From the Jungle to the Garden: Growing Trees for Markov Chain Monte Carlo Inference in Undirected Graphical Models

by

Jean-Noël Rivasseau,

M.Sc., Ecole Polytechnique, 2003

A THESIS SUBMITTED IN PARTIAL FULFILMENT OF
THE REQUIREMENTS FOR THE DEGREE OF

Master of Science

in

The Faculty of Graduate Studies

(Computer Science)

The University Of British Columbia

October 2005

© Jean-Noël Rivasseau, 2005

Abstract

In machine-learning, Markov Chain Monte Carlo (MCMC) strategies such as Gibbs sampling are important approximate inference techniques. They use a Markov Chain mechanism to explore and sample the state space of a target distribution. The generated samples are then used to approximate the target distribution.

MCMC is mathematically guaranteed to converge with enough samples. Yet some complex graphical models can cause it to converge very slowly to the true distribution of interest. Improving the quality and efficiency of MCMC methods is an active topic of research in the probabilistic graphical models field. One possible method is to "block" some parts of the graph together, sampling groups of variables instead of single variables.

In this thesis, we concentrate on a particular blocking scheme known as tree sampling. Tree sampling operates on groups of trees, and as such requires that the graph be partitioned in a special way prior to inference. We present new algorithms to find tree partitions on arbitrary graphs. This allows tree sampling to be used on any undirected probabilistic graphical model.

Contents

Abstract ii

Contents iii

List of Tables vi

List of Figures vii

Acknowledgements ix

1 Introduction 1

2 Probability Propagation: Exact Inference 3

2.1 Probabilistic Models 3

2.1.1 Directed Acyclic Graphs 4

2.1.2 Pairwise Markov Random Fields 5

2.1.3 Factor Graphs 6

2.1.4 Inference on graphical models 7

2.2 The Belief Propagation Algorithm 8

2.2.1 Belief Propagation for Factor Graphs 9

2.2.1.1 Message propagation expressions 10

2.2.1.2 Incorporating evidence 10

2.2.1.3 Obtaining the marginals of every random variable . . . 10

2.2.1.4 Factor Graph BP pseudo-code 11

2.2.2 Belief Propagation for Pairwise Markov Random Fields 11

2.3 Loopy Belief Propagation (LBP) 13

3 Monte Carlo Methods 15
3.1 Introducing Monte Carlo simulation 15
3.2 Markov Chain Monte Carlo methods 16
3.2.1 The Gibbs sampler 16

4 Combining Monte Carlo and Belief Propagation: MCMC Tree Sampling 19
4.1 Improving MCMC algorithms 19
4.1.1 From single sampling to group sampling 19
4.1.2 A special case of group sampling: tree sampling 21
4.2 Sampling from Trees: Forward Filtering Backward Sampling 21
4.2.1 Pairwise MRF Trees 21
4.2.2 Factor Trees 23
4.3 MCMC Tree Sampling 26
4.3.1 Tree partitions for Pairwise Markov Random Fields 28
4.3.2 Choosing a tree partition for Factor Graphs 31
4.4 MCMC Tree Sampling Implementation 32
4.4.1 Rao-Blackwellization 33
4.4.2 Pseudo-code and remarks 34

5 Finding MCMC Tree Partitions 36
5.1 Introductory remarks 36
5.1.1 Definition of the problem 36
5.1.2 Search heuristics to minimize the number of trees in an MCMC tree partition 36
5.2 The pairwise case 37
5.2.1 Essential choices 37
5.2.2 General overview 38
5.2.3 Simplifying the initial graph 39
5.2.4 Exploring and coloring the graph 40

5.2.5 Choosing an ordering on the priority queue . . . 42
5.2.6 A useful improvement . . . 45
5.2.7 Full Pseudo Code for our partitioning implementation . . . 48
5.3 The general case . . . 48
5.3.1 Changes in the factor graph case . . . 50
5.3.2 A Factor Graph partitioning example . . . 51

6 Experimental Results . . . 54
6.1 Experimental Setup . . . 54
6.2 Inference on Pairwise Graphs . . . 55
6.2.1 Fully Connected Graph . . . 55
6.2.2 Square Lattice MRF . . . 57
6.2.3 Random Graph . . . 57
6.2.4 Remarks . . . 57
6.3 Inference on Factor Graphs . . . 60
6.3.1 First QMR graph: low leak . . . 61
6.3.2 Second QMR graph: medium leak . . . 61
6.3.3 Third QMR graph: high leak . . . 61
6.3.4 Remarks . . . 61
6.4 Tree Partitioning Algorithm Results . . . 65
6.4.1 Square Lattices MRF . . . 65
6.4.2 Pairwise Random Graphs . . . 66
6.4.3 Random Factor Graphs . . . 67

7 Conclusion . . . 68

Bibliography . . . 69

List of Tables

6.1 Partitioning Algorithm applied to pairwise square-lattices MRF 65
6.2 Partitioning Algorithm applied to pairwise random graphs 66
6.3 Partitioning Algorithm applied to random factor graphs 67

List of Figures

1.1 Various tree partitions 2

2.1 A Directed Acyclic Graph 4
2.2 A Pairwise Markov Random Field 5
2.3 A square lattice Markov Random Field 6
2.4 A Factor Graph 7
2.5 The flow of belief messages on an example tree 9
2.6 Pseudo-code for Belief Propagation 12
2.7 Pseudo-code for Loopy Belief Propagation 13

3.1 Pseudo-code for the Gibbs sampler 18

4.1 A problematic distribution for MCMC approximation 20
4.2 Pseudo-code for Forward-Filtering Backward-Sampling, pairwise case . . 23
4.3 A Factor Tree 24
4.4 Sampling from a clique in a Factor Tree 25
4.5 Pseudo-code for Forward-Filtering Backward-Sampling, factor tree case 27
4.6 A MCMC Tree Partition 29
4.7 Two different MCMC tree partitions of a square lattice MRF 30
4.8 Sampling from one of the trees of a MCMC tree partition 30
4.9 An illegitimate MCMC tree partition for a Factor Graph 32
4.10 A corrected Factor Graph MCMC tree partition 33
4.11 Pseudo-code for the MCMC Tree Sampler algorithm 35

