therefore,

$$\mu_{x_2 \to \psi_1}(x_2) = \mu_{\psi_2 \to x_2}(x_2). \tag{3.64}$$

So now we need to determine the final message, from the factor node ψ_1 to the variable node x_1, using the equation

$$\mu_{\psi_1 \to x_1}(x_1) = \max_{x_2}[\psi_1(x_1, x_2) + \mu_{x_2 \to \psi_1}(x_2)]. \tag{3.65}$$

The numerical value of the term $[\psi_1(x_1, x_2) + \mu_{x_2 \to \psi_1}(x_2)]$ is shown in Table 3.5.

	$x_2 = 0$	$x_2 = 1$
$x_1 = 0$	0.9	1
$x_1 = 1$	1	09

Table 3.5: The numerical value of $[\psi_1(x_1, x_2) + \mu_{x_2 \to \psi_1}(x_2)]$.

And by applying the maximization to Table 3.5 we get $\mu_{\psi_1 \to x_1}(x_1)$, shown in Table 3.6.

$x_1 = 0$	$x_1 = 1$
1	1

Table 3.6: The numerical value of $\mu_{\psi_1 \to x_1}(x_1)$.

Once again a maximization is performed, and once again we need to store the backtracking information. From Table 3.5 we see that the value 1.0 when $x_1 = 0$ occurs when $x_2 = 1$, and the value 1.0 for $x_1 = 1$ occurs when $x_2 = 0$. These relationships have been marked in Figure 3.22 as (c) and (d), respectively. From Table 3.6 it is clear that there are two possible MAP configurations, and we use the backtracking information we have stored, shown in Figure 3.22, to ensure that all the variables are set to states from only one of the configurations.

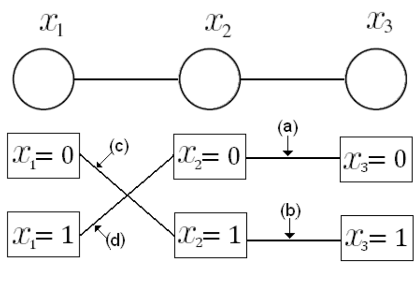

Figure 3.22: Each column represents a variable and the states that the variable can realize. The lines linking the states represent the backtracking paths of the two possible MAP configurations.

For instance, let us choose the state 0 as the MAP value for the root node x_1, and once we have set the MAP value of the root node we perform the backtracking. Referring to Figure 3.22 we see that if $x_1 = 0$, then $x_2 = 1$, and if $x_2 = 1$ then $x_3 = 1$. Similarly, if we assigned $x_1 = 1$, then from Figure 3.22 we see that $x_2 = 0$, and if $x_2 = 0$ then $x_3 = 0$.

The idea of backtracking can be summarized into two main operations. Firstly, whenever we perform a maximization during the forward pass, which is whenever we calculate a message from a factor node ψ_s to a variable node x_m, we need to save the states of the variable nodes connected to the factor node other x_m that give rise to the maximum values. Therefore, whenever we calculate (3.57) we also need to calculate and save

$$\phi_{\psi_s \to x_m}(x_m) = \underset{\mathbf{x}_s \setminus x_m}{\operatorname{argmax}}[\log(\psi_s(\mathbf{x}_s)) + \sum_{r \in \mathbf{x}_s \setminus x_m} \mu_{x_r \to \psi_s}(x_r)]. \tag{3.66}$$

Secondly, once we have performed the forward pass, we set the MAP value of the root node, and using the saved backtracking information from (3.66), we set the values of all the nodes to the same MAP configuration as the root node.

As with the sum-product algorithm, the major downfall of the max-sum algorithm is that it can only be applied to graphs that have a tree structure, or an equivalent tree structured factor graph. However, this limitation can be overcome by using the junction tree algorithm.

3.5 The junction tree algorithm

The junction tree algorithm [Jensen and Jensen, 1994] allows us to perform exact probabilistic inference on general graphical models. Essentially, the junction tree algorithm allows us to find a tree representation of an general graphical model on which we can run the general sum-product and max-sum algorithms.

3.5.1 Derivation

We begin with the elimination algorithm described in Section 3.2. Recall that the elimination algorithm can be applied to any arbitrary graph, and that the elimination of a node from the graph induced a dependency between that node's neighbours, adding an edge to the graph. Let us consider the example shown in Figure 2.3, and the process of running the elimination algorithm on it. Referring back to Figure 3.3, we see that the elimination of the node x_5 induces a dependency between the nodes x_2 and x_3, meaning that an edge is added to connect those nodes. Figure 3.23 shows the original graph from Figure 2.3 with the edge created by the elimination algorithm added in.

The graph in Figure 3.23 has an important property, it is a *triangulated* graph. A graph is triangulated if every cycle involving more than three nodes in the graph contains a chord. A chord is an edge between two nodes in a cycle, but it is not part of the cycle. In essence a chord is a shortcut in the cycle. For example, the graph shown in Figure 3.24(a) is not a triangulated graph because it contains the non-chordal cycle x_1-x_2-x_3-x_4-x_1, while the graph in Figure 3.24(b) is triangulated since it contains the chord x_2-x_4.

The triangulation property is important because *every triangulated graph has a corresponding junction tree*. A junction tree is a clique tree that exhibits the *junction tree property*. Before we explain the concept of the junction tree property, let us explore the concept of the clique tree. A

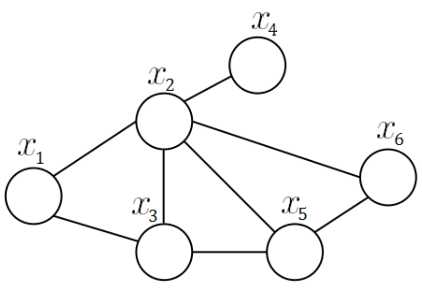

Figure 3.23: The triangulated representation of the graph in Figure 2.3.

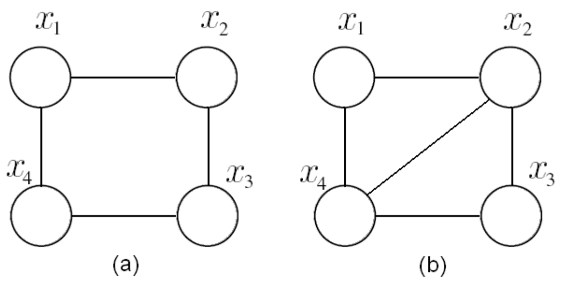

Figure 3.24: (a) A graph containing a non-chordal cycle. (b) A triangulated graph.

clique tree is an undirected graph containing no loops, in which the nodes are the cliques of an underlying graph, and the edges link clique nodes with intersecting cliques. Consider the graph in Figure 3.23. This graph has the following four maximal cliques: $\{x_1, x_2, x_3\}$, $\{x_2, x_3, x_5\}$, $\{x_2, x_5, x_6\}$ and $\{x_2, x_4\}$. Figure 3.25 shows two possible clique trees arising from these cliques, where the clique nodes are denoted as C_i.

Since each node represents a clique from the original graph, they also represent a clique potential from the original graph. Therefore, to calculate the joint probability distribution of a clique tree, we simply multiply together the potentials over each clique node. The joint probability distribution for both clique trees in Figure 3.25 is

$$p(\mathbf{x}) = \frac{1}{Z}\psi_1(x_1, x_2, x_3)\psi_2(x_2, x_3, x_5)\psi_3(x_2, x_4)\psi_4(x_2, x_5, x_6), \tag{3.67}$$

which is also the joint probability distribution for Figure 3.23. From (3.67) we see that the clique tree representation of a standard graph retains the joint probability distribution of that graph. However, this is only a global representation and individual potentials may not correspond to local probabilities. To overcome this problem, we introduce the concept of *separator nodes*. On each edge in the clique tree we add a node S_i that represents the intersection between the clique nodes on either side of the edge. Figure 3.26 shows the clique trees from Figure 3.25 with the separator nodes added in. Note that the separation sets are cliques themselves, and for each separator node we define a potential over its clique. These potentials are known as *separator potentials*, with the separator potential for the separator node S_i denoted as $\phi_i(\mathbf{X}_s)$, where $\mathbf{X}_s$ is the set of nodes in the separator set. If we initialize the separator potentials to be one, the joint probability distribution for a clique tree now becomes

$$p(\mathbf{x}) = \frac{\prod_i \psi_i(\mathbf{X}_i)}{\prod_i \phi_j(\mathbf{x}_j)}. \tag{3.68}$$

The normalizing factor $\frac{1}{Z}$ has been replaced with a separator potential over the empty set that has value of Z. Therefore the value Z appears in the denominator of 3.68. The use of separator potentials allows us to execute the belief propagation algorithms on the junction tree that will ensure that all the potentials become marginal probabilities, while preserving the joint probability distribution.

Now that we have introduced clique trees, let us return to the junction tree property. Consider the clique trees in Figure 3.25. The edges in both these clique trees connect cliques that intersect, and the connections may seem arbitrary at first, but there is one major difference between the two clique trees, and that is that one of them exhibits the junction tree property.

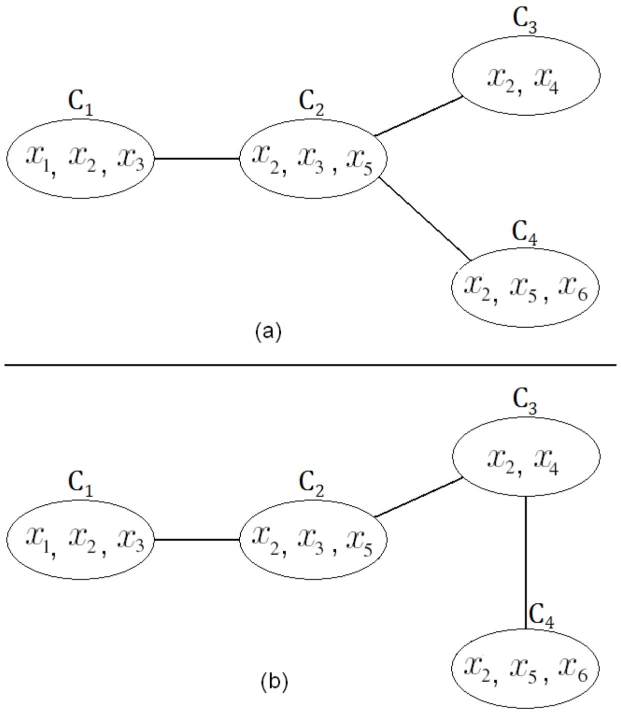

Figure 3.25: (a) A clique tree with the junction tree property. (b) A clique tree which does not exhibit the junction tree property.

A clique tree exhibits the junction tree property when all cliques in the unique path between any two cliques C_i and C_j contain the intersection $C_i \cap C_j$. Consider the node x_5 in Figure 3.25(b), and note that even though it appears in two of the cliques, those cliques are not connected by an edge, and the clique in the path between those two nodes does not contain the node x_5. This means that Figure 3.25(b) cannot be a junction tree. In Figure 3.25(a), the two cliques containing the node x_5 are connected and form a connected subtree in the graph. This is true for every variable in Figure 3.25(a), meaning that this graph is a junction tree.

The importance of the junction tree property is because it allows us to perform belief propagation on the clique tree. This means that if we have a triangulated graph, and then extract a corresponding junction tree for that graph, we can perform the sum-product and max-sum algorithms on the more general junction tree. Up to this point, belief propagation has been limited to graphs that were tree-structured or had tree-structured factor graphs. Before we explain why the junction tree property is necessary for belief propagation in a clique tree, let us consider how belief propagation is performed. Assume that we wish to calculate the marginal $p(x_1)$ for the graph shown in Figure 3.23, and that we have already determined its junction tree to be the one shown in Figure 3.25(a). First, we introduce the junction tree's separator sets, as shown in Figure 3.26(a).

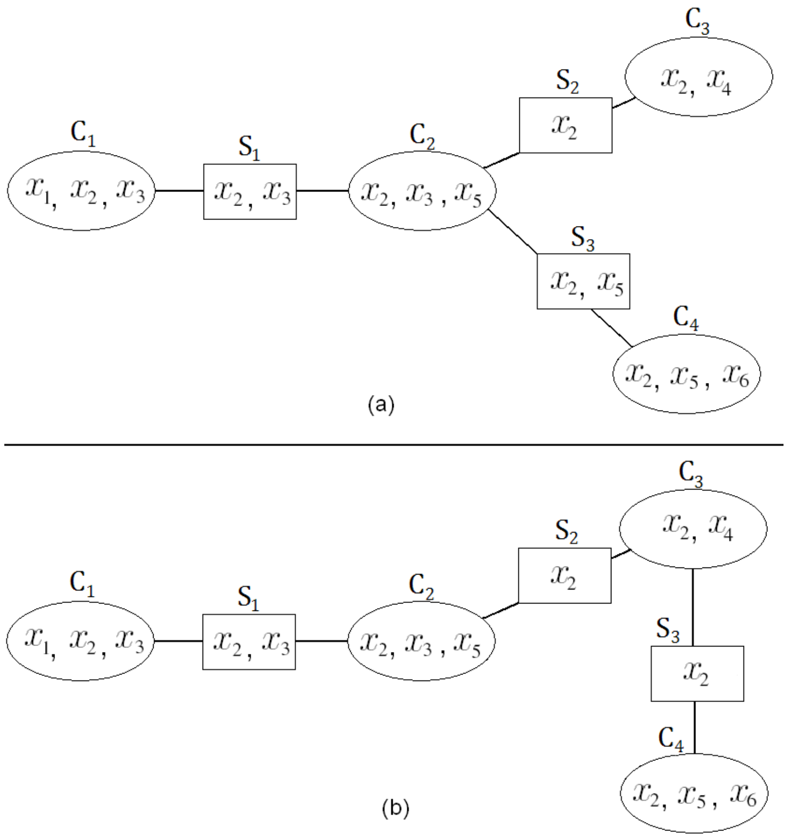

Figure 3.26: The clique trees from Figure 3.25 with the separator sets drawn in.

Note that Figure 3.26(a) has a similar structure to Figure 3.14, which is a standard factor tree. If we view the clique nodes as factor nodes, and the separator nodes as variable nodes, we find that belief propagation in a clique tree is almost identical to belief propagation in a factor tree. Therefore, we follow each step of normal factor graph belief propagation and attempt to apply it to a clique tree. We start by initializing the variable nodes, in this case the separator potentials over the separator nodes, to arrays of ones. We then determine the message passing order via the reverse depth-first search algorithm, using any node as the root node, as shown in Figure 3.27, where we choose clique node C_1 as the root node.

Once this is done, we start propagating messages inwards from the leaf nodes to the root nodes.

The first message we define is the message from a clique node to a separator node, which in a factor graph is the message from a factor node to a variable node given by (3.42). To calculate the message in (3.42), we multiply the factor by all the incoming messages from all its neighbouring variable nodes other than the one we are sending a message to, and then marginalized out all the variables in the

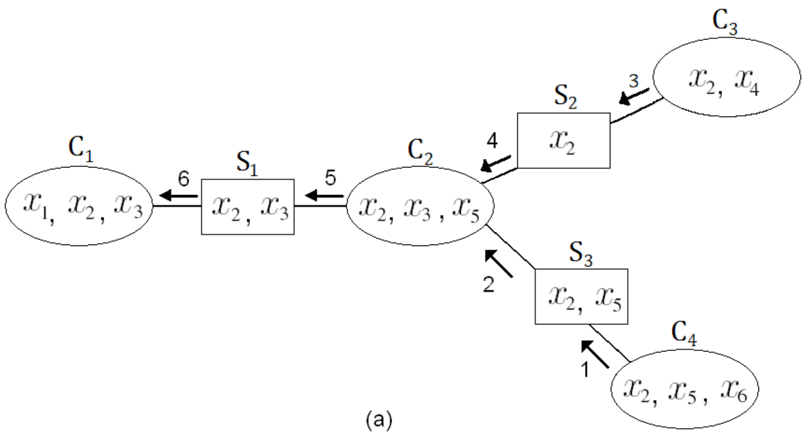

Figure 3.27: The resulting message passing order after applying the reverse depth first search algorithm to Figure 3.26. The edges are labeled with the directions in which the messages are to be passed as well as the order they will be passed in.

factor other than the variable we were sending the message to. In this case, we multiply the clique potential by the incoming messages from all its neighbouring separator nodes, other than the one we are sending the message to, and then marginalize out all the variables in the clique potential other than the ones appearing in the potential of the target separator node. Therefore, the message from a clique node C_s to a separator node S_m is

$$\mu_{C_s \to S_m}(\mathbf{x}_m) = \sum_{\mathbf{x}_s \backslash \mathbf{x}_m} \psi_s(\mathbf{x}_s) \prod_{r \in ne(C_s) \backslash S_m} \mu_{S_r \to C_s}(\mathbf{x}_r), \tag{3.69}$$

where $\mathbf{x}_m$ is the set of variables in the separator potential ϕ_m, $\mathbf{x}_s$ is the set of variables in the clique potential ψ_s, $\mathbf{x}_s \backslash \mathbf{x}_m$ is the set of all variable occurring in $\mathbf{x}_s$ that are not in $\mathbf{x}_m$ and $ne(C_s)\backslash S_m$ is the set of separator nodes that are neighbours of the clique node C_s other than S_m, the separator node we are sending the message to. Recall that to determine the marginal probability of a node after executing the sum-product algorithm we need to store all of the messages produced.

An alternative way of saving this information is to update the potentials during the message passing and simply obtain the marginals from the potentials after the message passing has been completed. Thus, we break (3.69) into two parts, an update and a message. We update the potential ψ_s by multiplying it with the incoming messages, thus the updated value ψ_s^* of the potential ψ_s is given by

$$\psi_s^*(\mathbf{x}_s) = \psi_s(\mathbf{x}_s) \prod_{r \in ne(C_s) \backslash S_m} \mu_{S_r \to C_s}(\mathbf{x}_r), \tag{3.70}$$

From (3.70), the message from an updated clique node C_s^* to a separator node S_m becomes

$$\mu_{C_s^* \to S_m}(\mathbf{x}_m) = \sum_{\mathbf{x}_s \backslash \mathbf{x}_m} \psi_s^*(\mathbf{x}_s). \tag{3.71}$$

Now we need to define the message from a separator node to a clique node, which in a factor tree is the message from a variable node to a factor node, given by (3.41). When calculating the message in

(3.41), we multiply the incoming messages from all the factor nodes neighbouring the variable node, other than the factor node we are sending the message to. However in the clique tree, the separator node has its own potential, so we start by multiplying together the incoming messages from all the clique nodes neighbouring the separator node, other than the clique node we are sending the message to, and then multiply that product with the separator potential. Therefore the message from an separator node S_m to a clique node C_s is

$$\mu_{S_m \to C_s}(\mathbf{x}_m) = \phi_m(\mathbf{x}_m) \prod_{r \in ne(S_m) \backslash C_s} \mu_{C_r^* \to S_m}(\mathbf{x}_r), \tag{3.72}$$

where $\mathbf{x}_m$ is the set of variables in the separator ϕ_m, $\mathbf{x}_r$ is the set of variables in the clique ψ_r and $ne(S_m)\backslash C_s$ is the set of clique nodes that are neighbours of S_m other than C_s, the clique node we are sending the message to. Note that the messages sent to the separator node are from updated clique nodes only.

To ensure consistancy among the potentials we need to update the value of the separator potentials as well, and to do this we split (3.72) into a message and an update as we did with (3.69). We update the separator potential ϕ_m by multiplying it with all the incoming messages to the separator node S_m. Thus the updated value ϕ_m^* of the potential ϕ_m is given by

$$\phi_m^*(\mathbf{x}_m) = \phi_m(\mathbf{x}_m) \prod_{r \in ne(S_m) \backslash C_s} \mu_{C_r^* \to S_m}(\mathbf{x}_r). \tag{3.73}$$

From (3.70), the message from an updated separator node S_m to a clique node C_s becomes

$$\mu_{S_m^* \to C_s}(\mathbf{x}_m) = \phi_m^*(\mathbf{x}_m). \tag{3.74}$$

From (3.74) we see that the message from an updated separator node to a neighbouring clique node is the updated potential of the separator node, therefore (3.70) becomes

$$\psi_s^*(\mathbf{x}_s) = \psi_s(\mathbf{x}_s) \prod_{r \in ne(C_s) \backslash S_m} \phi_r^*(\mathbf{x}_r), \tag{3.75}$$

The forward-pass of the sum-product algorithm on a junction tree is executed using (3.71), (3.73) and (3.75). When a clique node is visited we update its potential using (3.75). Whenever a separator node is visited we update its potential using (3.73) and (3.71). This process amounts to belief propagation, with the messages being saved within the potentials. As with the factor graph sum-product algorithm, the next step is to execute the backward pass to ensure consistency of the potentials. However, to retain the joint probability distribution, when updating the clique potentials for the second time we need to remove the value of the separator potentials from the previous update, therefore we must modify (3.75).

To execute the backward pass we visit the nodes in reverse message passing order, and update them as we visit. To update a clique node C_s^* during the backward pass we multiply its current potential ψ_s^* with the the updated values of its neighbouring separator node's potentials, other than the separator node S_m that it updates, and divide out the previous values of its neighbouring separator node's potentials. The equation to update a clique node C_s is given by the equation

$$\psi_s^{**}(\mathbf{x}_s) = \psi_s^*(\mathbf{x}_s) \frac{\prod_{r \in ne(C_s) \backslash S_m} \phi_r^{**}(\mathbf{x}_r)}{\prod_{r \in ne(C_s) \backslash S_m} \phi_r^*(\mathbf{x}_r)}. \tag{3.76}$$

We now generalize (3.76) so it can be used for the forward and backward pass. We denote a previous value without a * superscript, such as ψ_s, and an updated value with a * superscript, such as ψ_s^*, regardless of whether we are updating during the forward or backward pass. The equation to update a clique node C_s is given by the equation

$$\psi_s^*(\mathbf{x}_s) = \psi_s(\mathbf{x}_s)\frac{\prod_{r\in ne(C_s)\setminus S_m}\phi_r^*(\mathbf{x}_r)}{\prod_{r\in ne(C_s)\setminus S_m}\phi_r(\mathbf{x}_r)}. \tag{3.77}$$

Note that during the forward pass the value of $\prod_{r\in ne(C_s)\setminus S_m}\phi_r(\mathbf{x}_r)$ is one, since all the separator potentials are initialized to one. The message from an updated clique node C_s^* to a separator node S_m is still given by (3.71), and the equation to update a separator node S_m is still given by (3.73).