5.1 Pseudo-code outlining the structure of our partitioning algorithm 38

5.2 Simplification of a graph prior to partitioning 39
5.3 Coloring a graph for partitioning, part I 41
5.4 Coloring a graph for partitioning, part II 42
5.5 A comparison function defining an ordering on the priority queue 43
5.6 Partition of a graph, using an ordering on the priority queue 44
5.7 Suboptimal partitioning of a graph due to a trapped vertex 46
5.8 Another example of trapped vertices 47
5.9 Pseudo-code of the partitioning algorithm 49
5.10 Pseudo-code for the auxiliary functions of the partitioning algorithm . . 50
5.11 Partitioning a Factor Graph, part I 52
5.12 Partitioning a Factor Graph, part II 52
5.13 The final partition of a Factor Graph 53

6.1 Inference algorithms comparison: Fully connected pairwise graph 56
6.2 Inference algorithms comparison: Square-lattice MRF graph 58
6.3 Inference algorithms comparison: Random pairwise graph 59
6.4 Inference algorithms comparison: QMR graph with low leak 62
6.5 Inference algorithms comparison: QMR graph with medium leak 63
6.6 Inference algorithms comparison: QMR graph with high leak 64

Acknowledgements

I would like to thank all the people who helped me, in different ways, throughout my degree.

Academically, I am first and foremost grateful to my supervisor, Nando de Freitas. It is thanks to his teaching and support that I learnt probabilistic machine-learning. And it is thanks to its suggestions and advices that this thesis - hopefully - contains interesting results! Secondly, I would like to thank the whole Computer Science Department at UBC, and especially Firas Hamze for interesting discussions and for the initial ideas that form the core of this thesis.

I am also equally grateful to my whole family, especially my wife Marina, my brother Christian, my sister Marie, my parents Marie-France and Vincent, and my parents in-law Vera and Sergei. Their love and affection was absolutely essential for this thesis to see the light of the day, wether they were nearby or far away...

Finally, I would like to thank all my friends back in France. While I was away for two years, completing my Msc. degree in Vancouver, I have not forgotten them nor the times we had together. I sure hope to see you all again!

Chapter 1

Introduction

This thesis is about approximate inference in general discrete probabilistic models. The methods presented here do however extend to Gaussian models. They are applicable to general probabilistic graphical models such as directed acyclic graphs (DAGs, also known as Bayesian networks), conditional random fields, Markov random fields (MRFs) and factor graphs.

Performing inference on a large network (more than a thousand nodes) with a reasonable variable size (more than a hundred values) remains a huge computational task. Since the exact solution is impossible to compute, people generally turn to approximations. The Gibbs sampler is one of the most widely used approximate inference methods in this domain. However, correlation between variables in the probabilistic model can lead to a poor convergence rate for a single-site MCMC sampling technique. Various attempts at improving the robustness and efficiency of MCMC methods have thus been developed.

This thesis extends the *tree sampling* method, proposed in [6] for square-lattices Markov Random Fields, to arbitrary pairwise graphs and factor graphs. We also introduce new automatic partition schemes, obtaining results similar to those presented in Figure 1.1.

The tree sampler combines elements of Monte Carlo simulation as well as belief propagation. It requires that the graph be partitioned in trees first. The tree partitions of Figure 1.1 allow us to perform exact inference on each tree. These exact computations form the basis of a powerful blocked Gibbs sampler.

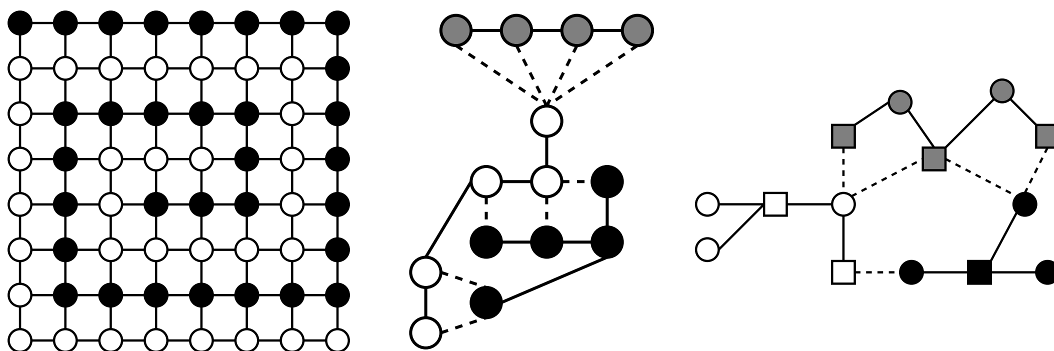


Figure 1.1: Several tree partitions for different types of graphs. On the left, a square-lattice MRF is partitioned into two trees, as in [6]. The middle figure is an example of a tree partition on an arbitrary pairwise graph, while the right figure corresponds to a factor graph.

This thesis is organized in the following manner. Chapters 2 and 3 are background chapters about probabilistic graphical models; in particular belief propagation (Chapter 2) and Monte Carlo methods (Chapter 3). They introduce all the necessary background material necessary for understanding the MCMC Tree Sampling algorithm and framework in Chapter 4.

From there we can describe in detail our partitioning algorithm, to which we devote entirely Chapter 5. Chapter 6 presents our experimental results.

All mathematical notation in this thesis is standard. We will designate a random variable with a bold letter $\mathbf{x}_i$, while the use of x_i will indicate one of its realizations. If a variable is conditioned on (observed), we will refer to its value using $\overline{x_i}$. We also adopt the notation $\mathbf{x}_{i:j} = \{\mathbf{x}_i, \mathbf{x}_{i+1}, \ldots, \mathbf{x}_{j-1}, \mathbf{x}_j\}$. The notation $\mathbf{x}_{i:j\backslash k}$ excludes the variable $\mathbf{x}_k$ from the set. Finally, in a graph, we denote by $\mathcal{N}(i)$ the neighbors of node i, and by $\mathcal{P}(i)$ its parents.

Chapter 2

Probability Propagation: Exact Inference

Probabilistic inference is at the heart of many scientific problems, including medical diagnosis, computer vision, image reconstruction and so on. In this chapter, we introduce probabilistic graphical models, which are a general framework for solving inference problems. We present the *belief propagation* algorithm, an instance of dynamic programming providing a solution to the exact inference problem.

2.1 Probabilistic Models

This section is a short introduction to probabilistic models. For an exhaustive background on probabilistic graphical models, see chapter 2 of Jordan's book [12], or Yedidia *et al.* [22].

A probabilistic model contains a set X of *random variables* $\mathbf{x}_{1:n}$. The set of possible values, for each variable, can be finite or not. Usually, some of these random variables will be *observed*, which means that their value is fixed and known, while others remain *hidden* (unknown).

We describe entirely a probabilistic model by specifying its *joint probability distribution*, which is a probability mass function $p(\mathbf{x}_{1:n})$ defined on the RVs. Using this distribution, one can answer any question about the model. We can, for instance, determine if a variable is independent of a subset of RVs, or compute conditional probabilities.

However, manipulating the joint distribution in an unfactorized representation is very costly. The graphical model framework proves to be an invaluable tool for solving multivariate statistical problems. Probabilistic graphical models are the result of a marriage between probabilistic modeling and graph theory. They allow us to express joint probability distributions in a factorized way, exploiting directly relationships between RVs.

On the following paragraphs, we will describe three graphical model representations: *directed acyclic graphs* [9], *pairwise Markov random fields* [14], and *factor graphs* [13]. Each of these models factorizes the joint distribution in a different way.

In fact, we can convert any of these three representations into another one. Details on the conversion process can be found in [22]. Since it requires the introduction of additional "artificial" random variables, it is better to represent a probabilistic model in its simplest form.

2.1.1 Directed Acyclic Graphs

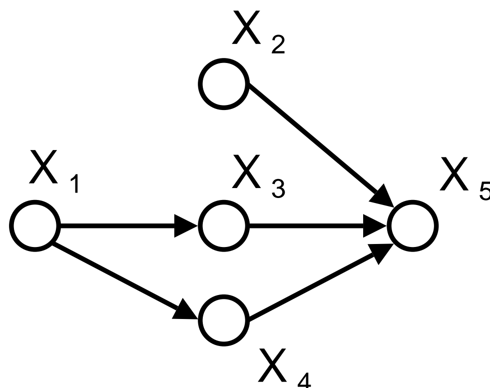


Figure 2.1: A directed acyclic graph with 5 random variables. $\mathbf{x}_5$ has 3 parents: $\mathcal{P}(\mathbf{x}_5) = \{\mathbf{x}_2, \mathbf{x}_3, \mathbf{x}_4\}$. $\mathbf{x}_1$ is the parent of $\mathbf{x}_3$ and $\mathbf{x}_4$.

DAGs are probably the most popular type of graphical model. Each node in a DAG is associated with a single random variable. Each edge in the graph is directed,

defining a parent node (by convention, the source of the edge) and a child node (the target of the edge).

The joint distribution of a DAG $\mathcal{G}(\mathcal{V}, \mathcal{E})$ factorizes as follows:

$$p(X) = \prod_{i \in \mathcal{V}} p(\mathbf{x}_i | \mathcal{P}(\mathbf{x}_i)) \tag{2.1}$$

where the $p(\mathbf{x}_i | \mathcal{P}(\mathbf{x}_i))$ represent directly conditional probabilities. They sum to one with respect to $\mathbf{x}_i$.

2.1.2 Pairwise Markov Random Fields

Pairwise Markov Random Fields belong to the group of undirected graphical models. Each node is again associated with a RV. We embed *potentials* in each variable node (noted as ϕ) and in the edges of the graph (noted as ψ). These potentials are arbitrary positive functions of one (ϕ) or two (ψ) RVs.

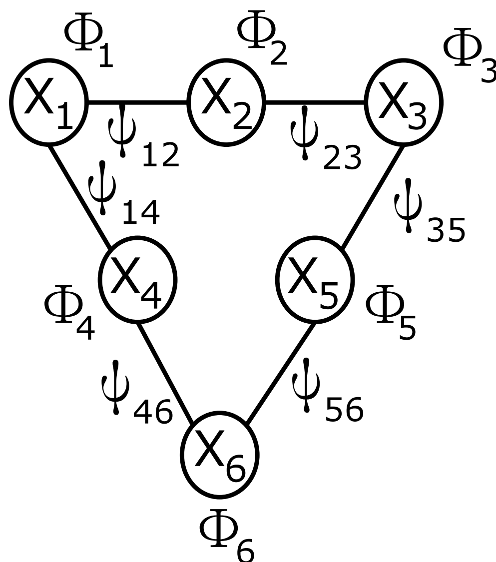


Figure 2.2: An example Pairwise Markov Random Field, with 6 random variables.

The joint probability equation of a pairwise MRF is written in the following form for a graph $\mathcal{G}(\mathcal{V}, \mathcal{E})$ with nodes $\mathcal{V}$ and edges $\mathcal{E}$:

$$p(X) = \frac{1}{Z} \prod_{i \in \mathcal{V}} \phi(\mathbf{x}_i) \prod_{(i,j) \in \mathcal{E}} \psi(\mathbf{x}_i, \mathbf{x}_j) \tag{2.2}$$

In computer vision, RVs on a MRF are often linked with four neighbors, forming a square lattice. There is also an additional observed node attached to every variable, as in Figure 2.3.

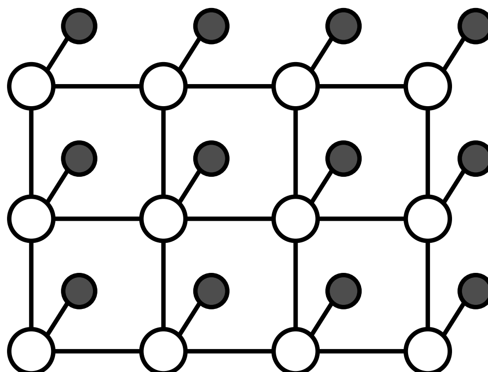


Figure 2.3: A square lattice pairwise Markov Random Field. The shaded circles correspond to observations and the white circles to the true "hidden" variables.