In summary, during both the forward and backward pass, when a clique node is visited we update its potential using (3.77). Whenever a separator node is visited we update its potential using (3.73) and (3.71).

Once the sum-product algorithm is performed, we evaluate the marginal probability distribution $p(x_n)$ of any node x_n in the original graph by selecting any clique node or separator node in the junction tree that contains the variable x_n and marginalizing that node's potential over all its variables other that x_n. For example, the value of the marginal $p(x_2)$ for the graph in Figure 3.23, after performing the sum-product algorithm on its junction tree in Figure 3.26(a) is

$$p(x_2) = \frac{1}{Z}\,\phi_2(x_2), \tag{3.78}$$

with the normalization factor

$$Z = \sum_{x_2}\phi_2(x_2). \tag{3.79}$$

Not only have the potentials been transformed into marginals, but the joint probability distribution has been preserved, and is given by

$$p(\mathbf{x}) = \frac{\prod_i \psi_i^{**}(\mathbf{x}_i)}{\prod_i \phi_i^{**}(\mathbf{x}_i)}. \tag{3.80}$$

The message equations required to perform the max-sum algorithm on clique tree's can be determined in the same way, however the backtracking step in the max-sum algorithm does become more complicated in a clique tree, therefore the handling of the max-sum algorithm in clique trees has been left for Section 3.5.2.

Now that we have the tools to perform belief propagation in a junction tree, we look back and determine why the junction tree property is necessary for belief propagation so be correct in a clique tree. Why can we not just apply belief propagation to any clique tree? Consider the clique tree in Figure 3.25(b), which is not a junction tree, and assume we applied the sum-product algorithm to it. Note that the clique node C_4 involves the variable x_5, and this variable is marginalized out of the message sent to C_3 from C_4, since it does not appear in C_3's potential. However, the variable does appear in C_2, yet the message from C_3 to C_2 contains no information about x_5,this information was marginalized out when calculating the message from C_4 to C_3. Therefore, once the message passing is completed we cannot guarantee that the information on the node x_5 is consistent in all the clique potentials. The junction tree property ensures that all cliques involving an arbitrary variable

x_i form a connected subtree in the clique tree, ensuring that after message passing all the cliques are consistent with each other.

If we have a triangulated graph, we extract a corresponding junction tree from it, and perform the sum-product algorithm to determine the marginal probability distribution for any node in the original graph. But what if the original graph is not triangulated? Recall Figure 3.23, which was created by reconstructing Figure 2.3 with the edges added by the elimination algorithm. Figure 2.3 is not a triangulated graph, but Figure 3.23 is, and it turns out that the elimination algorithm is the key to triangulating a non-chordal graph. Since we are only interested in the edges added in by the elimination algorithm we do not need to calculate the intermediate factors, we are purely interested in the graph theoretic process of eliminating the nodes. Therefore, to triangulate a graph, we visit each node in the graph, connect its neighbours with edges, and then eliminate the node, until there are no nodes left, all the while saving the edges added in. Once this is done we simply reconstruct the original graph, adding in the elimination edges resulting in the triangulated graph. Adding edges to a graph reduces the number of conditional independence statements associated with the graph, therefore increasing the set of probability distributions associated with the graph. Thus means that the set of probability distributions associated with the triangulated graph includes the set probability distributions associated with the original graph, and solving the inference problem on the triangulated graph also solves it for the original graph.

We have developed almost all the tools required to perform the junction tree algorithm, and we now address the final one, which is the procedure of extracting a junction tree from a triangulated graph. Not every clique tree created from a triangulated graph is a junction tree, such as Figure 3.26(b) created from the triangulated graph in Figure 3.23. Let us assign a weight to every edge in clique tree as the number of variables indexed by the separator node on the edge. The total weight of a clique tree is the sum of the weights of all the edges. If we compare the two clique trees in Figure 3.26, we see that Figure 3.26(a) has a total weight of 5 and Figure 3.26(b) has a total weight of 4. It turns out that Figure 3.26(a) has the maximum weight of any possible clique tree that can be formed from Figure 3.23. Therefore, the clique tree with the maximum weight over all clique trees that can be formed from a triangulated graph is a junction tree. Basically, we need to find the maximum spanning tree over the cliques, and we use Dijkstra's greedy algorithm to solve this problem.

The junction tree algorithm illustrates the beautiful connection between graph theoretic algorithms and probability theory. Within the junction tree algorithm we use several graph-theoretic algorithms, such as node elimination developed in Section 3.2, as well as the depth-first search algorithm and Dijkstra's algorithm, both of which are well known graph-theoretic algorithms.With the junction tree sum-product algorithm we now find the marginal probability distribution for any node in any triangulated graph, or arbitrary graph that can be efficiently triangulated. For graphs that cannot be easily triangulated, we use approximate inference algorithms such as the loopy belief algorithm presented in Section 3.6.

3.5.2 Max-sum in junction trees

The max-sum algorithm in a junction tree follows the same principles as the max-sum algorithm in a factor tree. However, due to the multi-variable nature of the nodes in the junction tree there are a few differences. Firstly, the messages calculated during the forward pass are easily determined by replacing the summation in equations (3.69) and (3.72) with a maximization, and the taking the logarithm of both equations to avoid numerical underflow problems. Since we do not need to save the messages created to determine the MAP configuration, we do not need to update the clique and separator potentials as we did in the sum-product algorithm for a junction tree, though it is possible to do so. Thus we focus on the message passing required to obtain the MAP configuration of a junction tree. The message from a clique node C_s to a separator node S_m in the junction tree max-sum algorithm is

$$\mu_{C_s \to S_m}(\mathbf{x}_m) = \max_{\mathbf{x}_s \backslash \mathbf{x}_m} \log(\psi_s(\mathbf{x}_s)) + \sum_{r \in ne(C_s) \backslash S_m} \mu_{S_r \to C_s}(\mathbf{x}_r), \tag{3.81}$$

where $\mathbf{x}_m$ is the set of variables of the separator potential ϕ_m, $\mathbf{x}_s$ is the set of variables in the clique ψ_s, $\mathbf{x}_s \backslash \mathbf{x}_m$ is the set of all variable occurring in $\mathbf{x}_s$ that are not in $\mathbf{x}_m$ and $ne(C_s)\backslash S_m$ is the set of separator nodes that are neighbours of C_s other than S_m, the separator node we are sending the message to. The message from a separator node S_m to a clique node C_s is

$$\mu_{S_m \to C_s}(\mathbf{x}_m) = \phi_m(\mathbf{x}_m) + \sum_{r \in ne(S_m) \backslash C_s} \mu_{C_r \to S_m}(\mathbf{x}_r), \tag{3.82}$$

where $\mathbf{x}_m$ is the set of variables in the separator ϕ_m, $\mathbf{x}_r$ is the set of variables in the clique potential ψ_r and $ne(S_m)\backslash C_s$ is the set of clique nodes that are neighbours of S_m other than C_s, the clique node we are sending the message to. Note that we have not explicitly applied the log function to the separator potential ϕ_m, because instead of initializing the separator potentials to arrays of ones, we initialize them to arrays of zeros, since $\log(1) = 0$. Therefore, the log function has already been applied to the separator potentials, and during the message passing they are summed with other log values from the clique nodes.

The main difference when applying the max-sum algorithm to a junction tree is the backtracking step. Since we are no longer working directly with the nodes, but with the cliques instead, we need to save the backtracking information during the forward pass differently and perform the backtracking step differently. We make use of the example graph shown in Figure 3.28 to illustrate the process.

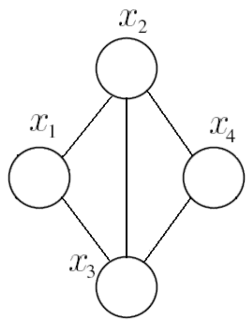

Figure 3.28: A two clique Markov random field

Assume we wish to calculate the MAP configuration for Figure 3.28, using the junction tree algorithm. We first obtain the graph's junction tree as shown in Figure 3.29.

We set the clique node C_1 as the root node and begin the forward pass. The first message passed is from the clique node C_2 to the separator node S_1, and has the form

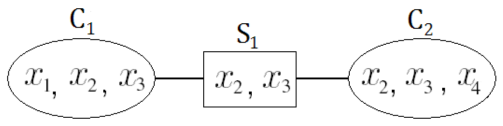

Figure 3.29: The junction tree for the Figure 3.28

.

$$\mu_{C_2\to S_1}(x_2,x_3) = \max_{x_4} \log(\psi_2(x_2,x_3,x_4)). \tag{3.83}$$

The result of (3.83) is a two dimensional table ranging over all the possible combinations of the nodes x_2 and x_3. When we perform this maximization we need to save the values of x_4 that correspond to each possible configuration of the x_2 and x_3. These are stored in a two dimensional table, with the same number of entries as $\mu_{C_2\to S_1}$. Continuing on with the message passing, since the separator node S_1 only has one neighbour other than C_2, it simply passes its incoming message as its outgoing message, therefore

$$\mu_{S_1\to C_1}(x_2,x_3) = \mu_{C_2\to S_1}(x_2,x_3), \tag{3.84}$$

Now we determine the MAP configuration of the nodes indexed by the clique node C_1 with the following equation

$$({x_1}^{max}, {x_2}^{max}, {x_3}^{max}) = \operatorname*{argmax}_{x_1,x_2,x_3}[\log(\psi_1(x_1,x_2,x_3)) + \mu_{S_1\to C_1}(x_2,x_3)]. \tag{3.85}$$

Now that we have set the MAP configurations of the nodes indexed by the root clique, we need to backtrack and set the MAP values of all the other nodes from the original graph to the same MAP configuration. Using the table we saved when calculating (3.83), we lookup the value of the node x_4 that corresponds to the values ${x_2}^{max}$ and ${x_3}^{max}$, and set the node x_4 to that value. It is therefore clear that backtracking in the junction tree max-sum algorithm is similar to the factor graph max-sum algorithm, the main difference being that instead of saving a simple backtracking path between nodes, we need to save multi-dimensional tables of values corresponding to the configurations of several variables. To save the backtracking information, whenever we calculate (3.81), we also need to calculate and save

$$\phi_{C_s\to S_m}(\mathbf{x}_m) = \operatorname*{argmax}_{\mathbf{x}_s\setminus\mathbf{x}_m} \log(\psi_s(\mathbf{x}_s)) + \sum_{r\in ne(C_s)\setminus S_m} \mu_{S_r\to C_s}(\mathbf{x}_r), \tag{3.86}$$

and later use this function to perform the backtracking step.

With the junction tree max-sum algorithm we now find the MAP configuration of any triangulated graph, or arbitrary graph that can be efficiently triangulated. For graphs that cannot be triangulated, we use approximate inference algorithms such as the loopy belief algorithm presented in Section 3.6.

3.6 Loopy belief propagation

So far in this thesis we have focused on exact inference algorithms, the most powerful of which being the junction tree algorithm. The junction tree algorithm works by triangulating non-chordal graphs and then extracting a junction tree from the triangulated graph to perform belief propagation. However, there are classes of graphs that cannot be triangulated very easily, or at all such as lattice structured graphs. An example of a lattice structured graph is shown in Figure 3.30.

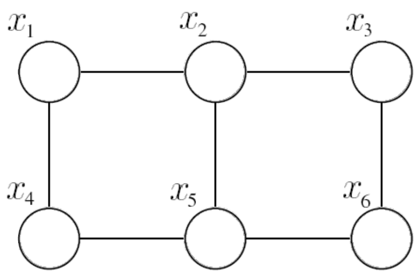

Figure 3.30: An example of a lattice structured graph.

When we encounter graphs such as Figure 3.30, or any class of graph that cannot be efficiently evaluated by exact inference algorithms, it is time to turn to approximate probabilistic inference algorithms, such as the loopy belief propagation algorithm [Murphy et al., 1999], [Tanaka and Titterington, 2003].

3.6.1 Derivation

The basic problem is that the graph we are performing inference on is not a tree, and cannot be converted into a junction tree. If we apply the sum-product algorithm to a graph with loops, then each node's outgoing messages affect its incoming messages, which affect its outgoing messages, and so on. The idea is to iterate the sum-product algorithm until the messages, and the potentials, converge.

Loopy belief propagation is in essence a series of localized applications of the sum-product algorithm to a graph. To outline this process we make use of the graph shown in Figure 3.31(a) and its corresponding clique graph shown in Figure 3.31(b). Assume we want to calculate the marginal probability $p(x_1)$.

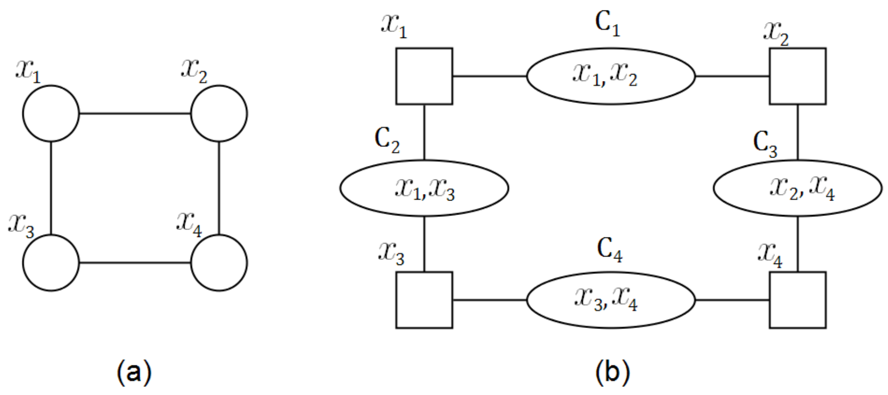

Figure 3.31: (a) A simple lattice graph. (b) The clique graph representation of (a).

Obviously we cannot apply the sum-product algorithm to Figure 3.31(b), because it is not a tree. However, consider the clique node C_1 and its neighbouring variable, or separator nodes, x_1 and x_2. These three nodes form a connected subtree within Figure 3.31(b), and in fact every clique node and its neighbouring separator nodes form a subtree within any clique graph. Figure 3.32 shows the four subtrees extracted from Figure 3.31(b).

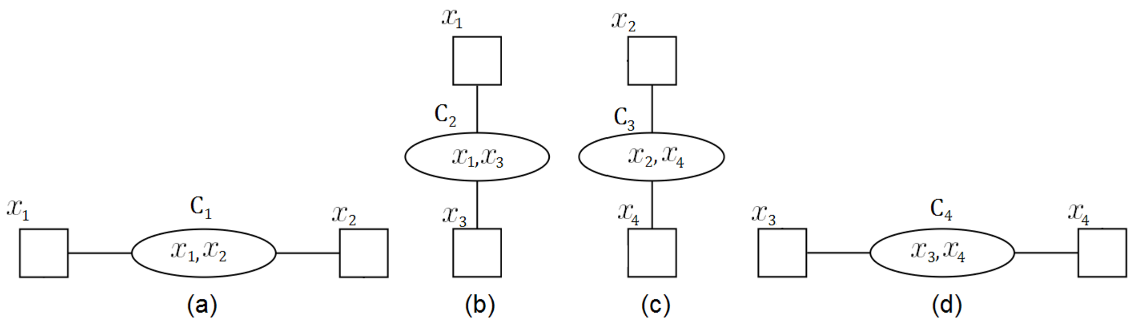

Figure 3.32: The four subtrees extracted from Figure 3.31(b).

Since all the graphs shown in Figure 3.32 are trees, we apply the sum-product or max-sum algorithms to them. However, the subtrees are not independent of each other, in this example each one shares separator nodes with two other trees. Therefore, when we apply the sum-product algorithm to one of the subtrees, we need to somehow share the messages created during the backward pass with all the other subtrees it shares nodes with. Note that the subtrees all overlap at separator nodes, so these nodes are a good place to store the required messages between the subtrees. The solution is to introduce a separator potential $\phi_n(x_n)$ over every separator node x_n, and we initialize these potentials to arrays of ones, just like the separator potentials in the junction tree algorithm in Section 3.5. Using the separator potentials, we store the message sent to a separator node x_n in the backward pass in one subtree, and use it in the forward pass in a subtree that shares the node x_n. The separator potentials act as a storage point for related messages. The general idea of loopy belief propagation is to select a clique node, perform the sum-product algorithm locally on the subtree formed by that clique node and its immediate neighbouring separator nodes, and then apply this process on every clique node in the graph until all the clique nodes have been visited. Once all the clique nodes have been visited, one iteration of the loopy belief propagation algorithm is completed. We continue to iterate the process until the clique node's potentials converge, or to some pre-determined cutoff point.

Let us consider the sum-product algorithm on a general subtree, with the inclusion of the separator potentials. Note that any subtree formed by a clique node and its immediate separator nodes exhibits the general structure shown in Figure 3.33.

Since the separator potentials need to be systematically updated by the sum-product algorithm in each subtree, the loopy belief propagation algorithm focuses on updating the clique and separator potentials rather than the message passing itself. If we are to perform the sum-product algorithm on the general graph in Figure 3.33, we start with the forward pass by propagating messages in from the leaf nodes to the root node. Since a separator node in a subtree has no neighbours other than the clique node we send its message to, the loopy belief message from a separator node x_m to a clique node C_s is simply the separator nodes potential $\phi_m(x_m)$. Therefore

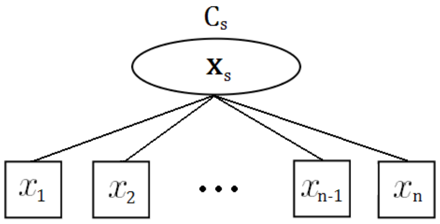

Figure 3.33: A subtree extracted from a clique graph with one clique node and its n separator node neighbours.

$$\mu_{x_m \to C_s}(x_m) = \phi_m(x_m). \tag{3.87}$$

So for the forward pass, we update the clique node's potential by multiplying it with the separator potentials of all its neighbouring separator nodes. Therefore,

$$\psi_s{}^*(\mathbf{x}_s) = \psi_s(\mathbf{x}_s) \prod_{m \in ne(C_s)} \phi_m(x_m), \tag{3.88}$$

where $\psi_s{}^*(\mathbf{x}_s)$ is the updated value of $\psi_s(\mathbf{x}_s)$ after the forward pass, and $ne(C_s)$ is the set of all separator nodes that are neighbours of the clique node C_s. Now that the clique node's potential is updated, we update the separator potentials of its neighbouring separator nodes. The message from an updated clique node C_s^* to a separator node x_m is simply the clique node's potential marginalized over all its variables other than x_m, and is given by the equation

$$\mu_{C_s{}^* \to x_m}(x_m) = \sum_{\mathbf{x}_s \setminus x_m} \psi_s{}^*(\mathbf{x}_s). \tag{3.89}$$

The separator potentials at the separator nodes act as storage for the messages passed between subtrees, therefore we only wish to keep the most recent messages sent to the separator node. Therefore, before we update a separator potential with a message from the updated clique node $C_s{}^*$, we need to divide out the previous message (if there was one) sent to the separator potential from the clique node C_s in the previous iteration, which we denote by $\mu_{C_s \to x_m}(x_m)$. We now update each of the separator nodes in the subtree by dividing out the old message sent to each separator node from the subtree's only clique node, C_s and multiplying each one's potential with the new message sent to it from the updated clique potential $\psi_s{}^*$. Therefore, the equation to update the separator node x_m is

$$\phi_m{}^*(x_m) = \phi_m(x_m) \frac{\mu_{C_s^* \to x_m}(x_m)}{\mu_{C_s \to x_m}(x_m)}. \tag{3.90}$$

Using equations (3.88), (3.89) and (3.90) we apply sum-product style belief propagation iteratively to every subtree in a general factor graph until the potentials converge or we reach a set number of iterations. To illustrate the updating of the potentials, we use the example graph shown in Figure 3.34.

Each pane in Figure 3.35 shows the step of visiting a factor node from Figure 3.34 and applying equations (3.88), (3.89) and (3.90) to that factor node and its neighbouring separator nodes. Note in the second pane in Figure 3.35 that the factor node ψ_2 is updated by $\phi_1{}^*$ and ϕ_3. These two

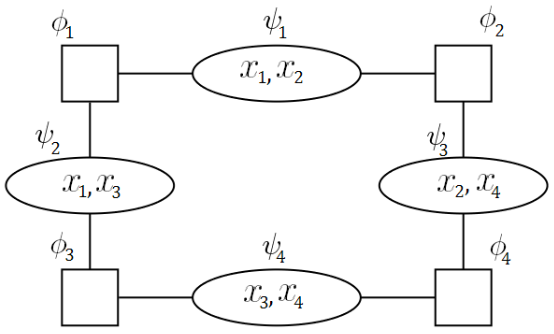

Figure 3.34: The graph Figure 3.31(b) showing the potentials assigned to each node.

separator sets are not consistent, since one has been updated and the other has not. We see that this is the case in almost every step. It is for this reason that we need to iterate the entire process shown in Figure 3.35 until the factor potentials converge.

Once the factor potentials have converged, we approximate the marginal probability distribution $p(x_n)$ for any node x_n by selecting any factor node that contains x_n and marginalizing that factor potential over all its variables other than x_n.