2.1.3 Factor Graphs

Factor graphs aim at capturing factorizations, rather than conditional probabilities like DAGs do. In a factor graph, vertices no longer correspond only to random variables. They also represent potentials. Each potential is associated to a *clique*, which is a subset of RVs[1]. We denote by $\mathcal{C} = \{C_1, C_2, \ldots, C_k\}$ the set of all cliques. The set of all corresponding potentials $\mathcal{P} = \{\psi_1, \psi_2, \ldots, \psi_k\}$ form the parameterization of the joint probability.

$$p(X) = \frac{1}{Z} \prod_{i \in \mathcal{V}} \phi(\mathbf{x}_i) \prod_{i \in \mathcal{P}} \psi_i(\mathbf{x}_{C_i}) \tag{2.3}$$

Equation (2.3) has effectively achieved factorization of the original joint probability distribution. Potentials represent how the different variables depend on each other. Each potential is an arbitrary positive function of the RVs of the corresponding clique. For a discrete problem with variables taking u different values, the size of a potential table associated with a clique of size s is u^s. For practical values of u, we are limited to cliques of size 10 at most.

[1]We also embed an internal potential ϕ in each RV, as in MRFs.

Graphically, a factor graph of n RVs and k potentials is represented by a graph of $n+k$ nodes. There is an edge between every potential node and the variables on which it depends. The convention is to represent potentials with square nodes and the random variables with round nodes, as illustrated in Figure 2.4.

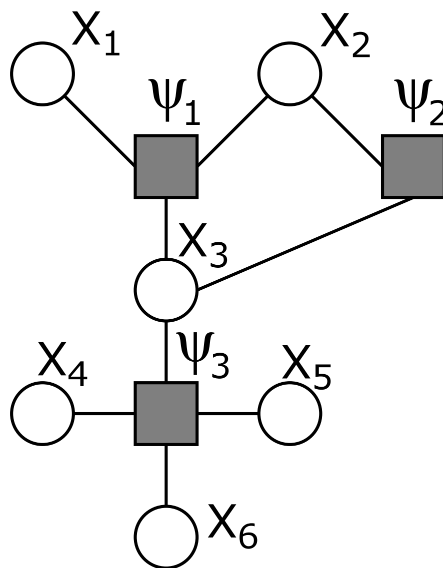


Figure 2.4: An example Factor Graph, with 6 random variables and 3 potentials. We have here $\mathcal{C} = \{\{\mathbf{x}_1, \mathbf{x}_2, \mathbf{x}_3\}, \{\mathbf{x}_2, \mathbf{x}_3\}, \{\mathbf{x}_3, \mathbf{x}_4, \mathbf{x}_5, \mathbf{x}_6\}\}$.

2.1.4 Inference on graphical models

Given two disjoint subsets X_A and X_B of X, the general probabilistic inference problem is to compute the distribution $p(X_A|X_B)$. X_B represents the subset of conditioned variables. Often, we will be interested in the marginals of every RV in X. We would then compute $p(\mathbf{x}_i|X_B)$ for $i \in \mathcal{V}$.

To compute these quantities, we must start from the expression of the joint probability $p(X)$ described in Equations (2.1), (2.2) and (2.3). We can obtain $p(\mathbf{x}_i|X_B)$ by summing in $p(X)$ over all configurations of the non conditioned variables.

$$p(\mathbf{x}_i|X_B) \propto \sum_{\mathbf{x}_{1:n} \setminus B \cup i} P(\mathbf{x}_{1:n}) \tag{2.4}$$

This naive way of computing marginals is impossible in practice, since it is clearly exponential in the number of variables. However, many summations are repeated. We should "reuse" intermediary summations and not compute them again. This idea of reusing intermediary factors leads to a general efficient solution to the exact inference problem for discrete *tree* graphs: the *sum-product* or *belief propagation* (BP) algorithm.

2.2 The Belief Propagation Algorithm

The belief propagation algorithm is well studied in the literature, descriptions being available in [19] or [22].

Belief propagation allows us to get the marginals of *all* variables, with a computational time only linear in the number of variables. Each node must send a carefully computed message to each of its neighbors, while respecting the following message-passing protocol:

A node can only send a message to a neighboring node when it has received messages from all its other neighbors.

In order for this to be possible, the graph must not contain any loops. BP only works on trees. This is its main limitation. We will investigate in Section 2.3, and later in Chapter 4, ways in which BP may still be used with arbitrary graphs.

On a tree, BP works as a "two-phase" algorithm. In the first phase, the messages flow from the leaves to the root. At the end of this phase, the root node has received all the messages from its neighbors. We could get the root node marginals at this point. But the real strength of BP is that we can get the marginals of *all* nodes just by doubling that amount of work. On the second phase we let the messages return from the root node to the leaves. Figure 2.5 illustrates this two-phase process.

In practice, we can use a depth-first traversal of the tree, starting from an arbitrary root, for both phases. In the first phase, each node will send a message to its *parent* when the depth-first traversal is finished with that node. In the second phase, the

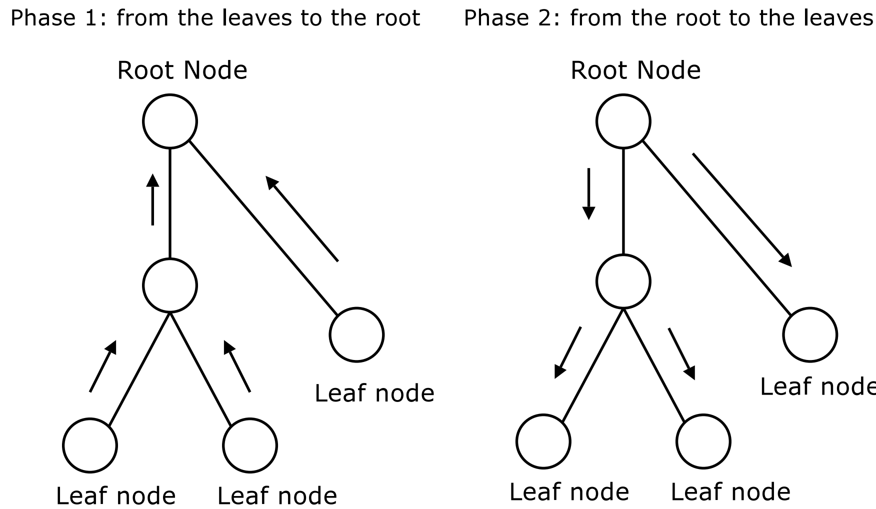


Figure 2.5: The flow of Belief-Propagation messages on an example tree. The directed arrows represent messages. The flow direction is inverted between the two phases.

parent of a discovered node (via depth-first traversal) will send to this discovered node (its child).

2.2.1 Belief Propagation for Factor Graphs

We will now look at the belief propagation algorithm implementation in the case of a discrete factor graph, also presented in [13]. We recall that a factor graph contains two different types of nodes, corresponding to random variables and to potentials. So we have two kinds of messages, the ones flowing from potentials to variables, $\mu_{i \to j}$ ($i \in \mathcal{P}$, $j \in X$), and the ones flowing from variables to potentials, $\nu_{i \to j}$ ($i \in X$, $j \in \mathcal{P}$).

2.2.1.1 Message propagation expressions

The expressions for our messages are given by the following pair of equations (see chapter 4 of [12]):

$$\text{From Variable to Potential:} \quad \nu_{i\to j}(x_i) = \phi(x_i) \prod_{k\in\mathcal{N}(i)\backslash j} \mu_{k\to i}(x_i) \tag{2.5}$$

$$\text{From Potential to Variable:} \quad \mu_{i\to j}(x_j) = \sum_{\mathbf{x}_{\mathcal{N}(i)\backslash j}} \psi(\mathbf{x}_{\mathcal{N}(i)}) \prod_{k\in\mathcal{N}(i)\backslash j} \nu_{k\to i}(x_k) \tag{2.6}$$

In terms of complexity, the $\mu_{i\to j}$ messages take most of the computational time. Assuming all variables can take the same number of values m, we carry out $m^{|\mathcal{N}(i)|}$ sums for potential i. In each of these sums, we have to compute a product of $|\mathcal{N}(i)|-1$ terms. By comparison, for the $\nu_{i\to j}$ messages, we compute m times a product of $|\mathcal{N}(i)|-1$ terms.

2.2.1.2 Incorporating evidence

Like in Chapter 4 of [12], we will use "evidence potentials" on the RVs to express conditioning (or observed variables). If RV $\mathbf{x}_i$ is conditioned and equal to $\overline{x_i}$, the evidence potential will be equal to $\delta(x_i, \overline{x_i})$. The final potential ϕ^E on RV $\mathbf{x}_i$ is the product of the evidence potential and the original potential ϕ.

$$\text{If } \mathbf{x}_i \text{ is conditioned:} \quad \phi^E(x_i) = \phi(x_i)\delta(x_i, \overline{x_i}) \tag{2.7}$$

$$\text{If } \mathbf{x}_i \text{ is not conditioned:} \quad \phi^E(x_i) = \phi(x_i)$$

This convenient transformation will be useful later in Chapter 4 for tree sampling. However, for simplicity, we will not denote explicitly the internal potentials as ϕ^E. We will keep writing them as ϕ, assuming the previous conversion has already taken place.

2.2.1.3 Obtaining the marginals of every random variable

Once messages have flowed twice (along each direction of the edges of the graph), we are ready to compute our marginals. We just take the products of all the messages

received at every variable node to obtain the marginals. Thus they are given (not normalized) by the following equation:

$$p(x_i) \propto \prod_{j \in \mathcal{N}(i)} \mu_{j \to i}(x_i) \tag{2.8}$$

2.2.1.4 Factor Graph BP pseudo-code

We present the complete pseudo-code of the sum-product algorithm in Figure 2.6. The implementation shown here uses two depth-first traversal of the tree.

In the initialization step, we need to record the *parent* of every node (the root will be its own parent). We usually do that with another depth-first traversal of the tree, which we combine with the depth-first traversal of phase 1.

In the first phase, we are sending from u to v, while we are sending from v to u in the second phase. This is of course to reverse the direction of the messages, as shown in Figure 2.5. Note also that in phase 1, we send a message when a vertex is finished. In phase 2, we send a message when a vertex is discovered.

2.2.2 Belief Propagation for Pairwise Markov Random Fields

The BP algorithm for Pairwise MRFs is a specialization of the one we just presented for factor graphs. The only changes appear in the expressions of the messages. Since every potential is linked to only one or two variables, we can elegantly combine Equations (2.5) and (2.6) into a single equation. The message $m_{i \to j}$ from a variable $\mathbf{x}_i$ to another one $\mathbf{x}_j$ is given by:

$$m_{i \to j}(x_j) = \sum_{x_i} \phi(x_i)\psi(x_i, x_j) \prod_{k \in \mathcal{N}(i) \setminus j} m_{k \to i}(x_i) \tag{2.9}$$

The message $m_{i \to j}$ is of the same size as $\mathbf{x}_j$, and not $\mathbf{x}_i$ (eg, it has the same number of values than $\mathbf{x}_j$ can take). However, the sum and product of messages occur over $\mathbf{x}_i$, the sending variable.

The unnormalized marginals are obtained in the same way as for factor graphs.

Initialization

- Choose an arbitrary root node r for the tree $\mathcal{T}$. For every node in $\mathcal{T}$, record its parent.

Messages propagation step

- Do a depth-first traversal of the tree $\mathcal{T}$, starting from the root node r. During the course of this traversal, whenever a node u is *finished* (e.g., when the exploration algorithm has returned from all its children):
  - Obtain u's parent, v. If $u = v$ (also meaning $u = r$: we are at the root), do nothing and go on to the next node. Else:
  - If $u \in \mathcal{V}$ (u is a variable node):

    Send the normalized message $\nu_{u\to v}$ from u to v, according to equation (2.5)
  - If $u \in \mathcal{P}$ (u is a potential node):

    Send the normalized message $\mu_{u\to v}$ from u to v, according to equation (2.6)
- Do a second depth-first traversal of the tree $\mathcal{T}$, starting from the root node r. During the course of this traversal, whenever a node u is *discovered* (e.g., when this node is first encountered by the exploration algorithm):
  - Obtain u's parent, v. If $u = v$ (also meaning $u = r$: we are at the root), do nothing and go on to the next node. Else:
  - If $v \in \mathcal{V}$ (v is a variable node):

    Send the normalized message $\nu_{v\to u}$ from v to u, according to equation (2.5)
  - If $v \in \mathcal{P}$ (v is a potential node):

    Send the normalized message $\mu_{v\to u}$ from v to u, according to equation (2.6)

Messages equations

$$\nu_{i\to j}(x_i) = \phi(x_i) \prod_{k\in\mathcal{N}(i)\setminus j} \mu_{k\to i}(x_i) \tag{2.5}$$

$$\mu_{i\to j}(x_j) = \sum_{\mathbf{x}_{\mathcal{N}(i)\setminus j}} \psi(\mathbf{x}_{\mathcal{N}(i)}) \prod_{k\in\mathcal{N}(i)\setminus j} \nu_{k\to i}(x_k) \tag{2.6}$$

Marginals

- For every variable $\mathbf{x}_i \in \mathcal{V}$,

$$\widetilde{p}(x_i) = \prod_{j\in\mathcal{N}(i)} \mu_{j\to i}(x_i)$$

- Normalize the marginals:

$$p(x_i) = \frac{\widetilde{p}(x_i)}{\sum_i \widetilde{p}(x_i)}$$

Figure 2.6: Belief Propagation algorithm for a Factor Graph $\mathcal{T}(\mathcal{V}, \mathcal{P})$. $\mathcal{V}$ represents the set of random variables, while $\mathcal{P}$ is the set of potentials attached to the graph.