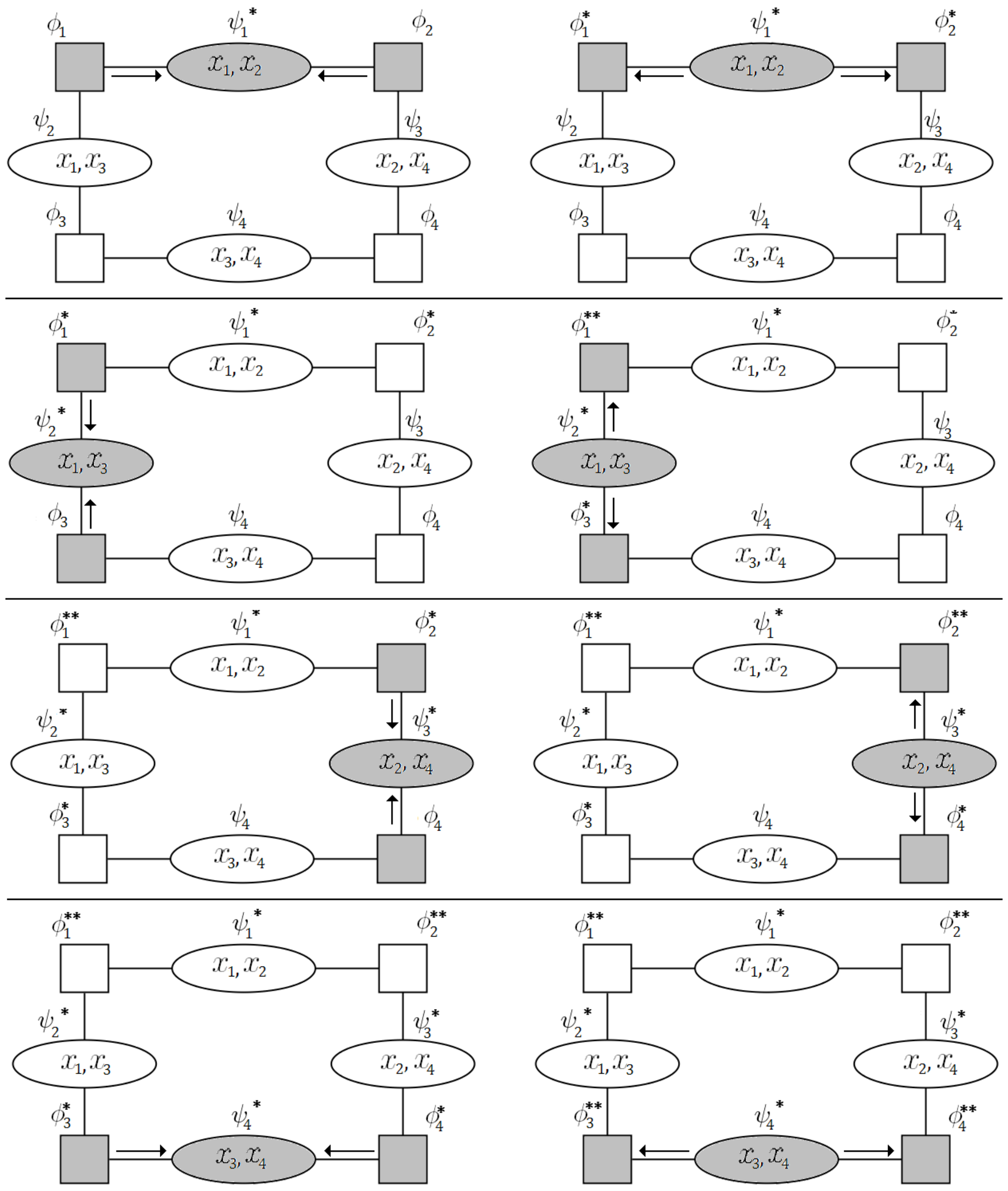
ϕ_1 ψ_1^* ϕ_2 x_1, x_2 ψ_2 x_1, x_3 ψ_3 x_2, x_4 ϕ_3 ψ_4 ϕ_4 x_3, x_4
ϕ_1^* ψ_1^* ϕ_2^* x_1, x_2 ψ_2 x_1, x_3 ψ_3 x_2, x_4 ϕ_3 ψ_4 ϕ_4 x_3, x_4
ϕ_1^* ψ_1^* ϕ_2^* x_1, x_2 ψ_2^* x_1, x_3 ψ_3 x_2, x_4 ϕ_3 ψ_4 ϕ_4 x_3, x_4
ϕ_1^{**} ψ_1^* ϕ_2^* x_1, x_2 ψ_2^* x_1, x_3 ψ_3 x_2, x_4 ϕ_3^* ψ_4 ϕ_4 x_3, x_4
ϕ_1^{**} ψ_1^* ϕ_2^* x_1, x_2 ψ_2^* x_1, x_3 ψ_3^* x_2, x_4 ϕ_3^* ψ_4 ϕ_4 x_3, x_4
ϕ_1^{**} ψ_1^* ϕ_2^{**} x_1, x_2 ψ_2^* x_1, x_3 ψ_3^* x_2, x_4 ϕ_3^* ψ_4 ϕ_4^* x_3, x_4
ϕ_1^{**} ψ_1^* ϕ_2^{**} x_1, x_2 ψ_2^* x_1, x_3 ψ_3^* x_2, x_4 ϕ_3^* ψ_4^* ϕ_4^* x_3, x_4
ϕ_1^{**} ψ_1^* ϕ_2^{**} x_1, x_2 ψ_2^* x_1, x_3 ψ_3^* x_2, x_4 ϕ_3^{**} ψ_4^* ϕ_4^{**} x_3, x_4

Figure 3.35: One iteration of the loopy belief propagation algorithm on Figure 3.34. The nodes marked gray form the subtree that the sum-product algorithm is being applied to in a specific step. The number of superscripts * a potential has indicates how many times it has been updated.

Chapter 4 - Learning

In this chapter we cover the process of estimating the parameters of a graphical model based on observed evidence (also known as training data). The observed evidence will consist of N samples, which we assume to be independent, where each sample consists of observations for every node in the network. The training data can take one of the following two forms:

1 Fully observed: Every sample has an observed value for every node in the model.

2 Partially observed: One or more of the nodes in one or more of the samples do not have an observed value. Therefore the training data contains hidden nodes.

We deal with the two forms of the training data differently. For fully observed data we apply the simple maximum likelihood estimation algorithm explained in Section 4.1, and for partially observed data we apply the Expectation-Maximization algorithm explained in Section 4.2. We begin each section by focusing on parameter estimation in Bayesian networks, and then extend the process to Markov random fields.

4.1 Maximum-likelihood-estimation

In this section we outline parameter estimation through fully observed data using maximum-likelihood-estimation [Dempster et al., 1977], [Didelez and Pigeot, 1998], [Ghahramani, 2003]. To perform learning on a Bayesian network, we need to estimate the parameters in such a way as to maximize the likelihood of the training data. We supply a concrete look at applying maximum-likelihood-estimation to a Bayesian network. Rather than dealing directly with the algebra we use a suitable example. Consider the graph shown in Figure 4.1, and assume we have a training set S, with N samples that are fully observed. An example of the training set S with 4 samples is shown in Table 4.1.

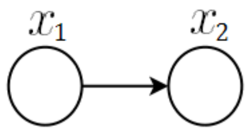

Figure 4.1: A simple Bayesian network with two nodes.

We know from (2.1) that the joint probability distribution of a Bayesian network factorises into the product of the conditional probability distributions associated with the network. Therefore, we attempt to learn the parameters of each conditional probability distribution separately. The Bayesian

	x_1	x_2
Sample 1	0	0
Sample 2	0	1
Sample 3	1	1
Sample 4	0	0

Table 4.1: An example of a training set for the graph shown in Figure 4.1.

network in Figure 4.1 has the following two conditional probability distributions: $p(x_1)$ and $p(x_2|x_1)$. Let us start by determining $p(x_1)$. Since it only involves the variable x_1, we need only consider the sample values for x_1 in the training set S. The simplest way to determine $p(x_1)$ is to count the number of times x_1 takes on a particular value in the given samples and divide the count by the number of total samples. Therefore, we pick the values of the model paramaters that make the training data for node x_1 'more likely' than any other values of the parameters would make them.

For example, in the training set shown in Table 4.1, x_1 takes the values $\{0, 0, 1, 0\}$. Therefore, $x_1 = 0$ occurs 3 times out of 4 samples, and $x_1 = 1$ occurs once out of 4 samples, so $p(x_1 = 0) = 3/4$ and $p(x_1 = 1) = 1/4$. Based on this we estimate the the conditional probability distribution $p(x_1)$ to have the value shown in Table 4.2.

$x_1 = 0$	$x_1 = 1$
0.75	0.25

Table 4.2: The maximum-likelihood estimate of the conditional probability distribution $p(x_1)$, estimated from the sample data in Table 4.1.

Let us now attempt to calculate the conditional probability distribution $p(x_2|x_1)$. We determine $p(x_2|x_1)$ in a similar fashion to $p(x_1)$, but instead of counting the values that the variables x_1 and x_2 take on separately, we count the different combinations of two variables. For example, in the sample set shown in Table 4.1 the variable pair (x_1, x_2) takes the combination pairs $\{(0, 0), (0, 1), (1, 1), (0, 0)\}$. We now count the number of times each combination occurs and divide it by the total number of samples to get the probability of each combination. Therefore, $p(x_2 = 0|x_1 = 0) = 2/4$, $p(x_2 = 0|x_1 = 1) = 1/4$, $p(x_2 = 1|x_1 = 1) = 1/4$ and $p(x_2 = 1|x_1 = 0) = 0/4$. Based on these values by forcing the sum of each column to be one, we determine the value of the conditional probability distribution $p(x_2|x_1)$ to be that shown in Table 4.3.

	$x_1 = 0$	$x_1 = 1$
$x_2 = 0$	0.66	0
$x_2 = 1$	0.33	1

Table 4.3: The maximum-likelihood estimate of the conditional probability distribution $p(x_2|x_1)$, estimated from the sample data in Table 4.1.

Based on these examples, we know that to estimate the parameters of a Bayesian network from a set of training data we estimate the value of each conditional probability distribution separately. To estimate the value of a conditional probability distribution $p(x_i|\pi_i)$ we determine the value of each entry in $p(x_i|\pi_i)$ separately. Each entry in $p(x_i|\pi_i)$ corresponds to a specific combination of the node x_i and its parents π_i, thus to calculate an entry we simply count the number of times its corresponding combination appears in the samples and divide it by the number of total samples.

The question now is, how do we perform parameter estimation in a Markov random field? We do almost exactly the same as with a Bayesian network, except we estimate the entries of separate clique potentials instead of the conditional probability distributions. Therefore, to estimate the value of the clique potential $\psi(\mathbf{x}_s)$, we would simply count the number of times each combination of the variables in the set $\mathbf{x}_s$ occurs in the training data and divide it by the total number of samples.

4.2 Expectation-maximization

So far we have covered the estimation of parameters from fully observed data, and in this section we outline maximum-likelihood-estimation of parameters from partially observed data via the well-known Expection-Maximization (EM) algorithm [Dempster et al., 1977], [Bishop, 2006], [Didelez and Pigeot, 1998], [Ghahramani, 2003]. The EM algorithm has been widely studied, and in this thesis we provide a concrete look of applying the algorithm to graphical models with discrete variables. Consider the Markov random field shown in Figure 4.2, and assume we have a partially observed training set S with N samples, an example of S with 4 samples is shown in Table 4.4.

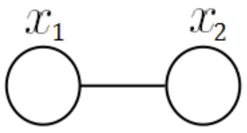

Figure 4.2: A simple Markov random field with two nodes.

	x_1	x_2
Sample 1	0	0
Sample 2	1	1
Sample 3	Hidden	1
Sample 4	0	0

Table 4.4: An example of a partially observed training set for the graph shown in Figure 4.2.

The EM algorithm consists of two main steps, the *expectation step* and the *maximization step*. In the expectation step we *infer*, using a suitable inference algorithm, the distribution over the hidden variables given the observed variables. In the maximization step we use the inferred distribution and the observed variables to update the parameters of the model. We then iterate these two steps until the parameters of the model converge, or a specified maximum number of iterations is reached.

For the expectation step step we need to perform inference, and to perform inference we need an estimate of the model's parameters and some observed evidence. For each sample in the training set, we use it as observed evidence to execute the sum-product algorithm on the model, using an estimate for the model's parameters, and then extract the marginal probability distribution over any nodes which were hidden in the sample. We then use this marginal to 'fill in' the missing information in the sample caused by the hidden nodes, creating an estimated sample. Using the estimated samples, which form a complete training set, we apply maximum-likelihood estimation to determine the models *expected sufficient statistics*. The expected sufficient statistics are a running estimate of the models parameters, and are used during the maximization step to update the models parameters.

For the first iteration of the EM algorithm we initialize the parameters of the model to one, and in the following iterations the parameters will have been updated by the previous maximization step.

We now turn to our example graph and sample set. In this example we illustrate the effect the EM algorithm has on one clique potential, as this is the core of the EM algorithm. This process is later expanded to incorporate graphs with any number of clique potentials. The Markov random field in Figure 4.2 has one maximal clique to which we assign the potential $\psi(x_1, x_2)$. We begin the expectation step by initializing the potential $\psi(x_1, x_2)$ to one and its expected sufficient statistics, $ESS(x_1, x_2)$, to zero, as shown in Table 4.5.

$\psi(x_1, x_2)$	$x_1 = 0$	$x_1 = 1$
$x_2 = 0$	1	1
$x_2 = 1$	1	1

$ESS(x_1, x_2)$	$x_1 = 0$	$x_1 = 1$
$x_2 = 0$	0	0
$x_2 = 1$	0	0

Table 4.5: The potential $\psi(x_1, x_2)$ and its expected sufficient statistics $ESS(x_1, x_2)$ initialized for the EM algorithm.

The samples $\{0, 0\}$, $\{1, 1\}$ and $\{0, 0\}$ are all fully observed within the domain of the clique potential $\psi(x_1, x_2)$, so we simply increase the entires each combination indexes in the expected sufficient statistics by the number of times that combination appears. The expected sufficient statistics after incorporating these samples are shown in Table 4.6.

$ESS(x_1, x_2)$	$x_1 = 0$	$x_1 = 1$
$x_2 = 0$	2	0
$x_2 = 1$	0	1

Table 4.6: The expected sufficient statistics $ESS(x_1, x_1)$ after incorporating the fully observed samples.

We now move on to the sample $(Hidden, 1)$, which is the first sample with a latent node. As stated earlier, we need to infer the distribution over the hidden node for the expected sufficient statistics. Therefore, we start by using this sample as evidence to run the sum-product algorithm on the Markov random field, using the potential $\psi(x_1, x_2)$ initialized in Table 4.5 as the current model parameters. Since this Markov random field contains only one clique potential, the sum-product algorithm is analogous to taking the appropriate slice of the potential $\psi(x_1, x_2)$ based on the evidence, as discussed in section 2.2.2. The resulting observed potential $\psi^E(x_1|x_2 = 1)$ is shown in Table 4.7.

$\psi^E(x_1\|x_2 = 0)$	$x_1 = 0$	$x_1 = 1$
$x_2 = 1$	1	1

Table 4.7: The partially observed potential $\psi^E(x_1|x_2 = 1)$.

We now need to obtain the marginal probability distribution over the hidden node x_1, and in this case it is simply the normalized version of $\psi^E(x_1|x_2 = 1)$, which is the marginal $\psi^M(x_1|x_2 = 1)$ shown in Table 4.8.

$\psi^M(x_1\|x_2 = 0)$	$x_1 = 0$	$x_1 = 1$
$x_2 = 1$	0.5	0.5

Table 4.8: The marginal distribution $\psi^M(x_1|x_2 = 1)$.

Next we need to resize the inferred marginal distribution to add it to the expected sufficient statistics. To do this we simply resize the potential $\psi^M(x_1|x_2 = 0)$ to the same dimensions as $\psi(x_1, x_2)$ and fill any newly created entries with zeroes, creating the new resized potential $\psi^R(x_1, x_2)$, shown in Table 4.9.

$\psi^R(x_1, x_2)$	$x_1 = 0$	$x_1 = 1$
$x_2 = 0$	0	0
$x_2 = 1$	0.5	0.5

Table 4.9: The resized potential $\psi^R(x_1, x_2)$ which will be added directly to the expected sufficient statistics.

Since $\psi^R(x_1, x_2)$ and $ESS(x_1, x_2)$ have the same shape, we simply add the resized potential to the expected sufficient statistics, resulting in the expected sufficient statistics shown in Table 4.10.

$ESS(x_1, x_2)$	$x_1 = 0$	$x_1 = 1$
$x_2 = 0$	2	0
$x_2 = 1$	0.5	1.5

Table 4.10: The expected sufficient statistics for the potential $\psi(x_1, x_2)$ once all the samples have been processed.

The expectation step is now complete, as we have obtained the expected sufficient statistics. We now move to the maximization step, in which we update the parameters of the model based on the expected sufficient statistics. We simply assign the expected sufficient statistics for the clique as its new clique potential for the next iteration, and reset the expected sufficient statistics to zero, the results of which are shown in Table 4.11.

$\psi(x_1, x_2)$	$x_1 = 0$	$x_1 = 1$
$x_2 = 0$	2	0
$x_2 = 1$	0.5	1.5

$ESS(x_1, x_2)$	$x_1 = 0$	$x_1 = 1$
$x_2 = 0$	0	0
$x_2 = 1$	0	0

Table 4.11: $\psi(x_1, x_2)$ and $ESS(x_1, x_2)$ after the first iteration of the EM algorithm.

After executing the expectation step and the maximization step, we have completed one iteration of the EM algorithm. We continue to iterate until the potential $\psi(x_1, x_2)$ converges. The process of applying the EM algorithm to a Markov random field with a single clique potential is summarized below.

```
Algorithm: EM on a Markov random field with a single clique potential $\psi(\mathbf{X}_s)$.

  1. Initialize the potential $\psi(\mathbf{X}_s)$ to one, and the expected
  sufficient statistics of $\psi(\mathbf{X}_s)$, $ESS(\mathbf{X}_s)$, to zero.

  2. For every sample $S_j$ in the sample set $\mathbf{S}$ execute the
  following steps:

    a) If the sample is fully observed, increase the value of
    the entry it indexes in $ESS(\mathbf{X}_s)$ by 1.

    b) If the sample is not fully observed then execute the
    following steps:

      i) Use the sample $S_j$ as evidence and execute the
      sum-product algorithm on the Markov random field using the current
      value of the potential $\psi(\mathbf{X}_s)$ as its parameters.

      ii) Obtain the marginal distribution over the hidden
      variables in the sample.

      iii) Reshape the marginal distribution to match the
      dimensions of $ESS(\mathbf{X}_s)$, filling the new entries
      with zeros.

      iv) Add the reshaped marginal to $ESS(\mathbf{X}_s)$.

  3. Normalize $ESS(\mathbf{X}_s)$ and assign it to $\psi(\mathbf{X}_s)$, and then reset
  $ESS(\mathbf{X}_s)$ to zero.

  4. Compare the log likelihood of this iteration with that of
  the previous iteration, and if they are similar within in
  some tolerance then the algorithm has converged and is
  complete. If the log likelihood has not converged, then
  return to step 2 and iterate once more.
```

To extend this algorithm for a Markov random field containing more than one clique potential, we once again exploit factorization of Markov random fields. We do this naively by simply applying the the algorithm discussed so far to each clique potential in the Markov random field. However, this would involve performing a separate inference step for each potential which heavily repeats computations. Fortunately, in Chapter 3 we developed inference algorithms, such as the junction tree and loopy belief propagation algorithms, which allow us to perform inference based on new evidence once, and then extract any marginal probability distributions we require afterwards. Using one of these algorithms allows us to factorise the inference step out of iteration through each potential, so we no longer repeat calculations. The general EM algorithm for a Markov random field with multiple clique potentials is shown below.

```
Algorithm: EM on a Markov random field with a multiple clique potentials.

  1. Initialize each potential $\psi_i(\mathbf{X}_s)$ to one, and the expected
  sufficient statistics each potential, $ESS_i(\mathbf{X}_s)$, to zero.

  2. For every sample $S_j$ in the sample set $\mathbf{S}$ execute the
  following steps:

    2.1) Use the sample $S_j$ as evidence and execute the
    sum-product algorithm on the Markov random field using the current value
    of the potentials as its parameters.

    2.2) For every clique potential $\psi_i(\mathbf{X}_s)$ in the Markov random field
    execute the following steps:

      a) If the sample is fully observed within the domain
      of the clique potential $\psi_i(\mathbf{X}_s)$ then increase the value
      of the entry it indexes in $ESS_i(\mathbf{X}_s)$ by 1.

      b) If the sample is not fully observed within the
      domain of the clique potential $\psi_i(\mathbf{X}_s)$ then execute
      the following steps:

        i) Obtain the marginal distribution over the
        hidden variables within the domain of the
        potential $\psi_i(\mathbf{X}_s)$.

        ii) Reshape the marginal distribution to
        match the dimensions of $ESS_i(\mathbf{X}_s)$, filling
        the new entries with zeros.

        iv) Add the reshaped marginal distribution to
        $ESS_i(\mathbf{X}_s)$.

  3. For every clique potential $\psi_i(\mathbf{X}_s)$ in the Markov random field,
  normalize $ESS_i(\mathbf{X}_s)$ and assign it to $\psi_i(\mathbf{X}_s)$, and then
```

```
reset $ESS_i(\mathbf{X}_s)$ to zero.

4. Compare the log likelihood of this iteration with
that of the previous iteration, and if they are
similar within in some tolerance then the algorithm
has converged and is complete. If the log likelihood
has not converged, then return to step 2 and iterate
once more.
```

Chapter 5 - Open-source software

The various classes and functions described in Chapter 3 are implemented in the open-source Python library GrMPy (Graphical Models for Python). The algorithms outlined in this thesis are the core of the library, and are invaluable in understanding how the code works. In this section we cover some of the improvements made to the algorithms while this thesis was written.

5.1 Overview

GrMPy has been developed as a cross-platform, open-source project using the freely available language Python. It also makes use of the NumPy [Jones et al., 2001-2008a] and SciPy [Jones et al., 2001-2008b] libraries that are both freely available from http://www.scipy.org. GrMPy includes a set of unit-tests to ensure future updates do not damage critical functionality. GrMPy also includes a full set of tutorial-like examples outlining the use of MLE and EM learning, as well as exact and approximate inference, on both Bayesian networks and Markov random fields.

5.2 New features

During the writing of this thesis, the GrMPy package has grown substantially, in both the number of features and the improved performance and robustness of the algorithms described in this thesis. A fundamental requirement of packages of this kind is that they can be easily extended. The new features described in this section were developed by a contributer to the GrMPy [Reikeras, 2009]. The quality of GrMPy's design is demonstrated by the ease with which it has been extended to incorporate Gaussian conditional probability distributions and Dynamic Bayesian Networks. For completeness these extensions are briefly explained in this section.

5.2.1 Gaussian conditional probability distributions

The GrMPy package now allows for nodes with continuous Gaussian distributions, instead of just the discrete distributions created from the class described in Section 6.4. This feature allows GrMPy to represent a much larger family of distributions, making it a much more powerful tool. Currently, only Bayesian networks support continuous nodes.

5.2.2 Dynamic Bayesian networks

Recently the GrMPy package has been extended to support Dynamic Bayesian networks (DBN). A DBN is a Bayesian network with a recurring structure, usually the same structure repeated over time.

An example of a DBN is the popular Hidden Markov model (HMM), such as the first-order HMM shown in Figure 5.1.

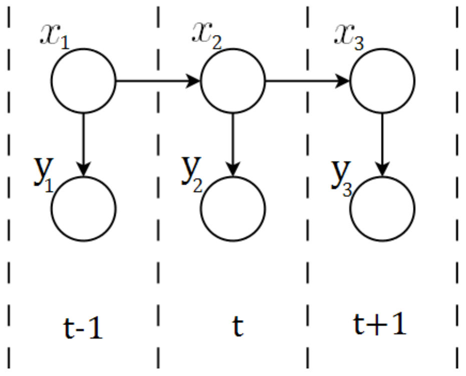

Figure 5.1: A first order Hidden Markov model.

Each pair of $\{x_n, y_n\}$ nodes in Figure 5.1 represent a time slice in the DBN. To define a DBN we need to only specify the structure of one time slice, and which nodes have edges to themselves in the next time slice. To perform inference on a DBN we then 'unroll' the repeated structure out into a Bayesian network, and perform standard junction tree or loopy belief propagation inference on it. The inference and parameter estimation algorithms for graphical models exhibiting a dynamic nature are often very model specific. For instance, the inference algorithms for a first order HMM cannot be applied to a general Bayesian network. In the GrMPy framework, general DBN's are supported, allowing the representation of a much broader class of distributions. DBNs are useful for modeling discrete time-dynamic processes.

Chapter 6 - Implementation

In this section we define the separate classes and data structures required to represent and manipulate graphical models. We also define the methods required to manipulate these classes and data structures, as well as the general methods required to perform the various graphical model algorithms discussed in this thesis. All of these classes and functions are implemented in the GrMPy package. Below we identify the required classes:

1 MODEL: This is an abstract class, which is the base class for all types of graphical models.

2 BNET: This class represents a Bayesian network.

3 MRF: This class represents a Markov random field.

4 CPD: This class represents a discrete conditional probability distribution.

5 CLIQUE: This class represents a clique with a discrete potential.

6 POTENTIAL: This class represents a discrete potential.

We also define the following classes, referred to as *inference engines*, to perform probabilistic inference,

1 INF_ENGINE: This is an abstract class, which is the base class for all types of inference engines.

2 JTREE: This class performs exact probabilistic inference on both Markov random fields and Bayesian networks via the junction tree algorithm.

3 LOOPY: This class performs approximate probabilistic inference on both Markov random fields and Bayesian networks via the loopy belief algorithm.

The reason we define classes for the inference algorithms is because when we run the junction tree algorithm or the loopy belief algorithm on a graphical model, we need to store all messages produced during the algorithm so that we calculate the marginal probability of any node in the graphical model. It is therefore convenient to create classes that can run the algorithms, store the messages and return the marginal probability of any node we request. We interface with the inference engine through the graphical model classes BNET and MRF. In the rest of this chapter we outline the attributes and methods for each of these classes that allow us to implement graphical model software.

The pseudo-code used to detail the algorithms in this chapter is based on the programming language Python. For simplicity we make use of a Python style *list* data structure in many of the algorithms. The *list* structure is a dynamic data structure which can store any type of data. In the algorithms to follow, *list* variables are often used to store the indices of nodes in the model, representing sets of nodes within the model. When we say that an integer i indexes a node, we mean that it refers to node x_i in the model. Thus, if a list containing the values $\{0, 2, 5\}$ indexes the node set $\{x_0, x_2, x_5\}$ in a model.

6.1 The MODEL class

This class is the abstract base class for both the BNET and MRF classes. This class defines a set of methods that are common to both the BNET and MRF classes, and are used to interface with the two class's inference engines. Each of the following paragraphs details a method within the MODEL class.

MODEL.init_inference_engine: Initializes the desired inference engine for the model.
To initialize the inference engine of a MODEL object we need the following parameters:

1 *boolean: exact.* A value which is TRUE if we are to run exact probabilistic inference using the junction tree algorithm, and FALSE if we are to run approximate probabilistic inference using the loopy belief algorithm.

2 *integer: max_iter.* An integer indicating how many iterations of the loopy belief algorithm can be executed before terminating it. The default is 10.

The algorithm for the MODEL init_inf_engine method is shown below.

```
MODEL.init_inf_engine(exact, max_iter=10)
  STEP 1: Determine whether to use a exact or approximate
          inference engine and initialize the appropriate
          inference engine.

    IF exact == TRUE
      self.engine <- JTREE.init(self)
    ELSE
      self.engine <- LOOPY.init(self, max_iter)
    END
```

MODEL.sum_product: Executes the sum-product algorithm on the model using the inference engine initialized in the MODEL.init_inference_engine method.
This function is merely an interface to the inference engine, and requires only the one parameter shown below.

1 *list: evidence.* A list of observed evidence with the length equal to the number of nodes in the MODEL. The value of evidence[i] is either None, to represent that the node x_i is unobserved, or is an integer value k to represent the observation $x_i = k$.

Given the evidence list we execute the sum-product algorithm within the inference engine using the algorithm shown in below.

```
MODEL.sum_product(evidence)
  STEP 1: Execute the sum−product method within the inference
          engine and return the result.

    loglik <− self.engine.sum_product(evidence)
    return loglik
```

MODEL.max_sum: Executes the max-sum algorithm on the model using the inference engine initialized in the *init_inference_engine* method.

This function is merely an interface to the inference engine, and requires only the one parameter shown below.

1 *list: evidence.* A list of observed evidence with the length equal to the number of nodes in the MODEL. The value of evidence[i] is either None, to represent that the node x_i is unobserved, or an integer value k to represent the observation $x_i = k$.

The method returns the following value

1 *list: mlc.* This list contains the values of the most-likely-configuration of the model as determined by the max-sum algorithm. It has a length equal to the number of the nodes in the model. If $mlc[i] = k$ then the variable node x_i has the most likely value k.

Given the evidence list we execute max-sum algorithm within the inference engine using the algorithm shown below.

```
MODEL.max_sum(evidence)
  STEP 1: Execute the max−sum method within the inference
          engine and return the result.

    mlc <− self.engine.max_sum(evidence)
    return mlc
```

MODEL.marginal_nodes: Returns the marginal probability distribution, or maximum probability, of a specified node or nodes in the model.

Just like the MODEL.sum_product and MODEL.max_sum methods, this method is merely an interface to the inference engine associated with a MODEL object, and requires the following parameters.

1 *list: query.* A list of the indices of the nodes we wish to obtain a single marginal probability distribution for.

2 *boolean: maximize.* The value of this parameter is FALSE if we require the marginal probability distribution of the nodes indexed in the parameter *query*, or it is TRUE if we wish to find the maximum probabilities of the nodes indexed in the parameter *query*. The default is FALSE.

The method returns the following value

1 *POTENTIAL: marginal.* This is the desired marginal probability for the nodes indexed by the parameter *query*.

Given the parameters we determine the required marginal probability distribution through the inference engine using the algorithm shown below.

```
MODEL.marginal_nodes(query, maximize=FALSE)
  STEP 1: Execute the method within the inference engine
          which returns the value of the marginalized node(s).

    marginal <- self.engine.marginal_nodes(query, maximize)
    return marginal
```

6.2 The BNET class

The BNET class represents a Bayesian network.

6.2.1 Attributes

We begin by detailing the implementation of Bayesian networks by first identifying the attributes involved. Based on Section 2.1, we know that a Bayesian network is composed of

1 A set of N nodes.

2 A set of directed edges connecting the nodes.

3 A set of conditional probability distributions associated with the nodes.

With the attributes identified we need to determine how all these attributes are associated with one another. The graph structure defined by the set of N nodes and directed edges can be represented as an N-by-N adjacency matrix, which we denote as *adj_mat*, filled with binary values such that *adj_mat[i, j]=1* if there exists a directed edge from the node i to the node j and *adj_mat[i, j]=0* otherwise. For example, the adjacency matrix for the Bayesian network shown in Figure 2.1 is shown in Table 6.1

So we add the array attribute *adj_mat* to the BNET class.

	x_1	x_2	x_3	x_4	x_5	x_6
x_1	0	1	1	0	0	0
x_2	0	0	0	1	0	1
x_3	0	0	0	0	1	0
x_4	0	0	0	0	0	0
x_5	0	0	0	0	0	1
x_6	0	0	0	0	0	0

Table 6.1: The adjacency matrix for the Bayesian network in Figure 2.1.

All the nodes in the model are discrete, and each node has a specific set of values it can realize. The number of values a node x_i can realize is referred to as the size of x_i. We need to store the size of each node in the model within the class. We store the sizes in a list, denoted as *node_sizes*, where each entry corresponds to the node it indexes. For example, assume that the nodes in Figure 2.1 have the following sizes, $\{x_1 : 2, x_2 : 4, x_3 : 3, x_4 : 5, x_5 : 2, x_6 : 10\}$. Then the list *node_sizes* will be $[2, 4, 3, 5, 2, 10]$. So if we wish to find the size of the node x_4, we simply extract the fourth entry in the list *node_sizes*, which is 5. We add the list attribute *node_sizes* to the BNET class.

The next attribute of a Bayesian network is the conditional probability distributions associated with its nodes. In Section 6.4 we define the class CPD to represent conditional probability distributions, all we need to know now is that a Bayesian network has several conditional probability distributions, and it is therefore necessary to store a list *cpds* of CPD objects as an attribute in the BNET class.

The final attribute of a Bayesian network is its inference engine, as discussed at the beginning of this Chapter 6. A Bayesian network requires and inference engine, such as the junction tree inference engine, which is the JTREE class defined in Section 6.8, or a loopy belief inference engine, which is the LOOPY class defined in section 6.9. All inference engine classes are derived from the INF_ENGINE class, so we add the INF_ENGINE attribute *engine* to the BNET class.

6.2.2 Methods

The following paragraphs detail each of the methods within the BNET class.

BNET.init: The initialization method for the class.

To initialize a BNET object we need the following parameters:

1 *array: adj_mat.* An adjacency matrix defining the structure of the Bayesian network. If *adj_mat[i, j]=1*, then there exits a directed edge from node x_i to node x_j, *adj_mat[i, j]=0* otherwise.

2 *list: node_sizes.* A list of integers expressing the size of each node in the Bayesian network. If *node_sizes[i]* = k, then node x_i can take on one of k discrete values.

3 *list: cpds.* A list of CPD objects defining the conditional probability distributions associated with the Bayesian network.

The algorithm for BNET init method is shown below.

```
BNET.init(adj_mat, node_sizes, cpds)
  STEP 1: Store all the parameters as attributes.

    self.adj_mat <- adj_mat
    self.node_sizes <- node_sizes
    self.cpds <- cpds
```

BNET.learn_params_MLE: Estimates the parameters of the model from a set of fully observed training data.

To estimate the parameters of a BNET object we need the following input parameters:

1 *list: samples.* A nested list of fully observed samples, such that sample[t][i] is the observed integer value of node x_i in sample t.

The algorithm for the BNET learn_params_MLE method is shown below.

```
BNET.learn_params_MLE(samples)
  STEP 1: Iterate through the list of CPD's for this BNET.

    N <- The length of the list self.cpds.
    for i <- 0 to N-1

    STEP 2: Extract the samples that apply to the domain of
            CPD i from the parameter samples.

      local_samples <- The observed values for the variables
                       indexed in self.cpds[i].domain within
                       the parameter samples. For example if
                       self.cpds[i].domain = [0, 2], then
                       local_samples will be all the entries
                       from samples for nodes 0 and 2.

    STEP 3: Count the number of times each different
            combination of values appears in the list
            local_samples  and divide them by the total number
            of samples.

      T <- A zero initialized array with the dimensions
           specified in the attribute self.cpds[i].node_sizes.
```

```
            For instance if self.cpds[i].node_sizes = [2, 7],
            then T will be a two dimensional array with 2
            rows and 7 columns.

       M <- The length of the list samples.
       for j <- 0 to M-1
         sample = local_samples[j]

         T <- Increase the value of the entry indexed by sample
              in the array T by one. For instance, if T is
              a two-dimensional array, and sample=[2, 5], then
              increase the entry T[2, 5] by 1.
       end

       T <- Divide every entry in T by M.

     STEP 4: Set the entries of CPD i to the estimated values.

       self.cpds[i].T <- T
     end
```

BNET.learn_params_EM: Estimates the parameters of the model from a set of fully or partially observed training data via the EM algorithm.

To estimate the parameters of a BNET via the EM algorithm we need the following input parameters:

1 *list: samples.* A nested list of observed samples, such that sample[t][i] is either a value k meaning that node $x_i = k$ in sample t, or is None to indicate the the value of the node x_i is hidden in sample t.

2 *int: max_iter.* The maximum number of iterations the EM algorithm can perform before ending. The default is 10.

3 *float: thresh.* The maximum value the log likelihood of two consecutive iterations can differ by to conclude that the EM algorithm has converged. The default is 10^{-4}.

4 *boolean: exact.* This parameter selects which inference engine to use during the EM algorithm. The value TRUE selects the exact junction tree inference engine, and the value FALSE selects the approximate loopy belief propagation inference engine. The default is TRUE.

5 *int: inf_engine_iter.* If the approximate inference engine is to be used by the EM algorithm, indicated by the parameter exact being TRUE, then this value is the maximum number of iterations the the loopy belief propagation inference engine can perform before ending. The default is 10.

The algorithm for BNET learn_params_EM method is shown below.

```
BNET.learn_params_EM(samples)
  STEP 1: Iterate through the list of CPDs for this BNET and
          reset the parameters of each to one.

    N <- The length of the list self.cpds.
    for i <- 0 to N-1
      self.cpds[i].T <- (self.cpds[i].T * 0) + 1
    end

  STEP 2: Initialize the data required to determine when the EM
          algorithm must stop iterating and begin iterating.

    loglik <- 0
    prev_loglik <- $-2^{32}$
    converged <- FALSE
    num_iter <- 0

    while (converged is FALSE) and (num_iter is less than max_iter)

    STEP 2.1: Execute a single iteration of the EM algorithm
              and obtain the log likelihood of the training data
              under the model parameters after the iteration.

      loglik <- self.EM_step(samples, exact, inf_max_iter)

    STEP 2.2: Determine whether the algorithm has converged by
              comparing the current log likelihood with the
              log likelihood from the previous iteration.

      delta_loglik <- The absolute value of
                             (loglik-prev_loglik).

      if (delta_loglik <= thresh)
        converged <- TRUE
      end

      num_iter <- num_iter + 1
    end
```

BNET.EM_step: Performs one iteration of the EM algorithm on a BNET.

To execute an iteration of the EM algorithm on a BNET we need the following input parameters:

1 *list: samples.* A nested list of observed samples, such that sample[t][i] is either a value k meaning that node $x_i = k$ in sample t, or is None to indicate the the value of the node x_i is hidden in sample t.

2 *boolean: exact.* This parameter selects which inference engine to use during the EM algorithm. The value TRUE selects the exact junction tree inference engine, and the value FALSE selects the approximate loopy belief propagation inference engine. The default is TRUE.

3 *int: inf_max_iter.* If the approximate inference engine is to be used by the EM algorithm, indicated by the parameter exact being TRUE, then this value is the maximum number of iterations the the loopy belief propagation inference engine can perform before ending. The default is 10.

This method returns the following values:

1 *float: loglik.* The log likelihood of all the samples under the current model parameters.

The algorithm for BNET EM_step method is shown below.

```
BNET.EM_step(samples, exact, inf_max_iter)
  STEP 1: Initialize the specified inference engine with the
          current model parameters.

    self.init_inference_engine(exact, inf_max_iter)

  STEP 2: Iterate through the list of CPDs for this BNET and
          reset the expected sufficient statistics of each to
          zero.

    N <- The length of the list self.cpds.
    for i <- 0 to N-1
      self.cpds[i].ESS <- (self.cpds[i].ESS  * 0) + 1
    end

  STEP 3: Loop through the set of samples and process each one
          into the expected sufficient statistics of each CPD.
          Also, create a running sum of the log likelihood of all
          the samples.

    loglik <-0
    M <- The number of samples.
    for i <- 0 to M-1
```

```
      sample <- samples[i]

    STEP 3.1: Use sample i as evidence execute the sum-product
              algorithm based on the models current parameters.

      sample_loglik <- self.sum_product(sample)
      loglik <- loglik + sample_loglik

    STEP 3.2: Loop through list of CPDs in the model and update
              the expected sufficient statistics of each one.

      N <- The length of the list self.cpds.
      for j <- 0 to N-1
        cpd <- self.cpds[j]

      STEP 3.2.1: Obtain the marginal distribution over the
                  nodes in this CPD's domain and use it to
                  update the CPD's expected sufficient
                  statistics.

        expected_vals <-
          self.inference_engine.marginal_nodes(cpd.domain)

        cpd.update_ess(sample, expected_vals, self.node_sizes)
      end
    end

  STEP 4: Loop through every CPD in the model and replace
          the CPD's conditional probability table with its
          expected sufficient statistics, and reset the
          expected sufficient statistics to zero.

    N <- The length of the list self.cpds.
    for j <- 0 to N-1
      self.cpds[j].CPT <- self.cpds[j].ESS
      self.cpds[j].ESS <- self.cpds[j].ESS * 0
    end

  STEP 5: Return the log likelihood of the training data under
          the current model parameters.

    return loglik
```

6.3 The MRF class

The MRF class represents a Markov random field.

6.3.1 Attributes

Before we define the classes required to implement it, we identify the attributes that make up a Markov random field. From Section 2.2 we know that a Markov random field consists of

1 A set of N nodes.

2 A set of undirected edges connecting the nodes.

3 A set of maximal cliques extracted from the graph.

4 A set of potentials defined over the maximal cliques.

As with the BNET class defined in Section 6.2, we represent the entire graph structure of a Markov random field as an adjacency matrix *adj_mat*, except in this case when *adj_mat[i, j]=1* there exists an undirected edge between the nodes x_i and x_j, therefore *adj_mat[j, i]* must also equal 1. This means the the *adj_mat* attribute is a symmetrical matrix, and in practice we only need to store half of it. The full adjacency matrix corresponding to the graph shown in Figure 2.3 is shown in Table 6.2.

	x_1	x_2	x_3	x_4	x_5	x_6
x_1	0	1	1	0	0	0
x_2	1	0	0	1	1	1
x_3	1	0	0	0	1	0
x_4	0	1	0	0	0	0
x_5	0	1	1	0	0	1
x_6	0	1	0	0	1	0

Table 6.2: The adjacency matrix for the Markov random field in Figure 2.3.

As with the BNET class we add the array attribute *adj_mat* to the MRF class. Next we need to store the sizes of the nodes in the model. As with the BNET class we use a list, denoted *node_sizes*, to store the size of each node in the model, where *node_sizes[i]* is the size of the node x_i. We now add the list attribute *node_sizes* to the MRF class.

Next we need to deal with the cliques and potentials attributes of a Markov random field. These attributes are linked through a 1-to-1 relationship, and since a potential is defined over a clique, we shall assign a potential as an attribute to every clique. Cliques are handled by the CLIQUE class in Section 6.5, and potentials are handled by the POTENTIAL class in Section 6.6. We know that a Markov random field has several cliques, and just like the way we handled the CPD class for the BNET class, we simply assign a list *cliques* of CLIQUE objects as an attribute of the MRF class.

Finally, as with the BNET class, we need to assign an attribute to the MRF class for the inference engine, which could be either a JTREE or LOOPY object, both of which are type INF_ENGINE. Therefore, we add the INF_ENGINE attribute *engine* to the MRF class.

6.3.2 Methods

The following paragraphs detail each of the methods within the MRF class.

MRF.init: The initialization method for the class.
To initialize a MRF object we need the following parameters:

1 *array: adj_mat.* A symmetric adjacency matrix defining the structure of the Markov random field. If *adj_mat[i, j]=1*, then there exits an undirected edge between node x_i to node x_j.

2 *list: node_sizes.* A list of integers expressing the size of each node in the Markov random field. If *node_sizes[i]* = k, then node x_i can take on one of k discrete values.

3 *list: cliques.* A list of CLIQUE objects defining the cliques, and their potentials, associated with the Markov random field.