We can write, similarly to Equation (2.8):

$$p(x_i) \propto \prod_{j \in \mathcal{N}(i)} m_{j \to i}(x_i) \tag{2.10}$$

2.3 Loopy Belief Propagation (LBP)

The sum-product algorithm can only be carried on a tree. On a graph with loops, it is impossible to respect the message passing protocol described in section 2.2 while ensuring that each node sends a message to each of its neighbors. However, Equations (2.5), (2.6) and (2.9) do not make any reference to a particular graph structure. By ignoring the message passing protocol, we can propagate the messages defined by the expressions of Sections 2.2.1.1 and 2.2.2 on *arbitrary graphs*[2].

Initialization

- For every $(u, v) \in \mathcal{E}$, set $m^0_{u \to v}(x_v) = 1$, for all possible values x_v of $\mathbf{x}_v$.

Messages propagation For $t = 1, \ldots, T$

- For every $u \in \mathcal{G}$:
  - For every $v \in \mathcal{N}(u)$:

$$m^t_{u \to v}(x_v) = \sum_{x_u} \psi(x_u, x_v) \prod_{k \in \mathcal{N}(u) \setminus v} m^{t-1}_{k \to u}(x_u)$$

Marginals

- For every variable $\mathbf{x}_u \in \mathcal{V}$,

$$\widetilde{p}(x_u) = \prod_{v \in \mathcal{N}(u)} \mu^T_{v \to u}(x_u)$$

Figure 2.7: Loopy Belief Propagation algorithm for a pairwise graph $\mathcal{G}(\mathcal{V}, \mathcal{E})$ and a predetermined number of steps T.

This process is named loopy belief propagation, and is probably the most popular

[2] The graphical model must still be discrete or Gaussian, which is the second limitation of Exact Belief Propagation.

algorithm based on belief propagation. Each of the nodes in the graph will send, in turn, a message to each of its neighbors. This is repeated for a certain number of steps. At step t, a node sends a new message m^t to its neighbors based on the messages received at step $t-1$. The pseudo-code is presented in Figure 2.7, with the message equations corresponding to a pairwise MRF.

LBP is an approximate inference algorithm, and there are no theoretical guarantees that it will converge to the true marginals. It can either fail to converge at all (*cycling* through different beliefs) or predict beliefs that are inaccurate. In Chapter 6, we will use LBP as a benchmark to compare the performance of our own inference algorithm, the tree sampler. LBP effectively fails to converge in some of our experiments. However, the study of the convergence properties of LBP could fill a chapter on its own. The reader can refer to [17] and [22] for a more in-depth presentation.

Chapter 3

Monte Carlo Methods

In the last chapter, we introduced probabilistic models and simple inference algorithms. In the current chapter, we will discuss a whole new class of algorithms: Monte Carlo methods. The first published paper on Monte Carlo, written by Metropolis and Ulam [15], dates back to 1949. The advent of massive cheap computational power has contributed to their "rediscovery" in the 1990s.

In essence, Monte Carlo methods work by generating many *samples*, and then approximating integrals and large sums by sample sums. The approximation gets better as more samples are obtained. If we were to generate an infinite number of samples, in most cases the law of large numbers would guarantee the correct exact results. The approximation obtained after a reasonable finite amount of time is excellent in many cases. For many applications, Monte Carlo methods are indeed the only viable algorithms.

3.1 Introducing Monte Carlo simulation

In Monte Carlo, one draws a large set of N samples $\{X^{(i)}\}_{i=1...N}$ from a *target distribution* $p(X)$ defined on a space $\mathcal{X}$. These N samples are used to approximate integrals of functions over the target distribution (a task usually intractable) by finite sums over the samples. The following equation illustrates this idea:

$$\int_{\mathcal{X}} f(X)p(X)dX \approx \frac{1}{N}\sum_{i=1}^{N} f(X^{(i)}) \tag{3.1}$$

In particular, we can obtain an approximation of the marginals by counting how many times every RV has been in a given state.

Many different *strategies* have been developed to explore the state space and generate the samples. Sampling from a high-dimensional space $\mathcal{X}$ is not an easy task. For example, sampling from Equation (2.3) is impossible for a factor graph of a reasonable size.

We will review MCMC methods in the next section. This class of Monte Carlo strategies is of special importance to us, since our own inference algorithm, introduced in Chapter 4, is based on MCMC.

3.2 Markov Chain Monte Carlo methods

A introduction to MCMC algorithms can be found in Andrieu *et al.* [1], and a more extensive treatment in Gilks *et. al* [7], Robert and Casella [5].

MCMC strategies use a Markov chain mechanism to explore the state space. This means that a given sample $X^{(i)}$ depends only on the previous sample $X^{(i-1)}$. This can be useful if we are trying to approximate a distribution where we can easily have an expression for the probability of a new sample given the last one. We will use this *transition distribution* to "jump" from one state (sample) to another, starting from an arbitrary initial state. If this transition distribution obeys some properties[1] (which is usually the case in practice), we can approximate the original distribution by MCMC sampling.

The most popular MCMC algorithm is the *Metropolis-Hastings* (MH) algorithm. In fact, most other MCMC algorithms are derived from this one, and can be interpreted as special cases of this basic method. We will not cover the MH algorithm but instead focus on the *Gibbs sampler*, an algorithm particularly suited to approximate inference on probabilistic graphical models.

3.2.1 The Gibbs sampler

If we have a n-dimensional vector $X = \{\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n\}$ and the expressions for all the full conditionals $\{p(\mathbf{x}_i|\mathbf{x}_1, \ldots, \mathbf{x}_{i-1}, \mathbf{x}_{i+1}, \ldots, \mathbf{x}_n)\}_{i=1\ldots n}$, we can sample in turn from

[1] The transition distribution should obey the *Irreducibility* and *Aperiodicity* properties. See [1].

each of these conditionals to generate a valid sample for Monte Carlo approximation.

We present a unified framework for running a Gibbs sampler on MRFs and factor graphs in Figure 3.1. The only differences appear in the equations giving the full conditional distributions. On a MRF, we obtain them with the following derivations:

$$p(\mathbf{x}_i \mid \mathbf{x}_{1:n\setminus i}) = \frac{P(\mathbf{x}_{1:n})}{\sum_{\mathbf{x}_i} P(\mathbf{x}_{1:n})}$$

$$= \frac{\frac{1}{Z}\prod_j \phi(\mathbf{x}_j)\prod_{(j,k)\in\mathcal{E}} \psi(\mathbf{x}_j, \mathbf{x}_k)}{\sum_{\mathbf{x}_i} \frac{1}{Z}\prod_j \phi(\mathbf{x}_j)\prod_{(j,k)\in\mathcal{E}} \psi(\mathbf{x}_j, \mathbf{x}_k)} \tag{3.2}$$

On Equation (3.2), the denominator is a sum, constant with respect to $\mathbf{x}_i$. The product on the nominator contains terms depending on $\mathbf{x}_i$, and terms independent of $\mathbf{x}_i$. Up to a proportionality constant, we can remove the independent terms, and write:

$$\text{MRF Full Conditionals:} \qquad p(\mathbf{x}_i \mid \mathbf{x}_{1:n\setminus i}) \propto \phi(\mathbf{x}_i) \prod_{j\in\mathcal{N}(i)} \psi(\mathbf{x}_i, \mathbf{x}_j) \tag{3.3}$$

For a factor graph, the derivations are similar and we obtain the following expression.

$$\text{Factor Graph Full Conditionals:} \qquad p(\mathbf{x}_i \mid \mathbf{x}_{1:n\setminus i}) \propto \phi(\mathbf{x}_i) \prod_{j\in\mathcal{N}(i)} \psi_j(\mathbf{x}_{C_j}) \tag{3.4}$$

Initialization

- Initialize $X^{(0)} = \{\mathbf{x}_1^{(0)}, \mathbf{x}_2^{(0)}, \ldots, \mathbf{x}_n^{(0)}\}$ at random.

Gibbs sampling steps

- For $t = 1, \ldots, T$
  - Sample $\mathbf{x}_1^{(t)} \sim p(\mathbf{x}_1 | \mathbf{x}_2^{(t-1)}, \ldots, \mathbf{x}_n^{(t-1)})$ according to Equation (3.5) or (3.6)

    $\vdots$
  - Sample $\mathbf{x}_i^{(t)} \sim p(\mathbf{x}_i | \mathbf{x}_1^{(t)}, \ldots, \mathbf{x}_{i-1}^{(t)}, \mathbf{x}_{i+1}^{(t-1)}, \ldots, \mathbf{x}_n^{(t-1)})$

    $\vdots$
  - Sample $\mathbf{x}_n^{(t)} \sim p(\mathbf{x}_n | \mathbf{x}_1^{(t)}, \ldots, \mathbf{x}_{n-1}^{(t)})$

Expression for the full conditionals

- For the Pairwise MRF case (Equation (3.3)):

$$p(\mathbf{x}_i^{(t)} = x_i | \mathbf{x}_1^{(t)}, \ldots, \mathbf{x}_{i-1}^{(t)}, \mathbf{x}_{i+1}^{(t-1)}, \ldots, \mathbf{x}_n^{(t-1)}) \propto \phi(x_i) \prod_{j \in \mathcal{N}(i)} \psi(x_i, \overline{x_j}) \tag{3.5}$$

- For the Factor Graph case (Equation (3.4)):

$$p(\mathbf{x}_i^{(t)} = x_i | \mathbf{x}_1^{(t)}, \ldots, \mathbf{x}_{i-1}^{(t)}, \mathbf{x}_{i+1}^{(t-1)}, \ldots, \mathbf{x}_n^{(t-1)}) \propto \phi(\mathbf{x}_i) \prod_{j \in \mathcal{N}(i)} \psi_j(x_i, \overline{x_{C_j \setminus x_i}}) \tag{3.6}$$

In these equations, $\overline{x_j} = x_j^{(t)}$ if $j < i$, or $\overline{x_j} = x_j^{(t-1)}$ if $j < i$.

Marginals

- For every variable $\mathbf{x}_i$:

$$\widetilde{p}(\mathbf{x}_i = x_i) = \frac{1}{T+1} \sum_{t=0}^{T} \delta(\mathbf{x}_i^{(t)} = x_i)$$

Figure 3.1: The Gibbs sampler algorithm.

Chapter 4

And among these I hold trees dear.

The Silmarillion J.R.R. TOLKIEN

Combining Monte Carlo and Belief Propagation: MCMC Tree Sampling

The two previous chapters presented different methods for performing inference on probabilistic graphical models. Belief propagation allows for exact inference, but its range of application is limited. On the other hand, Monte Carlo simulation is more practical, but remains an approximation. It fails to exploit fully the structural properties of the underlying graphical model.

This chapter will build on Chapters 2 and 3 in order to present a strategy for combining Monte Carlo simulation and exact belief propagation.

4.1 Improving MCMC algorithms

Basic MCMC algorithms such as the naive Gibbs sampler tend to be slow. They need lots of samples to converge. Over recent years, several different schemes have been used to try to improve the convergence of MCMC algorithms.

4.1.1 From single sampling to group sampling

An MCMC sample consists of a full instantiation of the random variables in the model. For interesting problems, it is impossible to sample directly from the full joint probability. In the Gibbs sampler, one random variable is sampled at a time. However, this is problematic when it is difficult to move through the target distribution in "one di-

mension"[1]. If we start in a low-probability region of the state space, it will take many samples to get to the interesting region. Figure 4.1 illustrates this idea graphically.