The algorithm for MRF init method is shown below.

```
MRF.init(adj_mat, node_sizes, cliques)
   STEP 1: Store all the parameters as attributes.

      self.adj_mat <- adj_mat
      self.node_sizes <- node_sizes
      self.cliques <- cliques
```

MRF.learn_params_MLE: Estimates the parameters of the model from a set of fully observed training data.
To estimate the parameters of a MRF object we need the following input parameters:

1 *list: samples.* A nested list of fully observed samples, such that sample[t][i] is the observed integer value of node x_i in sample t.

The algorithm for MRF learn_params_MLE method is shown below.

```
MRF.learn_params_MLE(samples)
   STEP 1: Iterate through the list of cliques for this MRF.

      N <- The length of the list self.cliques.
      for i <- 0 to N-1

      STEP 2: Extract the samples that apply to the domain of
              clique i from the parameter samples.
```

```
    local_samples <- The set of all samples for the variables
                     indexed in self.cliques[i].domain within
                     the parameter samples. For example if
                     self.cliques[i].domain = [0, 2], then
                     local_samples will be all the samples
                     for nodes 0 and 2.

  STEP 3: Count the number of times each unique combination
          of values appears in the list local_samples and
          divide them by the total number of samples.

    T <- A zero initialized array with the dimensions
         specified in the attribute self.cliques[i].node_sizes.
         For instance if self.cliques[i].node_sizes = [2, 7],
         then T will be a two dimensional array with 2
         rows and 7 columns.

    M <- The length of the list local_samples.
    for j <- 0 to M-1
      sample = local_samples[j]

      T <- Increase the value of the entry indexed by sample
           in the array T by one. For instance, if T is
           a two-dimensional array, and sample=[2, 5], then
           increase the entry T[2, 5] by 1.
    end

    T <- Divide every entry in T by M.

  STEP 4: Set the entries of the potential for clique i to
          the estimated values.

    self.cliques[i].unobserved_pot.T <- T
  end
```

MRF.learn_params_EM: Estimates the parameters of the model from a set of fully or partially observed training data via the EM algorithm.

To estimate the parameters of a MRF via the EM algorithm we need the following input parameters:

1 *list: samples.* A nested list of observed samples, such that sample[t][i] is either a value k

meaning that node $x_i = k$ in sample t, or is None to indicate the the value of the node x_i is hidden in sample t.

2 *int: max_iter.* The maximum number of iterations the EM algorithm can perform before ending. The default is 10.

3 *float: thresh.* The maximum value the log likelihood of two consecutive iterations can differ by to conclude that the EM algorithm has converged. The default is 10^{-4}.

4 *boolean: exact.* This parameter selects which inference engine to use during the EM algorithm. The value TRUE selects the exact junction tree inference engine, and the value FALSE selects the approximate loopy belief propagation inference engine. The default is TRUE.

5 *int: inf_max_iter.* If the approximate inference engine is to be used by the EM algorithm, indicated by the parameter exact being TRUE, then this value is the maximum number of iterations the the loopy belief propagation inference engine can perform before ending. The default is 10.

The algorithm for MRF learn_params_EM method is shown below.

```
MRF.learn_params_EM(samples, max_iter, thresh, exact,
                    inf_max_iter)
  STEP 1: Iterate through the list of cliques for this MRF and
          reset the parameters of each to one.

    N <- The length of the list self.cliques.
    for i <- 0 to N-1
      self.cliques[i].unobserved_pot.T <-
                    (self.cliques[i].unobserved_pot.T  * 0) + 1
    end

  STEP 2: Initialize the data required to determine when the
          EM algorithm must stop iterating and begin
          iterating.

    loglik <- 0
    prev_loglik <- $-2^{32}$
    converged <- FALSE
    num_iter <- 0

    while (converged is FALSE) and (num_iter is less than max_iter)

    STEP 2.1: Execute a single iteration of the EM algorithm
              and obtain the log likelihood of the training data
```

```
                 under the model parameters after the iteration.

        loglik <- self.EM_step(samples, exact, inf_max_iter)

     STEP 2.2: Determine whether the algorithm has converged by
               comparing the current log likelihood with the
               log likelihood from the previous iteration.

        delta_loglik <- The absolute value of
                           (loglik-prev_loglik).

        if (delta_loglik <= thresh)
          converged <- TRUE
        end

        num_iter <- num_iter + 1
     end
```

MRF.EM_step: Performs one iteration of the EM algorithm on a MRF.

To execute a iteration of the EM algorithm on a MRF we need the following input parameters:

1 *list: samples.* A nested list of observed samples, such that sample[t][i] is either a value k meaning that node $x_i = k$ in sample t, or is None to indicate the the value of the node x_i is hidden in sample t.

2 *boolean: exact.* This parameter selects which inference engine to use during the EM algorithm. The value TRUE selects the exact junction tree inference engine, and the value FALSE selects the approximate loopy belief propagation inference engine. The default is TRUE.

3 *int: inf_max_iter.* If the approximate inference engine is to be used by the EM algorithm, indicated by the parameter exact being TRUE, then this value is the maximum number of iterations the the loopy belief propagation inference engine can perform before ending. The default is 10.

This method returns the following values:

1 *float: loglik.* The log likelihood of all the samples under the current model parameters.

The algorithm for MRF EM_step method is shown below.

```
MRF.EM_step(samples, exact, inf_max_iter)
  STEP 1: Initialize the specified inference engine with the
          current model parameters.

    self.init_inference_engine(exact, inf_max_iter)

  STEP 2: Iterate through the list of cliques for this MRF and
          reset the expected sufficient statistics of each to
          zero.

    N <- The length of the list self.cliques.
    for i <- 0 to N-1
      self.cliques[i].ESS <- (self.cliques[i].ESS  * 0) + 1
    end

  STEP 3: Loop through the set of samples and process each one
          into the expected sufficient statistics of each
          clique and create a running sum of the log likelihood
          of all the samples.

    loglik <-0
    M <- The number of samples.
    for i <- 0 to M-1
      sample <- samples[i]

    STEP 3.1: Use sample i as evidence execute the sum-product
              algorithm based on the models current parameters.

      sample_loglik <- self.sum_product(sample)
      loglik <- loglik + sample_loglik

    STEP 3.2: Loop through every clique in the model and update
              its expected sufficient statistics.

      N <- The length of the list self.cliques.
      for j <- 0 to N-1
        clique <- self.cliques[j]

      STEP 3.2.1: Obtain the marginal distribution over the
                  nodes in this cliques domain and use it to
                  update the cliques expected sufficient
                  statistics.
        expected_vals <-
          self.inference_engine.marginal_nodes(clique.domain)

        clique.update_ess(sample, expected_vals,
                                 self.node_sizes)
      end
    end

  STEP 4: Loop through every clique in the model and replace
          the cliques potential with its expected sufficient
          statistics, and reset the expected sufficient
          statistics to zero.

    N <- The length of the list self.cliques.
    for j <- 0 to N-1
      self.cliques[j].unobserved_pot.T <- self.cliques[j].ESS
      self.cliques[j].ESS <- self.cliques[j].ESS * 0
    end

  STEP 5: Return the log likelihood of the training data under
          the current model parameters.

    return loglik
```

6.4 The CPD class

The CPD class represents the conditional probability distributions used in the definition of a Bayesian Network.

6.4.1 Attributes

Each conditional probability distribution has the following set of attributes:

1 It is associated with a unique node x_i in a Bayesian network, and the set of nodes $\mathbf{x_p}$ which are the parent nodes of the node x_i. We shall call the unified set $\{x_i, \mathbf{x_p}\}$ the *domain* of the conditional probability distribution.

2 A multi-dimensional table containing the values of the conditional probability distribution over the domain. We shall call this table the *conditional probability table.*

As an example, the conditional probability distribution $p(x_2|x_1)$ for the Bayesian network in Figure 2.2 has the domain $\{x_1, x_2\}$ and an example conditional probability table is shown in Table 2.2. We store the domain as a sorted list attribute *domain*, and the conditional probability table as a multi-dimensional array attribute *CPT*, which will have the number dimensions equal to the length of the domain list. We then associated each dimension of the array with node it indexes in the the ordered list. For instance, the first domain of the conditional probability table is associated with the node in first position in the domain list. As an example, consider Table 2.2, and its sorted domain list $\{x_1, x_2\}$. The first dimension, which is the rows of Table 2.2, corresponds to the different realizations of the first node in the domain list, the node x_1. The second dimension, which is the columns of Table 2.2, corresponds to the different realizations of the second node in the domain list, the node x_2.

For convenience we also store the sizes of the nodes in the conditional probability distributions domain in list attribute *fam_size*. Therefore, *fam_size[i]* is the size of the node stored in *domain[i]*. Thus, if a CPD has *domain*$= [x_2, x_5, x_6]$ and *fam_size* $= [2, 4, 3]$, then the node x_2 has the size 2, and the node x_6 has the size 3.

There is one final attribute to be added to this class, the attribute to store the expected sufficient statistics of the CPD, which are utilised when learning the parameters of a CPD from training data. In section 4.2 we saw that the expected sufficient statistics of a CPD is simply a multi-dimensional array with the same dimensions as the CPD. Therefore, we simply add the array attribute *ESS* to the class, and initialize it to the same dimensions as the the *CPT* attribute.

6.4.2 Methods

The following paragraphs detail each of the methods within the CPD class.

CPD.init: The initialization method for the class.

To initialize a CPD object we need the following parameters:

1 *int: node_id.* The index of the node in the Bayesian network this CPD is assigned to.

2 *list: node_sizes.* A list of integers expressing the size of each node in the entire Bayesian network.

3 *array: adj_mat.* An adjacency matrix defining the structure of the Bayesian network, which is the *adj_mat* attribute of the BNET class.

4 *array: CPT.* This is a multi-dimensional array of floats representing the conditional probability table associated with this conditional probability distribution. If it is not specified, we initialize it to the required dimensions and fill it with 1's.

The algorithm for CPD init method is shown below.

```
CPD.init(node_id, node_sizes, adj_mat, CPT)
  STEP 1: Determine the domain of the CPD, as well as the sizes
          of the nodes in the domain.
```

```
    self.domain <- A sorted list starting with node_id and
                   followed by the indices of the parents of
                   the node indexed by node_id, determined from
                   the adjacency matrix adj_mat.

    self.fam_size <- A list of the values in node_sizes
                     indexed by the entries in self.domain. For
                     example, if node_sizes=[2, 4, 7] and
                     self.domain=[0, 2], then
                     self.fam_size=[2, 7].

  STEP 2: Determine a CPT was defined, and if so save it as a
          attribute. If a CPT was not defined, initialize table
          of ones with the dimensions specified in the list
          self.fam_size and save it as an attribute.

    if CPT was specified
      self.CPT <- CPT
    else
      CPT <- An array of 1's with the dimensions specified
             in the attribute self.fam_size. For instance if
             self.fam_size = [2, 7], then CPT will be a two
             dimensional table with 2 rows and 7 columns, and
             all entries will be 1.
      self.CPT <- CPT
    end

  STEP 3: Set the expected sufficient statistics of the CPD to
          a zero value array the same shape as CPT.

    self.ESS <- self.CPT * 0
```

CPD.convert_to_pot: Returns a POTENTIAL representation of a CPD.

No parameters are required to convert a CPD to a POTENTIAL object. This method returns the following values,

1 *POTENTIAL: pot.* This is the desired POTENTIAL created from the CPD object.

The algorithm for the CPD convert_to_pot method is shown below.

```
CPD.convert_to_pot(domain)
  STEP 1: Initialize a new POTENTIAL with the same attributes
          as this CPD and return it.

   pot <- POTENTIALS.init(self.domain, self.fam_size, self.CPT)

   return pot
```

CPD.update_ess: Updates the CPD's expected sufficient statistics.

This method is part of the EM algorithm used to learn the parameters of a BNET. To update a CPD objects expected sufficient statistics we need the following parameters:

1 *list: sample.* A sample of observations with its length equal to the number of nodes in the model this CPD is part of. The value of sample[i] is either None, to represent that the node x_i is hidden, or an integer value k to represent the sample $x_i = k$.

2 *POTENTIAL: expected_vals.* A POTENTIAL object containing the marginal probability distribution over the hidden nodes in the sample.

3 *list: node_sizes.* A list of integers expressing the size of each node in the model this CPD is part of. If *node_size[i]=k*, then the node x_i can take on k possible values.

The algorithm for the CPD update_ess method is shown below.

```
CPD.update_ess(sample, expected_vals, node_sizes)
  STEP 1: Determine which nodes in the sample are hidden.

    hidden <- A list of integers indexing which entries in the
              parameter samples are None. For example, if
              samples = [None, 1, None] then hidden <- [0,2].

  STEP 2: If none of the nodes in this CPD's domain are hidden
          in the input sample, then add the expected values to
          the expected sufficient statistics, otherwise reshape
          the expected values to the correct dimensions before
          adding it to the expected sufficient statistics.

    if hidden is not a subset of self.domain
      self.ess <- self.ess + expected_vals.T
    else
      expected_vals <- Reshape expected_val to the same
```

```
                        dimensions as self.ess. Insert zeros
                        into new entries.

    self.ess <- self.ess + expected_vals.T
  end
```

6.5 The CLIQUE class

The CLIQUE class represents both a clique in a Markov random field, and the potential attached to it. It is also used to store separator potentials and information about neighbouring cliques for the junction tree and loopy belief propagation algorithms.

6.5.1 Attributes

Each clique has the following attributes

1 The set of nodes of the clique, which we shall call the *domain* of the clique.

2 A potential associated with the clique.

First, for simplicity we assign an unique ID number to each node in the model, thus we introduce the integer attribute *id* to the CLIQUE class. The domain of the clique is stored in a sorted list *domain*, similar to the domain list in the CPD class. Potentials are represented by the POTENTIAL class defined in Section 6.6, and each clique has one potential assigned to it. However, it is useful to assign two potentials to each clique, one representing the observed version of the potential and one representing the unobserved version of the potential. We need this two representations since we lose information about the original unobserved potential when we enter evidence into to it. For example, assume we have the potential $\psi(x_1, x_2, x_3)$, and we observed that $x_3 = 1$. If we now enter this evidence into the potential, we obtain the new potential $\psi^E(x_1, x_2)$ which no longer includes information about the node x_3. If we now observe $x_3 = 0$, we cannot enter this evidence into $\psi^E(x_1, x_2)$, we need the original potential $\psi(x_1, x_2, x_3)$. So for each clique we store the original potential, and the observed working potential. If we observe new evidence, we simply copy the original potential into the working potential and enter that evidence into the working potential. So we add the POTENTIAL attributes *observed_pot* and *unobserved_pot* to the CLIQUE class.

The next attribute that must be added to this class, one that stores the messages from a clique to its neighbouring cliques. When executing the junction tree or loopy belief algorithms, we need to store the messages, i.e. the separator potentials, and we store these separator potentials in the CLIQUE class. The separator potential between a clique and one of its neighbouring cliques depends on the set of variable nodes that separate the clique from its neighbour. Thus together with these separator potentials we store the set of nodes that separate a clique from each one of its neighbours. We combine the separator potential and separator sets into a key-indexed nested list attribute, denoted as *nbrs*. For every clique $\mathbf{X_j}$ neighbouring the clique $\mathbf{X_i}$, the clique $\mathbf{X_i}$'s *nbr* attribute has an entry $nbr[j][0]$ which will be a list of the indices of the variable nodes separating the cliques X_i and X_j, so it will be a list of the indices of the set $\{X_i \cap X_j\}$. There is also an entry $nbr[j][1]$ that is a POTENTIAL object representing the separator potential between the cliques X_i and X_j, meaning it represents the potential $\phi(X_i \cap X_j)$. Consider the example junction tree shown in Figure 3.26(a). The clique $\psi_2(x_2, x_3, x_5)$ has the three neighbouring clique nodes $\psi_2(x_2, x_4, x_5)$, $\psi_3(x_2, x_3)$ and $\psi_4(x_2, x_5, x_6)$. Each entry in clique $\psi_2(x_2, x_3, x_5)$'s *nbrs* attribute is listed below.

1 $nbrs[1][0] = [2, 3]$.

2 $nbrs[1][1]$ is a POTENTIAL object representing the separator potential $\phi_1(x_2, x_3)$.

3 $nbrs[3][0] = [2]$.

4 $nbrs[3][1]$ is a POTENTIAL object representing the separator potential $\phi_2(x_2)$.

5 $nbrs[4][0] = [2, 5]$.

6 $nbrs[4][1]$ is a POTENTIAL object representing the separator potential $\phi_3(x_2, x_5)$.

The use of the *nbrs* attribute will become clear in Section 6.8 when the implementation of the junction tree algorithm is discussed.

There is one final attribute to be added to this class, the attribute to store the expected sufficient statistics, which is utilised when learning the potential of a clique from training data. In section 4.2 we saw that the expected sufficient statistics of a clique potential is simply a multi-dimensional array with the same dimensions as the cliques potential. Therefore, we simply add the array attribute *ESS* to the class, and initialize it to the same dimensions as the the T attribute of the class's POTENTIAL attribute *unobserved_pot*.

6.5.2 Methods

The following paragraphs detail each of the methods within the CLIQUE class.

CLIQUE.init: The initialization method for the class.
To initialize a CLIQUE object we need the following parameters:

1 *int: id.* A unique identification integer.

2 *list: domain.* A list of values indexing the nodes in this cliques domain.

3 *list: sizes.* A list of integers expressing the size of each node in the domain of this clique. If *size[i]=k*, and *domain[i]=j*, then the node x_j can take on k possible values.

4 *array: T.* This is a multi-dimensional array of floats representing the values of this cliques potential. If it is not specified, we initialize it to the required dimensions and fill it with 1's.

The algorithm for the CLIQUE init method is shown below.

```
\begin{lstlisting}[language=python]
CLIQUE.init(id, domain, sizes, T)
  STEP 1: Save the required input parameters as attributes.
    self.id <- id
    self.domain <- domain

  STEP 2: Create the unobserved potential which will not be
          altered by any of the inference algorithms and can
          only be changed explicitly.

    self.unobserved_pot <- POTENTIAL(self.domain, self.size, T)

  STEP 3: Create the observed potential to be unobserved as
          well. This potential will be working potential used
          by the inference engines when executing the
          sum-product or max-sum algorithms.

    self.observed_pot <- POTENTIAL(self.domain, self.size, T)

  STEP 4: Set the expected sufficient statistics of the clique
          to a zero value array the same shape as the
          unobserved potentials look up table.

    self.ESS <- self.unobserved_pot.T * 0

  STEP 5: Initialize the neighbours attribute to an empty key
          indexed list, since it can only be properly
          initialized when we know which cliques neighbour
          this clique.

    self.nbrs <- An empty key indexed list.
```

CLIQUE.enter_evidence: This method alters a clique's observed potential to reflect observed evidence.

To enter evidence into a CLIQUE objects potential we need the following parameters:

1 *list: evidence.* A list of observed evidence with the length equal to the number of nodes in the domain of this clique. The value of evidence[i] is either None, to indicate that the node indexed by self.domain[i] is unobserved, or an integer value k to represent the observation $x_{domain[i]} = k$.

2 *boolean: maximize.* This value is TRUE if this clique is going to be used in a max-sum inference algorithm and is FALSE otherwise. The default is FALSE.

The algorithm for the CLIQUE enter_evidence method is shown below.

```
CLIQUE.enter_evidence(evidence, maximize)
  STEP 1: Initialize the observed potential as a copy of
          the unobserved potential, and if the clique
          is being used in a max-sum algorithm then take
          the logarithm of the observed potential values.

    self.observed_pot <- self.unobserved_pot

    if maximize == TRUE
      self.observed_pot <- log(self.observed_pot)
    end

  STEP 2: Enter evidence into the cliques potential.

    self.observed_pot.enter_evidence(evidence)
```

CLIQUE.init_sep_pots: The method to initialize the separator potentials between a clique and its neighbouring cliques.

To initialize a CLIQUE object's separator potentials the following parameters are required:

1 *list: node_sizes.* A list of integers expressing the size of each node in the model. If *node_size[i]=k*, then the node x_i can take on k possible values.

2 *list: onodes.* A list of integers indexing all the observed nodes in the model.

3 *boolean: maximize.* This value is TRUE if this clique is used in a max-sum inference algorithm and is FALSE otherwise. The default is FALSE.

The algorithm for the CLIQUE init_sep_pots method is shown below.

```
CLIQUE.init_sep_pots(node_sizes, onodes, maximize)
  STEP 1: Iterate through the keys of the self.nbrs attribute.

    keys <- A list of the keys which index the self.nbrs
            attribute.

    N <- The length of the list keys.
    for i <- 0 to N-1
      key <- keys[i]
```

```
STEP 1a: Determine the domain and node sizes of the
         separator potential between this clique and the
         clique indexed by key, and use these attributes to
         create a new POTENTIAL object.

  sep_pot_domain <- A copy of the separator potentials
                    domain, which is stored in self.nbrs[key][0]

  node_sizes <- Set every value in node_sizes indexed by
                the values in onodes to 1. For instance,
                if node_sizes=[2, 5, 3], and onodes=[0, 2],
                then node_sizes will become [1, 5, 1].

  sep_pot_sizes <- A list of the values in node_sizes
                   indexed by the entries in sep_pot_domain.
                   For example, if node_sizes=[2, 4, 7] and
                   sep_pot_domain=[1, 2], then
                   sep_pot_sizes=[4, 7].

  sep_pot_T <- A multi-dimensional array of 1's with the
               dimensions specified in the list sep_pot_sizes.
               For example, if sep_pot_sizes=[4, 7], then
               sep_pot_T will be a 2-dimensional table with
               4 rows and 7 columns.

  if maximize == TRUE
    sep_pot_T <- sep_pot_T * 0
  end

  sep_pot = POTENTIALS.init(sep_pot_domain, sep_pot_sizes,
                            sep_pot_T)

STEP 1b: Store the new potential in the self.nbrs list.

  self.nbrs[key][1] <- sep_pot
```

CLIQUE.update_ess: Updates the clique's expected sufficient statistics.

This method is used as part of the EM algorithm. To update a CLIQUE objects expected sufficient statistics we need the following parameters:

1 *list: sample.* A sample of observations with its length equal to the number of nodes in the

model. The value of sample[i] is either None, to represent that the node x_i is hidden, or an integer value k to represent the sample $x_i = k$.

2 *POTENTIAL: expected_vals.* A POTENTIAL object containing the marginal probability distribution over the hidden nodes in the sample.

3 *list: node_sizes.* A list of integers expressing the size of each node in the model. If *node_size[i]=k*, then the node x_i can take on k possible values.

The algorithm for the CLIQUE update_ess method is shown below.

```
CLIQUE.update_ess(sample, expected_vals, node_sizes)
  STEP 1: Determine which nodes in the sample are hidden.

    hidden <- A list of integers indexing which entries in the
              parameter samples are None. For example, if
              samples = [None, 1, None] then hidden <- [0,2].

  STEP 2: If none of the nodes in this clique's domain are
          hidden in the input sample, then add the expected
          values to the expected sufficient statistics,
          otherwise reshape the expected values to the correct
          dimensions before adding it to the expected
          sufficient statistics.

    if hidden is not a subset of self.domain
      self.ess <- self.ess + expected_vals.T
    else
      expected_vals <- Reshape expected_val to the same
                       dimensions as self.ess. Insert zeros
                       into new entries.
      self.ess <- self.ess + expected_vals.T
    end
```

6.6 The POTENTIAL class

The POTENTIAL class represents the potential attached to a clique. Unlike the CPD class, the POTENTIAL class has methods allowing arithmetical operation to be performed between POTENTIAL objects.

6.6.1 Attributes

We know from Section 2.2.2 that every potential has the following attributes:

1 A clique over which the potential is defined, the domain of the potential.

2 Every node in the potential's domain has a size.

3 A multi-dimensional array holding the values of the potential.

We store the domain as a sorted list *domain*, and the potential values array as as a multi-dimensional array *T*, with the number dimensions equal to the length of the *domain* list. We then associate each dimension of the array with the node it indexes in the the ordered list. For instance, the first domain of the table *T* is associated with the node in first position in the *domain* list. As an example, consider the potential shown in Table 2.4, and its sorted domain list $\{x_2, x_3\}$. The first dimension, the rows of Table 2.4, corresponds to the different values the first node in the domain list, the node x_2, can realize. The second dimension, the columns of Table 2.4, corresponds to the different values the second node in the domain list, the node x_3, can realize.

We also store the sizes of the nodes that are in the domain of the potential, in the list attribute *node_sizes*. Therefore, *node_sizes[i]* is the size of the node stored in *domain[i]*. Thus, if a potential has *domain*= $[x_1, x_3, x_4]$ and *node_sizes* $= [3, 7, 2]$, then the node x_3 has the size 7, and the node x_4 has the size 2.

6.6.2 Methods

The following paragraphs detail each of the methods within the POTENTIAL class.

POTENTIAL.init: The initialization method for the class.

To initialize a POTENTIAL object we need the following parameters:

1 *list: domain.* A list of values indexing the nodes in this potentials domain.

2 *list: sizes.* A list of integers expressing the size of each node in the domain of this potential. If *size[i]=k*, and *domain[i]=j*, then the node x_j can take on k possible values.

3 *array: T.* A multi-dimensional array of floats representing the values of this potential. If it is not specified, we initialize it to the required dimensions and fill it with 1's.

The algorithm for the POTENTIAL init method is shown below.

```
POTENTIAL.init(domain, sizes, T)
  STEP 1: Save the required input parameters as attributes.
    self.domain <- domain
    self.sizes <- sizes

  STEP 2: Determine whether the parameter $T$ has been
          specified. If it has been specified then save it as
          an attribute, and if it has not then create a table
          with the dimensions specified in the parameter *sizes*
```

```
            and save it as an attribute.

      if T has been specified
        self.T <- T
      else
        self.T <- An array of 1's with the dimensions specified
                  in the attribute self.sizes. For instance if
                  self.sizes = [2, 7], then self.T will be a two
                  dimensional table with 2 rows and 7 columns,
                  and all entries will be 1.
      end
```

POTENTIAL.arithmetic: This method implements the adding/subtracting/dividing/multiplying of potentials together.

To perform arithmetic on two POTENTIAL objects we need the following parameters:

1 *POTENTIAL: pot.* The other potential object which we are arithmetically combining with this one. The domain of pot must be a subset of the domain of this potential.

2 *char: op.* A character indicating which arithmetic operation to perform. The options are: '+' for addition, '-' for subtraction, '*' for multiplication and '/' for division. The default is '+'.

The algorithm for the POTENTIAL arithmetic method is shown below.

```
POTENTIAL.arithmetic(pot, op='+')
  STEP 1: Replicate the dimensions of the input potential's
          table in such away that its domain matches that of
          this potential.

    pos <- A list of the indices in self.domain where the entries
           in pot.domain appear. For instance, if
           self.domain = [1, 4, 5], and pot.domain = [1, 5],
           then pos = [0, 2].


    sz <- A list of ones with the same length as self.sizes.

    sz <- Substitute the values in sz indexed by the values in
          pos with the values in the list pot.sizes. For example,
          if sz = [1, 1, 1, 1], and pos = [0, 2], and
          pot.sizes = [4, 3] then sz will become
```

```
         sz = [ 4, 1, 3, 1].

    Ts <- The table pot.T reshaped to the dimensions specified
          in sz.

    N <- The length of the list self.sizes
    for i <- 0 to N-1
      if i is not in pos
        Ts <- Replicate the i'th dimension of Ts the number of
              times specified in self.sizes[i]. So if i=3, and
              self.sizes[3]=4 then replicate the 3rd dimension of
              Ts 4 times.
      end
    end

  STEP 2: Now that the input potential has the same domain and
          size as this potential, determine which mathematical
          operation to perform and execute it.

    if op == '+'
      self.T <- self.T + Ts
    elseif op == '-'
      self.T <- self.T - Ts
    elseif op == '*'
      self.T <- self.T * Ts
    elseif op == '/'
      self.T <- self.T / Ts
    end
```

POTENTIAL.enter_evidence: This method introduces evidence into a potential.
To modify a POTENTIAL object to reflect observed evidence we need the following parameter:

1 *list: evidence.* A list of observed evidence with the length equal to the number of nodes in the POTENTIALS domain. The value of evidence[i] is either None, to represent that the node $x_{POTENTIAL.domain[i]}$ is unobserved, or an integer value k to represent the observation $x_{POTENTIAL.domain[i]} = k$.

The algorithm for the POTENTIAL enter_evidence method is shown below.

```
POTENTIAL.enter_evidence(evidence)
  STEP 1: Enter the evidence into the potential by taking
```

```
          appropriate slices of its dimensions.

    N <- The length of the list evidence.
    for i <- 0 to N-1
      if evidence[i] is not None
        self.T <- Replace dimension i in self.T with the slice
                  indexed by evidence[i] in dimension i. For
                  example if i=2, and evidence[i] = 5, and self.T
                  has 3 dimensions, then
                  self.T <- self.T[:, 5, :].
      end
    end

  STEP 2: Update the list of node sizes to reflect that the
          nodes which have been observed have size 1.

    odom_ndx <- A list of the indices of all the nodes in
                evidence which are not None. For example
                if evidence = [None, 3, 5, None], then
                odom_ndx = [1, 2].

    self.sizes <- Replace every entry indexed by the list
                  odom_ndx with the value 1. For instance,
                  if self.sizes = [2, 4, 7, 3], and
                  odom_ndx = [1, 2], then self.sizes would
                  become [2, 1, 1, 3].
```

POTENTIAL.marginalize_pot: Marginalizes a potential over a domain.
To marginalize a POTENTIAL object we need the following parameters:

1 *list: onto* A list of integers indexing the marginal nodes. This list is therefore the domain of the marginalized return potential. This list of integers must be a subset of this potential's current domain.

2 *boolean: maximize.* This value is FALSE if we wish to marginalize this potential onto the domain specified in the parameter onto, and is TRUE if we wish to maximize this potential onto the domain specified in the parameter onto. The default value is FALSE.

This method returns the following values:

1 *POTENTIAL: small_pot.* The potential after marginalization or maximization. This return potential has the domain specified in the parameter *onto*.

2 *key indexed list: maximizers.* This structure is used to determine the backtracking information in the max-sum algorithm. It has the structure

1 maximizer[key] = [dependants, argmax].

2 key: Is the index of a node.

3 dependants: Is a list of nodes that gave rise to the maximum value of the node x_{key}.

4 argmax: The values of the nodes indexed in the list dependants which gave rise to the maximum value of the node x_{key}.

The algorithm for the POTENTIAL marginalize_pot method is shown below.

```
POTENTIAL.marginalize_pot(onto, maximize=False)
  STEP 1: Determine the variable nodes within the potential
          that must be marginalized out.

    sum_over <- The set difference between the list self.domain
                and the parameter list onto. Therefore, this
                list indexes the nodes which need to be
                marginalized/maximized out of the potential.

  STEP 2: Make a copy of this potentials table to marginalize
          or maximize for the new potential.

        small_T <- A copy of self.T.

  STEP 3: If we are to sum the potential, jump to STEP
          3.1, and if we wish to maximize, jump to STEP 3.2.

    if maximize is FALSE

      STEP 3.1: Sum every dimension in small_T that corresponds
                to a node indexed in sum_over.

        N <- The length of the list sum_over.
        for i <- 0 to N-1
          pos <- The position of the node sum_over[i] in the
                 self.domain list, which corresponds to the
                 dimension in the array self.T that we need to
                 marginalize over. If sum_over[i] = 2, and
                 self.domain = [0, 2, 3], pos = 1.

          small_T <- Sum the dimension indexed pos in small_T.
                     This will decrease the number of
```

```
                        dimensions in small_T by 1. For example, if
                        pos=2, sum over the second dimension of
                        small_T.
        end
    else

      STEP 3.2: Maximize over every dimension in small_T that
                corresponds to a node indexed in sum_over.
                While maximizing, store the values of the
                unobserved nodes indexed in the list sum_over
                which give rise to the maximum values of the
                domain made up of the nodes indexed in the list
                onto.

        dependants <- The nodes in self.domain which are not
                      observed.

        N <- The length of the list sum_over.
        for i <- 0 to N-1
          if sum_over[i] is in dependants
            Remove sum_over[i] from the list 'dependants.
          end

          pos <- Find the position of the node sum_over[i] in the
                 self.domain list. This position corresponds to
                 the dimension in the array small_T that we need
                 to maximize over.

          argmax <- A list of the indices of the maximum values
                    in the dimension indexed by pos in the
                    table small_T. For example, if pos=2, find
                    the indices of the maximum values in the
                    second dimension of small_T.

        maximizers[sum_over[i]] <- [dependants, argmax]

        small_T <- Maximize the dimension indexed pos in
                   small_T. This will decrease the number of
                   dimensions in small_T by 1.
      end
    end

    ns <- A list of sizes of the nodes indexed in the list
```

```
          onto. For example, if ns[i]=2 then
           the node indexed by onto[i] has the size 2.

  STEP 4: Create the new potential obtained by marginalizing
          or maximizing this potential and return it as output.

    small_pot <- POTENTIALS.init(onto, ns, small_T)

    return [small_pot, maximizers]
```

6.7 The INF_ENGINE class

The INF_ENGINE class is an abstract base class from which the inference engine classes JTREE and LOOPY are derived. INF_ENGINE has no attributes or methods. This class allows us to add an inference engine attribute to a MRF or BNET model, without having to know in advance whether it will be a LOOPY or JTREE inference engine. This class also allows us to easily extend our software with other inference engines, as long as they are derived from INF_ENGINE.

6.8 The JTREE class

The JTREE class, which is derived from the INF_ENGINE class, is used to perform exact probabilistic inference on MRF and BNET objects using the junction tree algorithm. We refer to this class as an inference engine.

6.8.1 Attributes

The data required to execute the junction tree sum-product and max-sum algorithms is summarized by the following attributes:

1 A model, either, a Bayesian network or, a Markov random field.

2 A junction tree consisting of clique nodes and undirected edges between them.

3 A set of triangulated cliques associated with the nodes in the junction tree.

4 A forward and backward message passing order.

To perform the junction tree algorithm on a model, we need the information associated with the model. We accomplish this by assigning the model of the inference engine as an attribute of the inference engine. This creates a two-way flow of information between the model object and the inference engine object. Therefore, we include the attribute *MODEL: model* in the JTREE class.

Next we need to represent the structure of the junction tree extracted from the model, and we do this with a simple adjacency matrix attribute, *array: adj_mat*. If *adj_mat[i, j] = 1* then there exists an undirected edge between clique node ψ_i and clique node ψ_j in the junction tree.

We use an ordered list, *list: tri_cliques*, of CLIQUE objects to store the triangulated cliques. The order of the list *tri_cliques* is related to the *adj_mat* attribute, i.e. entry i in the *tri_cliques* list is the CLIQUE object for clique node i in the adjacency matrix attribute *adj_mat*.

We use the depth-first search algorithm to determine the forward and backward message passing order. We store the forward pass order as a list of integers indexing the nodes of the junction tree, in the attribute *list: post_order*. We store the values in the list in such a way that the first entry in *post_order* indexes the first node to visit during the forward pass, and the second entry indexes the second node to visit, and so on. Similarly, we store the backward pass order in the attribute *list: pre_order*. However, we cannot just store the order in which to traverse the nodes. For each node x_i in the the message passing order we need to store the indices of the nodes to which the node x_i passes its messages. For the forward pass order we assign an attribute *list: post_order_parents*. This attribute is a list of lists, where *post_order_parents[i]* contains a list indexing the nodes that the node indexed by *pre_order[i]* passes its messages to during the forward pass. Similarly we assign the attribute *list: pre_order_children* for the backward pass order, where *pre_order_children[i]* contains a list indexing the nodes that the node indexed by *pre_order[i]* pass its messages to during the backward pass.

6.8.2 Methods

The following paragraphs detail each of the methods within the JTREE class.

JTREE.init: The initialization method for the class.

To initialize a JTREE object we need the following parameter:

1 *MODEL: model.* The model we wish to perform junction tree inference on.

This method simply selects the correct method to initialize the junction tree based on which type of model, Bayesian network or Markov random field, was passed to it. The algorithm for the JTREE init method is shown below.

```
JTREE.init(model)
  STEP 1: Determine which model the junction tree inference
          engine is coupled to and initialize it for that
          model.

    if model is a BNET object
      self.init_for_bnet(model)
    else
      self.init_for_mrf(model)
    end
```

JTREE.init_for_mrf: Initializes the junction tree inference for a Markov random field.

The initialization phase of the junction tree algorithm involves calculating the junction tree, the triangulated cliques associated with it, and the message passing order to be used in the propagation phase. To initialize the junction tree algorithm for a Markov random field we need the following parameter:

1 *MRF: mrf*. The Markov random field we wish to perform junction tree inference on.

The algorithm for the JTREE init_for_mrf method is shown below.

```
JTREE.init_for_mrf(mrf)
  STEP 1: Save the required input parameters as attributes.

    self.model <- mrf

  STEP 2: Triangulate the Markov random field to obtain the
          triangulated cliques and the corresponding junction
          tree.

    [self.jtree, clq_doms] <- triangulate(mrf.model_graph,
                                          self.model.node_sizes)

  STEP 3: Multiply each POTENTIAL from the original graph into
          the triangulated clique POTENTIAL it has been
          assigned to.

    self.tri_cliques <- An empty list which will be used to
                        store the triangulated cliques for
                        the junction tree.

    N <- The length of the list clq_doms.
    for i <- 0 to N-1
      ns <- A list of the sizes of the variable nodes indexed
            by the list clq_doms[i]. For example, if ns[3]=2 then
            the node indexed by clq_doms[i][3] has the size 2.

      tri_clique <- CLIQUE.init(clq_doms[i], ns)

      M <- The number of user defined cliques from the original
           graph, which is the length of the list
           self.model.cliques.

      for j <- 0 to M-1
```

```
        inter <- A list indexing all the nodes which appear in
                 the intersection between the list clq_doms[i] and
                 the list self.model.cliques[j].domain.

        if inter is not an empty list
          temp_pot <- self.model.cliques[j].\
                      unobserved_pot.marginalize_pot(inter)[0]
          tri_clique.unobserved_pot.arithmetic(temp_pot, '*')
        end
      end
      self.tri_cliques <- Append tri_clique to this list.
    end

  STEP 7: Initialize the storage for the separator potentials
          connected to each CLIQUE.

    y <- A nested list containing the indices of 1's in each row
         in self.jtree. Therefore y[i] is a list of the indices of
         all the columns in row i that contain the value 1.

    N <- The length of the list y.
    for i <- 0 to N-1
      M <- The length of the list y[i].
      for j <- 0 to M-1
        sep <- A list indexing the variable nodes in the
               intersection between triangulated cliques i and
               j. This is the intersection between the lists
               self.tri_clique[i].domain and
               self.tri_clique[j].domain.

        self.tri_cliques[i].nbrs[j] <- [sep, None]
      end
    end

  STEP 8: Execute the depth first search algorithm on the
         JTREE to determine the message passing
         order.

    N <- The length of the list self.tri_cliques.

    [self.jtree, self.pre_order] <- dfs(self.jtree, N)

    self.post_order <- The reverse of the self.pre_order list.
```

JTREE.init_for_bnet: Initializes the junction tree inference for a Bayesian network.

The initialization phase of the junction tree algorithm involves calculating the junction tree, the triangulated cliques associated with it, and the message passing order to be used in the propagation phase. To initialize the junction tree algorithm for a Bayesian network we need the following parameter:

1 *BNET: bnet.* The Bayesian network we wish to perform junction tree inference on.

The algorithm for the JTREE init_for_bnet method is shown below.

```
JTREE.init_for_bnet(bnet)
  STEP 1: Save the required input parameters as attributes.

     self.model <- bnet

  STEP 2: Moralize the Bayesian network.

     self.model_graph <- moralize(self.model.model_graph)

  STEP 3: Triangulate the moral graph to obtain the domains
          of the triangulated cliques and the corresponding
          junction tree.

     [self.jtree, clq_doms] <- triangulate(self.model_graph,
                                            self.model.node_sizes)

  STEP 4: Assign each of the BNET's CPD's to one of the
          triangulated cliques.

    N <- The length of the list self.model.cpds.

     self.clq_ass_to_node <- A list of length N. This list will
                              store the assignments of nodes to
                              triangulated cliques. For example,
                              if clq_ass_to_node[i]=3, then
                              variable node i within the BNET
                              has been assigned to triangulated
                              clique 3 in the JTREE.

     for i <- 0 to N-1
       fam <- A list containing the indices of all the nodes
              connected to node i in the moralized graph.
```

```
      clqs_containing_fam <- A list containing the indices of
                             all the triangulated cliques which
                             have the set of nodes indexed in
                             fam as a subset.

      c <- The index of the 'lightest' triangulated clique in
           the list clqs_containing_fam. The weight of a clique is
           the sum of the sizes of all the nodes in the clique.

      self.clq_ass_to_node[i] <- c
    end

  STEP 5: Convert the BNET's CPDs into POTENTIALs.

    node_pots <- An empty list of length N.

    for i <- 0 to N-1
      fam <- A list containing the indices of all the nodes
             connected to node i in the original directed
             graph.

      node_pots[i] <- self.model.cpds[i].convert_to_pot(fam)
    end

  STEP 6: Multiply each POTENTIAL converted from a CPD into
          the POTENTIAL for the triangulated clique it has
          been assigned to.

    N <- The length of the list clq_doms.
    self.tri_cliques <- An empty list.

    for i <- 0 to N-1
      ns <- The sizes of all the nodes indexed in clq_doms[i].
            For example, if ns[j]=5 then the node  indexed
            clq_doms[i][j] has the size of 5.

      tri_clique <- CLIQUE.init(i, clq_doms[i], ns)

      node_ass <- A list containing all the indices in
                  self.clq_ass_to_node where the entry i appears.

      M <- The length of the list node_ass
      for j <- 0 to M-1
```

```
        tri_clique.unobserved_pot.arithmetic(node_pots[j], '*')
      end

      self.tri_cliques[i] <- tri_clique
    end

  STEP 7: Initialize the storage for the separator potentials
          connected to each *CLIQUE*.

    y <- A nested list containing the indices of 1's in each row
         in *self.jtree*. Therefore *y[i]* is a list of the indices of
         all the columns in row $i$ that contain the value 1.

    N <- The length of the list $y$.
    for i <- 0 to N-1
      M <- The length of the list *y[i]*.
      for j <- 0 to M-1
        sep <- A list indexing the variable nodes in the
               intersection between triangulated cliques $i$ and
               $j$. This is the intersection between the lists
               *self.tri_clique[i].domain* and
               *self.tri_clique[j].domain*.

        self.tri_cliques[i].nbrs[j] <- [sep, None]
      end
    end

  STEP 8: Execute the depth first search algorithm on the
          *JTREE* to determine the message passing order.

    N <- The length of the list *self.tri_cliques*.

    [self.jtree, self.pre_order] <- dfs(self.jtree, N)

    self.post_order <- The reverse of the *self.pre_order* list.
```

JTREE.sum_product: This method executes the sum-product algorithm on the junction tree initialized by either the JTREE.init_for_mrf or JTREE.init_for_bnet methods. To execute the sum-product algorithm on the junction tree we need only the following parameter:

1 *list: evidence.* A list of observed evidence with the length equal to the number of variable nodes in the MODEL that this junction tree was initialized from. The value of evidence[i] is

either None, to represent that the node x_i is unobserved, or an integer value k to represent the observation $x_i = k$.

The algorithm for the JTREE sum_product method is shown below.

```
JTREE.sum_product(evidence)
  STEP 1: Enter the evidence into the clique potentials.

    onodes <- A list containing the indices of the entries in
              the list evidence which are not None. Therefore, if
              evidence=[None, 1, None, 2], onodes=[1, 3].

    N <- The length of the list self.tri_cliques.
    for i <- 0 to N-1
      self.tri_cliques[i].enter_evidence(evidence)

      self.tri_cliques[i].init_sep_pots(self.model.node_sizes,
                                        onodes, FALSE)
    end


  STEP 2.1: Belief propagation: The forward pass. Start by
            iterating through the forward message passing
            order stored in the list self.post_order.

    N <- The length of the list self.post_order.
    for i <- 0 to N-1
      n <- self.post_order[i]

    STEP 2.2: Iterate through the list of nodes which node n
              has to send messages to.

      po_parents <- self.post_order_parents[n]
      M <- The length of the list po_parents.
      for j <- 0 to M-1
        p <- po_parents[j]

      STEP 2.3: Calculate the message from clique node n to
                clique node p by marginalizing the potential of
                clique node n onto the variable nodes
                separating clique nodes n and p.

        dom <- self.tri_cliques[n].nbrs[p][0]
        self.tri_cliques[n].nbrs[p][1] =
              self.tri_cliques[n].observed_pot.marginalize_pot(
              dom, False)[0]

      STEP 2.4: Send the message from clique node n to clique
                node p by multiplying the message into node p's
                potential.

        self.tri_cliques[p].observed_pot.arithmetic(
                self.tri_cliques[n].nbrs[p][1], '*')
      end
    end

  STEP 3.1: Belief propagation: The backward pass. Start by
            iterating through the backward message passing
            order stored in the list self.pre_order.

    N <- The length of the list self.post_order
    for i <- 0 to N-1
      n = self.pre_order[i]

    STEP 3.2: Iterate through the list of nodes which node n
              has to send messages to.

      po_children <- self.pre_order_children[n]

      M <- The length of the list po_children
      for j <- 0 to M-1
        c <- po_children[j]

      STEP 3.3: Calculate the message from clique node n to
                clique node c by marginalizing the potential of
                clique node n onto the variable nodes
                separating clique nodes n and c.

        dom <- self.tri_cliques[n].nbrs[c][0]

        self.tri_cliques[n].nbrs[c][1] =
              self.tri_cliques[n].observed_pot.marginalize_pot(
              dom, False)[0]

      STEP 3.4: Send the message from clique node n to clique
```

```
                node c by multiplying the message into node c's
                potential.

          self.tri_cliques[c].observed_pot.arithmetic(
            self.tri_cliques[n].nbrs[c][1], '*').
        end
      end

    STEP 4: Normalize all the clique potentials.

      N <- The length of the list self.tri_cliques.
      for i <- 0 to N-1
        Normalize the potential of self.tri_clique[i].
      end
```

JTREE.max_sum: This method executes the max-sum algorithm on the junction tree initialized by either the JTREE.init_for_mrf or JTREE.init_for_bnet methods. To execute the max-sum algorithm on the junction tree we need only the following parameter:

1 *list: evidence.* A list of observed evidence with the length equal to the number of variable nodes in the MODEL that this junction tree was initialized from. The value of evidence[i] is either None, to represent that the node x_i is unobserved, or an integer value k to represent the observation $x_i = k$.

The algorithm for the JTREE max_sum method is shown below.

```
JTREE.max_sum(evidence)
  STEP 1: Enter the evidence into the clique potentials.

    onodes <- A list containing the indices of the entries in
              the list evidence which are not None. Therefore,
              if evidence = [None, 1, None, 2], onodes = [1, 3].

    N <- The length of the list self.tri_cliques.
    for i <- 0 to N-1
      self.tri_cliques[i].enter_evidence(evidence, TRUE)

      self.tri_cliques[i].init_sep_pots(self.model.node_sizes,
                                        onodes, TRUE)
    end
```

```
STEP 2.1: Belief propagation: The forward pass and storing
          the backtracking information. Start by iterating
          through the forward message passing order stored
          in the list self.post_order.

  max_track <- Empty key indexed list used to store
               backtracking data.

  order <- Empty list used to store the order in which to
           backtrack through the variable nodes.

  N <- The length of the list self.post_order.
  for i <- 0 to N-1
    n <- self.post_order[i]

    po_parents <- self.post_order_parents[n]

  STEP 2.2: Iterate through the list of nodes which node n.
            has to send messages to.

    M <- The length of the list po_parents.
    for j <- 0 to M-1
      p <- po_parents[j]

    STEP 2.3c: Calculate the message from clique node n to
               clique node p by maximizing the potential of
               clique node n onto the variable nodes
               separating clique nodes n and p.

      dom <- self.tri_cliques[n].nbrs[p][0]

      [self.tri_cliques[n].nbrs[p][1], maximizers] =
            self.tri_cliques[n].observed_pot.marginalize_pot(
            dom, TRUE)

    STEP 2.4: Save the backtracking data calculated during
              maximization of the clique node n's potential.

      keys <- All the keys which index the list maximizers.
      K <- The length of the list keys.
      for k <- 0 to K -1
        key <- keys[k]
```

```
          if key not in order.
            order <- Insert key at first position in order.
          end
          max_track[key] <- maximizers[key]
        end

      STEP 2.5: Send the message from clique node $n$ to clique
                node $p$ by multiplying the message into node $p$'s
                potential.

        self.tri_cliques[p].observed_pot.arithmetic(
          self.tri_cliques[n].nbrs[p][1], '+').
      end
    end

  STEP 3:  Calculate and save the backtracking data for the
           variable nodes in the root clique of the junction
           tree which are not separation nodes.

    non_sep_nodes <- All the nodes in the root clique which are
                     not in one of the separation potential
                     attached to the root clique.

    p <- The index in self.tri_cliques of the root clique.

    N <- The length of the list non_sep_nodes.
    for i <- 0 to N-1
      [temp_pot, maximizers] <-
          self.tri_cliques[p].unobserved_pot.marginalize(
          non_sep_nodes[i], TRUE]

      keys <- All the keys which index the list maximizers.
      K <- The length of the list keys.
      for k <- 0 to K -1
        key <- keys[k]

        if key not in order.
          order <- Insert key at first position in order.
        end

        max_track[key] <- maximizers[key]
      end
```

```
     STEP 4: Execute backtracking to determine the MAP
             configuration of the variable nodes.

       map <- A copy of the parameter evidence, since the most
              likely value of an observed node is its observed
              value. Using backtracking we fill in the None
              entries in the list.

       N <- The length of the list order.
       for i <- 0 to N-1
         node <- order[i]
         if map[node] is None
           dependants <- max_track[node][0]

           argmax <- max_track[node][1]

           M <- The length of the list dependants.
           for j <- 0 to M-1
             dep_node <- dependants[j]
             value <- argmax[j]

             if map[dep_node] is None
               map[dep_node] <- argmax[j]
             end
           end
         end
       end

   STEP 5: Return the output.

     return map
```

JTREE.marginal_nodes: This method returns the marginal probability distribution over a set of variable nodes in the model.

To obtain the marginal probability distribution over a set of nodes we need the following input parameters:

1 *list: query.* A list of integers indexing a set of nodes in the model. These nodes must be the subset of one of the triangulated cliques in the model.

This method outputs the following values:

1 *POTENTIAL: marginal.* A potential object expressing the marginal probability distribution over the nodes indexed by the parameter query.

The algorithm for the JTREE marginal_nodes method is shown below.

```
JTREE.marginal_nodes(query)
  STEP 1: Determine which triangulated clique the nodes indexed
          in the parameter query are a subset of.

    N <- The length of the list self.tri_cliques.
    for i <- 0 to N-1
      if query is a subset of self.tri_cliques[i].domain
        c <- i
        Exit the loop.
      end
    end

  STEP 2: Marginalize the potential of the clique indexed by c
          onto the domain indexed in the parameter query.

    marginal <-
        self.tri_cliques[c].observed_pot.marginalize_pot(query)

  STEP 3: Return the marginal probability distribution.

    return marginal
```

6.9 The LOOPY class

The LOOPY class, which is derived from the INF_ENGINE class, is used to implement approximate probabilistic inference on MRF and BNET objects via the loopy belief algorithm.

6.9.1 Attributes

Since the loopy belief propagation algorithm works directly with the graph structure and the cliques already present in the model it is being applied to, it only requires the following attribute:

1 A model, either Bayesian network or Markov random field, for which to perform the inference.

As with the junction tree algorithm, to perform inference using the loopy belief algorithm we need all the information associated with the model it is being applied to, so we add the attribute *MODEL: model* to the LOOPY class.

6.9.2 Methods

The following paragraphs detail each of the methods within the LOOPY class.

LOOPY.init: The initialization method for the class.
To initialize a LOOPY object we need the following parameter:

1 *MODEL: model.* The model we wish to perform approximate inference on.

This method simply selects the correct method to initialize loopy belief propagation based on which type of model, Bayesian network or Markov random field, was passed to it. The algorithm for the LOOPY init method is shown below.

```
LOOPY.init(model)
  STEP 1: Determine which model the loopy belief propagation
          inference engine is coupled to and initialize it
          for that model.

    if model is a BNET object
      self.init_for_bnet(model)
    else
      self.init_for_mrf(model)
    end
```

LOOPY.init_for_mrf: Initializes the loopy belief propagation algorithm for a Markov random field.
The initialization phase of the loopy belief propagation algorithm involves finding the separation between neighbouring cliques and initializing the storage for the separation potentials between those cliques. To initialize a LOOPY object for a Markov random field we need the following parameter:

1 *MRF: mrf.* The Markov random field on which to perform approximate inference on.

The algorithm for the LOOPY init_for_mrf method is shown below.

```
LOOPY.init_for_mrf(mrf)
  STEP 1: Save the required input parameters as attributes.

    self.model <- mrf
    self.cliques <- self.model.cliques

  STEP 2: Determine which cliques are neighbours, and find the
          indices of the variable nodes which separate the
          neighbouring cliques.

    N <- The length of the list self.cliques.
  for i <- 0 to N-1
    for j <- 0 to N-1
      sep <- A list indexing the variable nodes in the
             intersection between the cliques i and
             j. This is the intersection between the lists
             self.cliques[i].domain and
             self.cliques[j].domain.

      if sep is not an empty list
        self.cliques[i].nbrs[j] <- [sep, None]
      end
    end
  end
```

LOOPY.init_for_bnet: Initializes the loopy belief propagation algorithm for a Bayesian network.

The initialization phase of the loopy belief propagation algorithm for a Bayesian network involves first converting the conditional probability distributions into potentials over cliques, and then finding the separation between neighbouring cliques and initializing the storage for the separation potentials between those cliques. To initialize a LOOPY object for a Bayesian network we need the following parameter:

1 *BNET: bnet.* The Bayesian network on which to perform approximate inference on.

The algorithm for the LOOPY init_for_bnet method is shown below.

```
LOOPY.init_for_bnet(bnet)
  STEP 1: Save the required input parameters as attributes.

    self.model <- bnet

  STEP 2: Convert all the CPD objects within the list
          self.model.cpds into clique objects with potentials.

    self.cliques <- An empty list.

   N <- The length of the list self.model.cpds.
    for i <- 0 to N-1
      domain <- A sorted list of all the
                variable nodes connected to variable node
                indexed by i, as well as the value i.
```

```
                  Therefore, if the $i$=2, and the variable
                  nodes connected to 2 are 1, 5 and 6 then
                  domain<-[1, 2, 5, 6]

        clq_pot <- self.model.cpds[i].convert_to_pot(domain)

        clq <- CLIQUE.init(i, clq_pot.domain, clq_pot.size,
                          clq_pot.T)

        self.cliques <- Append the CLIQUE clq to this list.

    STEP 3: Determine which cliques are neighbours, and find the
            indices of the variable nodes which separate the
            neighbouring cliques.

      N <- The length of the list self.cliques.
      for i <- 0 to N-1
        for j <- 0 to N-1
          sep <- A list indexing the variable nodes in the
                 intersection between the cliques $i$ and
                 $j$. This is the intersection between the lists
                 self.cliques[i].domain and
                 self.cliques[j].domain.

          if sep is not an empty list
            self.cliques[i].nbrs[j] <- [sep, None]
          end
        end
      end
```

LOOPY.sum_product: This method executes the loopy belief propagation sum-product algorithm on the model specified in the JTREE.init method. To execute the sum-product algorithm we need the following parameters:

1 *list: evidence.* A list of observed evidence with the length equal to the number of variable nodes in the MODEL that this LOOPY object was initialized from. The value of evidence[i] is either None, to represent that the node x_i is unobserved, or an integer value k to represent the observation $x_i = k$.

2 *int: max_iter.* The maximum number of iterations the loopy belief propagation algorithm can achieve.

The algorithm for the LOOPY sum_product method is shown below.

```
LOOPY.sum_product(evidence, max_iter)
  STEP 1: Entering evidence into the clique potentials.

    onodes <- A list containing the indices of the entries in
              the list evidence which are not None. Therefore,
              if evidence=[None, 1, None, 2], onodes=[1, 3].

    N <- The length of the list self.cliques.
    for i <- 0 to N-1
      self.cliques[i].enter_evidence(evidence)

      self.cliques[i].init_sep_pots(self.model.node_sizes,
                                     onodes, FALSE)
    end

  STEP 2: Initialize the lists which will store the belief at
          each clique for the current iteration as well as the
          previous iteration.

    bel <- A list with length equal to that of self.cliques.
    old_bel <- A list with length equal to that of self.cliques.

  STEP 3: Iterate the message passing until the beliefs at each
          clique have converged or the maximum number
          iterations has been reached.

    converged <- FALSE
    count <- 0
    while converged is FALSE and count is less that max_iter
      count <- count + 1

    STEP 3.1: Execute the collection phase, start by iterating
              through all the CLIQUE objects in the list
              self.cliques.

      old_cliques <- A list with length equal to that of
                     self.cliques.
      N <- The length of the list self.cliques.
      for i <- 0 to N-1

      STEP 3.2: Initialize the belief of clique node i to
                be equal to its potential.
      bel[i] <- A copy of self.clique[i].observed_pot.

    STEP 3.3: Iterate through the list of cliques i's
              neighbours and multiply clique i's belief
              with the message sent from each neighbour.

      keys <- A list of the keys which index the
              self.cliques[i].nbrs.

      M <- The length of the list keys.
      for j <- 0 to M-1
        key <- keys[j]
        bel[i].arithmetic(self.cliques.nbrs[key][1], '*')
      end

      bel[i] <- Normalize this belief.

      old_clqs[i] <- A copy of self.cliques[i].
    end

  STEP 4.1: Execute the distribution phase, starting by
            iterating through all the CLIQUEs in the
            self.cliques list.

    N <- The length of the list self.cliques.
    for i <- 0 to N-1
      keys <- A list of the keys which index the
              self.cliques[i].nbrs.

    STEP 4.2: Iterate through the list of cliques i's
              neighbours to update the messages from
              clique i to each of its neighbours.

      M <- The length of the list keys.
      for j <- 0 to M-1
        key <- keys[j]

      STEP 4.3: Before updating clique i's outgoing message
                to its neighbour indexed by key, divide the
                old message received from its neighbour from
                clique i's belief.

        sep_set <- self.cliques[i].nbrs[key][0]
```

```
          msg_to_nbr <- A copy of bel[i].

          msg_to_nbr.arithmetic(old_clqs[key].nbrs[i][1], '/')

        STEP 4.3: Calculate the message from clique i to its
                  neighbour indexed by key by marginalizing the
                  belief of clique i onto the domain which
                  separates it from its neighbour indexed by
                  key.

          msg_to_nbr <- msg_to_nbr.marginalize_pot(sep_set,
                                                   FALSE)

        STEP 4.4: Update the message from clique i to its
                  neighbour indexed by key.

          self.cliques[i].nbrs[key][1] <- A copy ofmsg_to_nbr.
        end
      end

    STEP 5: If more than 3 iterations have passed, compare the
            beliefs from the previous iteration with the ones
            from current iteration and check for convergence.

      if count is greater than 3
        Compare the lists bel and old_bel element wise for
        differences in the potential values. If all elements
        are similar with in an acceptable tolerance, then set
        converged <- TRUE, else leave it as FALSE.
      end

      old_bel <- A copy of bel.
    end

  STEP 6: Save the clique beliefs as an attribute which will
          be used to determine the marginal distributions over
          query nodes in the LOOPY.marginal_nodes method.

    self.marginal_domains <- A copy of bel.
```

LOOPY.max_sum: This method executes the loopy belief propagation max-sum algorithm on the model specified in the LOOPY.init method. To execute the max-sum algorithm we need the following parameters:

1 *list: evidence.* A list of observed evidence with the length equal to the number of variable nodes in the MODEL that this LOOPY object was initialized from. The value of evidence[i] is either None, to represent that the node x_i is unobserved, or an integer value k to represent the observation $x_i = k$.

2 *int: max_iter.* The maximum number of iterations the loopy belief propagation algorithm can achieve.

The algorithm for the LOOPY max_sum method is shown below.

```
LOOPY.max_sum(evidence, max_iter)
  STEP 1: Enter the evidence into the clique potentials.

    onodes <- A list containing the indices of the entries in
              the list evidence which are not None.
              Therefore, if evidence=[None, 1, None, 2],
              onodes=[1, 3].

    N <- The length of the list self.cliques.
    for i <- 0 to N-1
      self.cliques[i].enter_evidence(evidence)

      self.cliques[i].init_sep_pots(self.model.node_sizes,
                                    onodes, TRUE)
    end

  STEP 2: Initialize the lists which will store the belief at
          each clique for the current iteration as well as the
          previous iteration. Also initialize the key indexed
          list which will store the back tracking information.

    bel <- A list with length equal to that of self.cliques.
    old_bel <- A list with length equal to that of self.cliques.
    max_track <- A empty key-indexed list.

  STEP 3: Iterate the message passing until the beliefs at each
          clique have converged or the maximum number
          iterations has been reached.

    converged <- FALSE
```

```
   count <- 0
   while converged is FALSE and count is less that max_iter
     count <- count + 1

   STEP 3.1: Execute the collection phase, start by iterating
             through all the CLIQUE objects in the list
             self.cliques.

     old_cliques <- A list with length equal to that of
                    self.cliques.
     N <- The length of the list self.cliques.
     for i <- 0 to N-1

     STEP 3.2: Initialize the belief of clique node i to
               be equal to its potential.

       bel[i] <- A copy of self.clique[i].observed_pot.

     STEP 3.3: Iterate through the list of cliques i's
               neighbours and add the message sent from
               each neighbour to clique i's belief.

       keys <- A list of the keys which index the
               self.cliques[i].nbrs.

       M <- The length of the list keys.
       for j <- 0 to M-1
         key <- keys[j]

         bel[i].arithmetic(self.cliques.nbrs[key][1], '+')
       end
       old_clqs[i] <- A copy of self.cliques[i].
     end

   STEP 4.1: Execute the distribution phase, starting by
             iterating through all the CLIQUEs in the
             self.cliques list.

     N <- The length of the list self.cliques.
     for i <- 0 to N-1
       keys <- A list of the keys which index the
               self.cliques[i].nbrs.
```

```
    STEP 4.2: Iterate through the list of cliques i's
              neighbours to update the messages from
              clique i to each of its neighbours.

     M <- The length of the list keys.
     for j <- 0 to M-1

     STEP 4.3: Before updating clique i's outgoing message
               to its neighbour indexed by key, divide the
               old message received from its neighbour from
               clique i's belief.

       sep_set <- self.cliques[i].nbrs[key][0]

       msg_to_nbr <- A copy of bel[i].

       msg_to_nbr.arithmetic(old_clqs[key].nbrs[i][1], '-')

     STEP 4.3: Calculate the message, and backtracking data,
               from clique i to its neighbour indexed by key,
               by maximizing the belief of clique i onto the
               domain which separates it from its neighbour.

       [msg_to_nbr, maximizers] <-
                    msg_to_nbr.marginalize_pot(sep_set, TRUE)

     STEP 4.4: Update the message from clique i to its
               neighbour indexed by key.

       self.cliques[i].nbrs[key][1] <- A copy of msg_to_nbr.

     STEP 4.5: Save the backtracking data calculated while
               maximizing over belief of clique node i in
               the key indexed max_track list.

       max_keys <- A list of the keys which index maximizers.
       K <- The length of the list max_keys.
       for k <- 0 to K -1
         max_key <- max_keys[k]
         max_track[max_key] <- maximizers[max_key]
       end
     end
```

```
   STEP 5: If more than 3 iterations have passed, compare the
           beliefs from the previous iteration with the ones
           from this iteration and check for convergence.

     if count is greater than 3
       Compare the lists bel and old_bel element wise for
       differences in the potential values. If all elements
       are similar with in an acceptable tolerance, then set
       converged <- TRUE, else leave it as FALSE.
     end

     old_bel <- A copy of bel.
   end

 STEP 6: Once the beliefs have converged, use the information
         saved in max_track to backtrack and set the MAP values
         of all the nodes and return the output.

   mlc <- self.back_track(evidence, max_track)

   return mlc
```

LOOPY.back_track: This method uses the back-tracking data created during the loopy belief propagation max-sum algorithm to determine the maximum state of each variable in the model. To execute the back-track method we need the following parameters:

1 *list: evidence.* A list of observed evidence with the length equal to the number of variable nodes in the MODEL that this LOOPY object was initialized from. The value of evidence[i] is either None, to represent that the node x_i is unobserved, or an integer value k to represent the observation $x_i = k$.

2 *list: max_track.* The back-tracking data obtained during the message passing of the loopy-belief max-sum algorithm.

The algorithm for the LOOPY back_track method is shown below.

```
LOOPY.back_track(evidence, max_track)
  STEP 1: Create the list to store the MAP values and set the
          MAP, values of the observed nodes to their observed
          values.
```

```
    mlc <- A copy of the parameter evidence.

  STEP 2: Determine which nodes have been observed.

    set_nodes <- A list of the indices of the entries in the
                 list evidence which are not None.

  STEP 3: Determine the best node to begin backtracking from.

    node <- The index of the node which has the most nodes
            connected directly connected to it.

    if mlc[node] is None
      marg <- self.marginal_nodes([node], True)

      mlc[node] <- The maximum value of marg.

      set_nodes <- Append the value node to this list.
    end

  STEP 4: Iterate through all the keys, which index the nodes
          in the model, in the max_track list.

    max_keys <- A list of the keys which index max_track.
    N <- The length of the list max_keys.
    for i <- 0 to N-1
      node <- max_keys[i]

    STEP 4.1: If this MAP value of this node has not yet been
              set, then determine the value and set it in
              the mlc list.

      if mlc[Node] is None and the list max_track[node][0] is
        a subset of set_nodes.

      STEP 4.2: Determine which nodes maximize this node, and
                the values that those nodes assume to maximize
                this node.

        dep_nodes <- max_track[node][0]
        argmax <- max_track[node][1]

        ndx <- An empty list.
        M <- The length of the list dep_nodes.
        for j <- 0 to M-1
          if mlc[dep_nodes[i]] not None
            ndx <- Append mlc[dep_nodes[i]] to this list.
          end
        end

        possible_vals <- All the entries in the list argmax
                         indexed by the list ndx.

        mlc[node] <- The maximum value in the list possible_vals.

        set_nodes.append(node)
      end
    end
```

LOOPY.marginal_nodes: This method returns the marginal probability distribution over a set of variable nodes in the model.

To obtain the marginal probability distribution over a set of nodes we need the following input parameters:

1 *list: query.* A list of integers indexing a set of nodes in the model. These nodes must be the subset of one of the cliques in the model.

This method outputs the following values:

1 *POTENTIAL: marginal.* A potential object expressing the marginal probability distribution over the nodes indexed by the parameter query.

The algorithm for the LOOPY marginal_nodes method is shown below.

```
LOOPY.marginal_nodes(query)
  STEP 1: Determine which clique the nodes indexed in the
          parameter query are a subset of.

    N <- The length of the list self.marginal_domains.
    for i <- 0 to N-1
      if query is a subset of self.marginal_domains[i].domain
        m <- i
        Exit the loop.
      end
    end

  STEP 2: Marginalize the potential of the clique indexed by m
          onto the domain indexed in the parameter query.

    marginal <-
         self.marginal_domains[m].marginalize_pot(query)

  STEP 3: Return the marginal probability distribution.

    return marginal
```

6.10 Supporting functions

In this section we outline the functions required by the classes discussed so far to implement graphical models, that are not explicitly part of any one class. These supporting functions allow us to triangulate and moralize graphs.

moralize: This function moralizes a directed graph into a undirected graph, and requires the following input parameter:

1 *array: adj_mat.* An adjacency matrix defining the structure of the directed graph. If *adj_mat[i, j]=1*, then there exits a directed edge from node x_i to node x_j.

This method outputs the following values:

1 *array: moral_adj_mat.* An adjacency matrix defining the structure of the moralized graph. If *moral_adj_mat[i, j]=1*, then there exits an undirected edge from node x_i to node x_j.

The algorithm for the moralize function is outlined below.

```
moralize(adj_mat)
  STEP 1: Determine the number of nodes in the graph
          represented by the adjacency matrix, and copy all
          the current edges in the graph into the moralized
          graph.

    N <- The number or rows in adj_mat.
    moral_adj_mat <- A copy of adj_mat.

  STEP 2: Visit each node in the graph.
```

```
    for i <- 0 to N-1

      STEP 2.1: Determine which nodes in the directed graph
                are parents of node i.

        parents <- The indices of the entries in column i in
                   adj_mat which are 1. These are the indices
                   of the nodes which have a directed edge
                   pointing to node i.

      STEP 2.2: Visit every parent node and connect it to every
                other parent node in the moralized graph.

        M <- The length of the list parents.
        for j <- 0 to M-1
          moral_adj_mat[parents[j], parents] = 1
          moral_adj_mat[parents[j], parents[j]] = 0
        end
    end

  STEP 3: Return the moralized graph.

    return moral_graph
```

triangulate: This function triangulates an undirected graph and determines the maximal cliques within the triangulated graph. It requires the following input parameters:

1 *array: adj_mat.* An adjacency matrix defining the structure of the undirected graph. If *adj_mat[i, j]=1*, then there exits an undirected edge from node x_i to node x_j.

2 *list: order.* A list indexing the nodes in the graph in the order they are to be eliminated.

Determining an optimal elimination order for item 2 in the input parameters is not within the scope of this thesis, however a method to do so is included in the GrMPy package. The triangulate method outputs the following values:

1 *array: tri_adj_mat.* An adjacency matrix defining the structure of the triangulated graph. If *tri_adj_ma[i, j]=1*, then there exits an undirected edge from node x_i to node x_j.

2 *list: max_cliques.* A nested list indexing the domains of the maximal cliques in the triangulated graph.

The algorithm for the triangulate function is outlined below.

```
triangulate(adj_mat, order)
  STEP 1: Initialize an array of zeros used to indicate which
          nodes have been eliminated, and initialize an empty
          list to store the maximal cliques.

    N <- The number of nodes in the non-triangulated graph,
         which is the number of rows in adj_mat.
    eliminated <- A zero-filled list of length N.
    max_cliques <- An empty list.

  STEP 2: Visit each node in the order specified in the input
          parameter order.

    for i <- 0 to N-1
      j <- order[i]

    STEP 2.1: Determine which nodes are both connected to the
              node j and not yet eliminated.

      nbrs <- The indices of the entries in column j in  adj_mat
              that are equal to 1. These are the indices of the
              nodes connected to the node j by an edge.

      unelim_nodes <- The indices of the entries in the list
                      eliminated which are equal to 0. These are
                      the indices of the nodes that have not
                      yet been eliminated.

      fam_nodes <- The intersection between the lists nbrs and
                   elim_nodes.

    STEP 2.2: Connect all the uneliminated neighbours of node
              j, indexed in the list fam_nodes, together
              in input graph.

      M <- The length of the list fam_nodes.
      for m <- 0 to M-1
        adj_mat[m, fam_nodes] = 1
        adj_mat[m, m] = 0
      end

    STEP 2.3: Mark node j as being eliminated.
```

```
      eliminated[j] = 1

    STEP 2.4: Determine the elimination clique created by
              eliminating node j.

      elim_clique <- A copy of the list fam_nodes with the
                     value j appended to it.

    STEP 2.5: Determine whether the elimination clique is a
              maximal clique by checking if it is a subset of
              any of the maximal cliques created so far.

      C <- The length of the list max_cliques.
      exclude <- FALSE
      for c <- 0 to C-1
        if elim_clique is a subset of max_cliques[c]:
          exclude <- TRUE
          End the loop
        end
      end

      if exclude is False:
        max_cliques <- Append elim_clique to the list max_cliques.
      end
    end

  STEP 3: Return the triangulated graph and the nested list of
          maximal cliques.

    return [adj_mat, max_cliques]
```

Chapter 7 - Applications

In this section we outline some of the non-trivial problems the GrMPy package has been applied to so far. The two applications covered in this section are image denoising and audio-visual speech recognition. The image denoising application was created to test the early functionality of GrMPy. The audio-visual speech recognition application is the work of a contributer to GrMPy [Reikeras, 2009]. Note that these applications make use of a fraction of GrMPy's features. There are many tutorial-like examples, explaining the use of most of the packages features, available at http://dip.sun.ac.za/vision/trac-git/agouws-GrMPy.bzr.

7.1 Image denoising

As part of the development of GrMPy, the package was tested on the non-trivial problem of removing noise from an image. The problem and the image it is taken from [Bishop, 2006], though a different inference algorithm is used. If we are given a binary image where a certain percentage of the pixels are flipped, how do we remove the noise using the graphical model framework described in this thesis? We model the denoised image as an unobserved Markov random field, where every pixel i represents a binary node x_i that can take on the value of 0 or 1. Each node x_i is four-connected by edges to its neighbouring nodes creating a lattice structure. To represent the noisy image we connect a binary node y_i to each node x_i representing the denoised image, forming another layer in the model. Each node y_i is set to the observed value of the noisy pixel i. This is known as an *Ising Model*, and is shown in Figure 7.1.

It is clear that the maximal cliques in the model shown in Figure 7.1 are simply the edges in the graph, and that there are two types of maximal cliques. The first type of maximal clique connects denoised image nodes $\{x_i, x_j\}$, and represents the relationship between neighbouring pixels in an image. The potential function for these cliques is given by,

$$\psi(x_i, x_j) = e^{\frac{-(x_i - x_j)^2}{\sigma_i}}, \tag{7.1}$$

where σ_i allows us to adjust how similar we expect neighbouring pixels to be. This function takes advantage of the fact the neighbouring pixels in natural images are usually similar. The second type of maximal clique connects denoised image nodes to noisy image nodes $\{x_i, y_i\}$, and represents the noise in the image. The potential function for these cliques is given by,

$$\psi(x_i, y_i) = e^{\frac{-(x_i - y_i)^2}{\sigma_n}}, \tag{7.2}$$

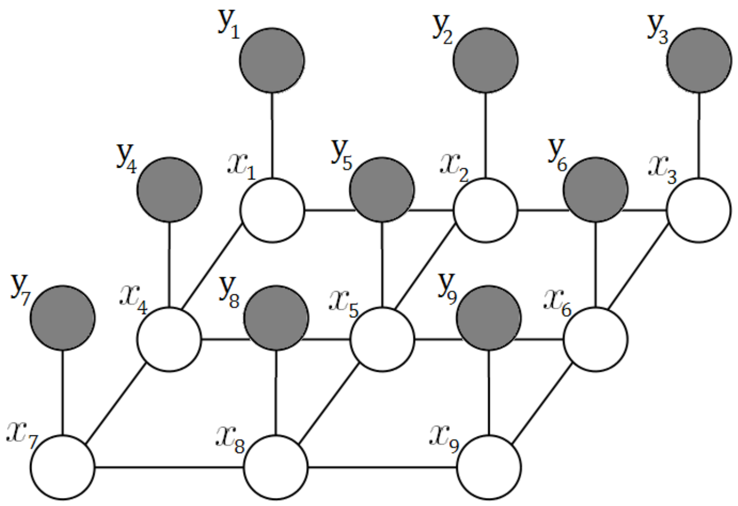

Figure 7.1: The Markov random field created from a 3x3 image. The shaded nodes indicate the observed nodes.

where σ_n allows us to adjust how bad we expect the noise to be. This function takes advantage of the fact that most pixels in the noisy image have not been corrupted, meaning that the value of a noisy pixel and its corresponding denoised pixel are likely to be similar.

Since this model exhibits a lattice structure, it is computationally expensive to triangulate it, therefore the best choice for inference is the loopy belief propagation algorithm. Thus we simply execute the max-sum algorithm using loopy belief propagation to find the MAP configuration of the hidden nodes which represent the denoised image. We then use the MAP values of the hidden nodes to reconstruct the denoised image. Figure 7.2 is taken from [Bishop, 2006], and in Figure 7.3 we show the results of corrupting this image with two different degrees of noise, and the result removing that noise via the method explained in this section.

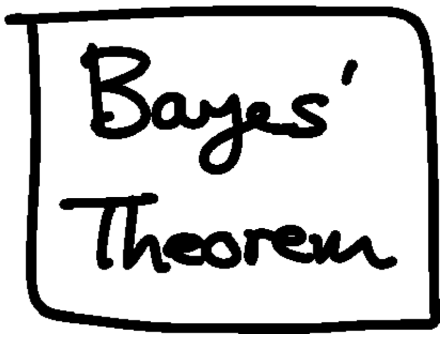

Figure 7.2: An uncorrupted image.

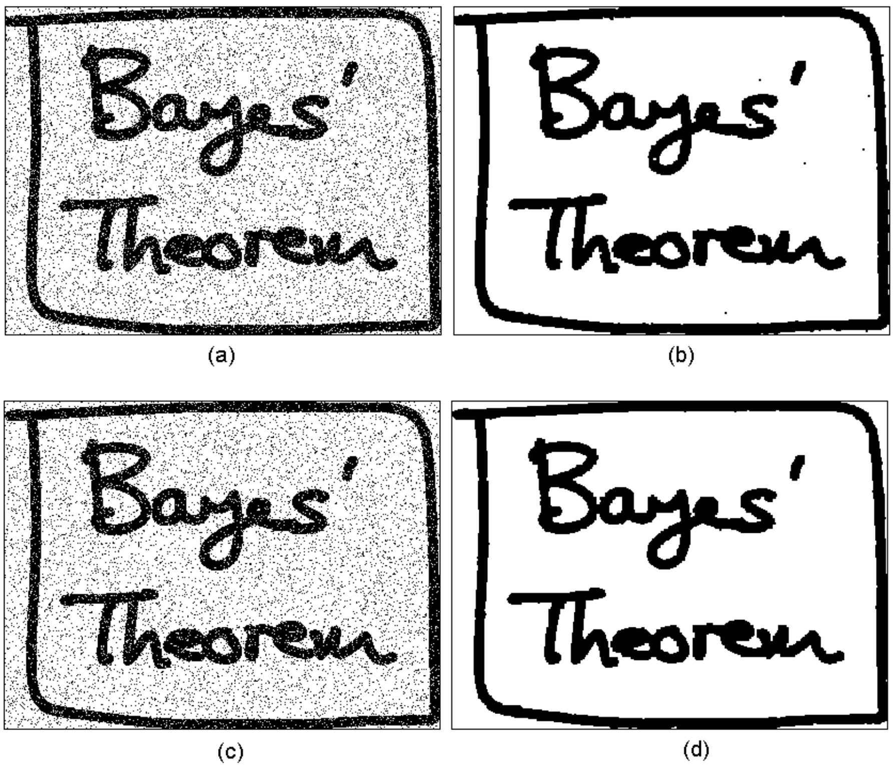

Figure 7.3: (a) Figure 7.2 with 10% of the pixels corrupted. (b) The image from (a) after being denoised. (c) Figure 7.2 with 6% of the pixels corrupted. (d) The image from (c) after being denoised

7.2 Audio-visual speech recognition

The GrMPy package is being used to implement and experiment with various probabilistic models for performing Audio-Visual Speech Recognition [Reikeras, 2009].

In Audio-Visual Speech Recognition (AVSR) informative features are extracted from synchronous audio and video recordings of a person speaking [Neti et al., 2000]. These features are then used to train probabilistic models representing each separate word, and the models are combined to form the a basis for a AVSR system. The type of probabilistic models used for modeling dynamic systems such as speech, fall under the class of Dynamic Bayesian Networks (DBN). As seen in Section 5.2.2, DBNs are an important class of probabilistic graphical models used to model discrete time-dynamic processes.

A DBN is defined by a pair of Bayesian networks $(B_1, B_{t,t-1})$, where B_1 represents the initial state of the system, and B_{t,t_1} the transition for one time-slice to the next. In general Bayesian networks can be separated into sets of observed and latent variables. In AVSR, the observed variables correspond to features extracted from audio and video, and the latent variables represent the discrete states of system which can be interpreted as underlying processes responsible for explaining the observations. Speech recognition is performed by finding the maximum-a-posteriori (MAP) configuration of the latent variables for each word's probabilistic model, given the observed feature data. The model with the greatest MAP will then represent the most likely word spoken.

A central problem to any AVSR system is the audio-visual data fusion problem [Neti et al., 2000]. It is well known that the acoustic and visual features of audio-visual speech is generally not time-synchronous. This a consequence of speakers forming the lips and facial expression in anticipation of the sound to be produced. The asynchrony property is something that can be fairly easily modeled using a DBN by allowing state-asynchrony between the audio and visual states of the corresponding observation streams [Nefian et al., 2002].

GrMPy allows the rapid implementation of various probabilistic models for AVSR, which can easily be compared for performance and accuracy in a consistent framework ideal for research.

Chapter 8 - Future work and conclusion

GrMPy is an ongoing project, and there are many aspects of its performance and functionality that can be improved. In this section we take a broad look at some of the areas within GrMPy that can be improved.

The GrMPy package's overall performance can be improved greatly. There are two main performance bottlenecks that need to be addressed, the algorithms used and Python itself. Within the implemented algorithms there are performance issues that were overlooked during initial development, issues which were not apparent from the mathematics before going to code. For instance, consider the EM algorithm as it is explained in Section 6.2. At the end of every iteration the algorithm updates the parameters of the model, to be used during the inference in the next iteration. However, it updates the parameters of the model, not the inference engine. Therefore, we need to reinitialize the inference engine based on the updated model parameters at the beginning of every iteration, which repeats a large number of calculations, such as triangulating the graph. The solution would be to initialize the inference engine once, and to perform the EM algorithm on the cliques within the inference engine, and update the parameters of the model from the engines parameters once the EM algorithm has terminated. There are likely several performance bugs such as this one lurking within the implementation, and correcting these errors will result in a moderate improvement in performance.

The main performance bottleneck within the GrMPy package is the use of Python, the language it has been implemented in. Python allows for easy implementation of mathematical algorithms, and is also a great platform for open-source software. Open-source packages such as Numpy and Scipy offer a large number of quality mathematical tools allowing quick development and of mathematical code with excellent readability. However, since Python is an interpreted language its execution is around 20 times slower than a compiler language such as C++, especially when the code executes many loops, and it should be clear from Chapter 6 that graphical model algorithms involve many loops. Rewriting the code into C++ is not practical, as the code will no longer be as easy to use or understand. An alternative approach would be to extend Python with C, allowing the slower and frequently used functions within GrMPy to be implemented in C and called from the Python interpreter. The arithmetic and marginalize_pot methods in the POTENTIAL class explained in Section 6.6 execute many loops and are the most frequently called methods within the package. Improving the execution speed of these two methods alone should result in a large performance improvement. Another approach is to make use of the cython package, which allows a more natural mixture of C and Python.

Though GrMPy has a fairly large amount of functionality compared to most related software, see Section 1.1, there is no shortage of new functionality that can included. Listed below are some of the features we would like to include in GrMPy in the future:

1 Support for continuous Gaussian nodes in Markov random fields.

2 A larger set of distributions for both Markov random fields and Bayesian networks.

3 Support for other approximate inference algorithms, such as generalized belief propagation.

3 The option to save/load models and their parameters.

4 A graphical user interface to create and manipulate graphical models.

5 Optimized inference and parameter estimation methods for common graphical models, such as Hidden Markov Models and Kalman filters.

In this thesis we presented an open-source Python implemented toolbox of classes and functions aimed at solving problems pertaining to graphical models. We outlined the implemented algorithms, as well as the mathematical theory behind the algorithms, and substantiated the design of the toolbox based on the theory. To illustrate the packages correctness and usefulness we applied it to the non-trivial problem of image denoising, and obtained satisfactory results. More examples of the packages functionality can be found as tutorials and unit-tests at http://dip.sun.ac.za/vision/trac-git/agouws-GrMPy.bzr. In the time it took to write this thesis, GrMPy has already grown passed its original expectations and is being utilised as a research tool, as demonstrated in Section 7.2. GrMPy's overall design makes it easy for others to extend its functionality, as demonstrated in Section 5.2. In the future we hope for GrMPy to be a valuable, accessible and comprehensive tool for any research in the field of graphical models, as well as an aid in offering insight to the inner working of graphical model algorithms.

Bibliography

C. M. Bishop. *Pattern recognition and Machine Learning*. Springer, 2006.

A. P. Dempster, N. M. Laird, and N. M. Rubin. Maximum likelihood from incomplete data via the EM algorithm. *JOURNAL OF THE ROYAL STATISTICAL SOCIETY, SERIES B*, 39(1):1–38, 1977.

V. Didelez and I. Pigeot. Maximum likelihood estimation in graphical models with missing values. *Biometrika*, 85:960–6, 1998.

Z. Ghahramani. Graphical models: Parameter learning. In *Handbook of Brain Theory and Neural Networks*, pages 486–490. MIT Press, 2003.

F. V. Jensen and F. Jensen. Optimal junction trees. In *UAI*, pages 360–366. Morgan Kaufmann, 1994.

E. Jones, T Oliphant, P Peterson, et al. NumPy: Open source numerical tools for Python, 2001-2008a. URL `http://www.numpy.org/`.

E. Jones, T Oliphant, P Peterson, et al. SciPy: Open source scientific tools for Python, 2001-2008b. URL `http://www.scipy.org/`.

J. M. Mooij. libDAI 0.2.3: A free/open source C++ library for Discrete Approximate Inference. http://www.libdai.org/, 2009.

K P. Murphy. The Bayes net toolbox for MATLAB. *Computing Science and Statistics*, 33:2001, 2001.

K. P. Murphy, Y. Weiss, and M. I. Jordan. Loopy belief propagation for approximate inference: An empirical study. In *Proceedings of Uncertainty in AI*, pages 467–475, 1999.

A. V. Nefian, L. Liang, X. Pi, X. Liu, and K.P. Murphy. Dynamic Bayesian networks for audio-visual speech recognition. *EURASIP J. Appl. Signal Process.*, 2002(1):1274–1288, 2002. ISSN 1110-8657.

C. Neti, G. Potamianos, J. Luettin, I. Matthews, H. Glotin, D. Vergyri, J. Sison, A. Mashari, and J. Zhou. Audio-visual speech recognition. Technical Report WS00AVSR, Johns Hopkins University, CLSP, 2000.

H. Reikeras. Private communication, 2009.

K. Tanaka and D. M. Titterington. Loopy belief propagation and. In *XIII Proceedings of the 2003 IEEE Signal Processing Society Workshop (17-19 September, 2003)*, pages 329–338. IEEE Computer Society Press, 2003.