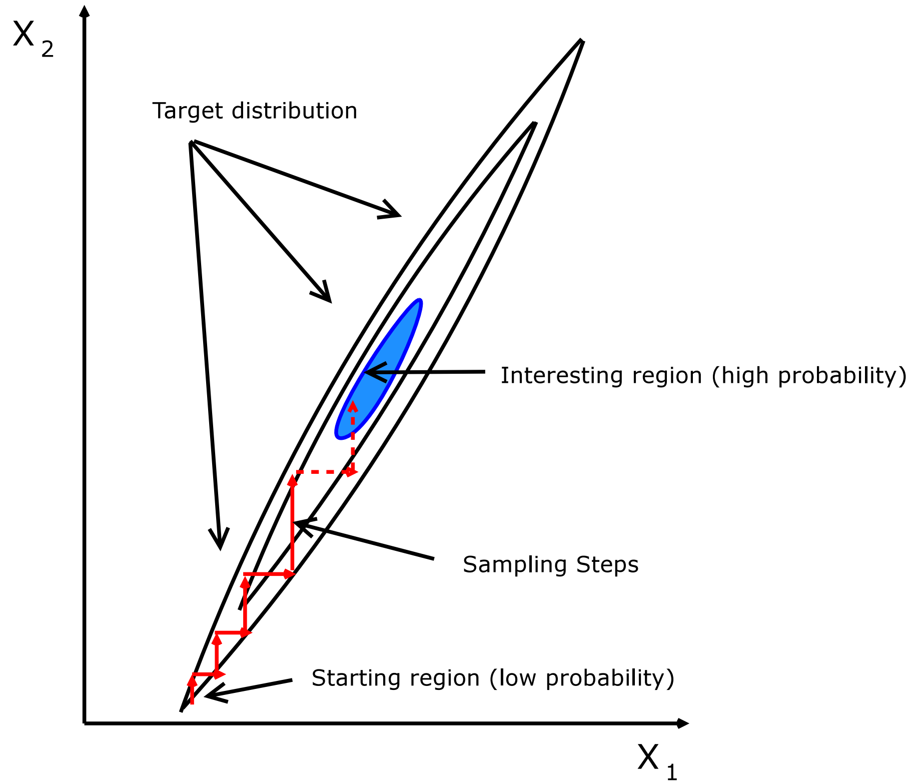


Figure 4.1: A example distribution in two dimensions ($X = (\mathbf{x}_1, \mathbf{x}_2)$), which is hard to approximate by MCMC methods. If we use Gibbs sampling, we sample $\mathbf{x}_1$ conditioned on $\mathbf{x}_2$, and vice versa. The individual sampling steps are drawn by a red line in one dimension. It is hard to get to the region where most of the distribution weight (density) is, since we start in a remote region.

Note that if we could sample from the full joint distribution from the start, convergence would be much faster as we would instantly move to the regions of high-probability. Said otherwise, "moving in two dimensions" is dramatically better in Figure 4.1. This example demonstrates a weakness of the naive Gibbs sampler, and motivates the search for better solutions.

[1]By "moving in one dimension", we mean that we will sample a variable conditioned on all the others (Gibbs). Our sampling distribution is thus effectively in one dimension.

Generally, we would like to exploit the structural properties of the graphical model. RVs should thus be sampled in *blocks*[2], or groups, to avoid the situation of Figure 4.1. However, the trade-off is that sampling in groups is harder. The key here is to create specially crafted groups, where sampling can be done efficiently.

4.1.2 A special case of group sampling: tree sampling

There is an efficient way for sampling on trees called *forward-filtering backward-sampling* (FFBS), described by Carter and Kohn in [4] and Wilkinson and Yeung in [21]. This algorithm allows us to draw independent samples of the whole tree. We describe it in the next section for MRFs and factor graphs. Sampling on a factor tree is an extension of the original algorithm. It is one of the main contribution of this thesis.

4.2 Sampling from Trees: Forward Filtering Backward Sampling

4.2.1 Pairwise MRF Trees

For pairwise graphs, the key to tree sampling is that we can decompose the full probability $p(X)$ as follows:

$$\begin{aligned} p(X) &= p(\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n) \\ &= p(\mathbf{x}_n | \mathbf{x}_1, \ldots, \mathbf{x}_{n-1}) p(\mathbf{x}_1, \ldots, \mathbf{x}_{n-1}) \end{aligned}$$

Here $\mathbf{x}_n$ is a leaf in the tree, which means that given its parent $\mathcal{P}(\mathbf{x}_n)$, $\mathbf{x}_n$ does not depend on *any other variables* of the graph. This is clear by looking at Equation (2.2). Thus we write:

$$p(X) = p(\mathbf{x}_n | \mathcal{P}(\mathbf{x}_n))\ p(\mathbf{x}_1, \ldots, \mathbf{x}_{n-1})$$

[2]Blocking in MCMC or in other sampling techniques is a well-discussed topic in the machine learning literature. Various blocking schemes have been developed: see for instance [11] and [21] for Gibbs blocking, and [3] for cutsets in Bayesian networks.

If there are other leaves in $\mathcal{G}$, we can repeat this procedure for $p(\mathbf{x}_1, \ldots, \mathbf{x}_{n-1})$, yielding:

$$p(X) = p(\mathbf{x}_1) \prod_{i>1} p(\mathbf{x}_i | \mathcal{P}(\mathbf{x}_i)) \tag{4.1}$$

This is in fact the same expression as the factorization for the joint probability distribution of a DAG in Equation (2.1). On a DAG, by choosing a suitable sampling ordering, we can directly generate a sample of the whole tree.

On an undirected model, we don't have expressions for $p(\mathbf{x}_1)$ and the conditional probabilities $p(\mathbf{x}_i|\mathcal{P}(\mathbf{x}_i))$. We first obtain $p(\mathbf{x}_1)$ by carrying out the first phase of Belief-Propagation, the one where the messages flow from the leaves to the root $\mathbf{x}_1$. To compute the conditional probabilities, we write:

$$\begin{aligned} p(\mathbf{x}_i|\mathcal{P}(\mathbf{x}_i)) &\propto \prod_{j \in \mathcal{N}(i)} m_{j \to i}(x_i) \\ &\propto m_{\mathcal{P}(i) \to i}(x_i) \prod_{j \in \mathcal{N}(i) \setminus \mathcal{P}(i)} m_{j \to i}(x_i) \end{aligned} \tag{2.10}$$

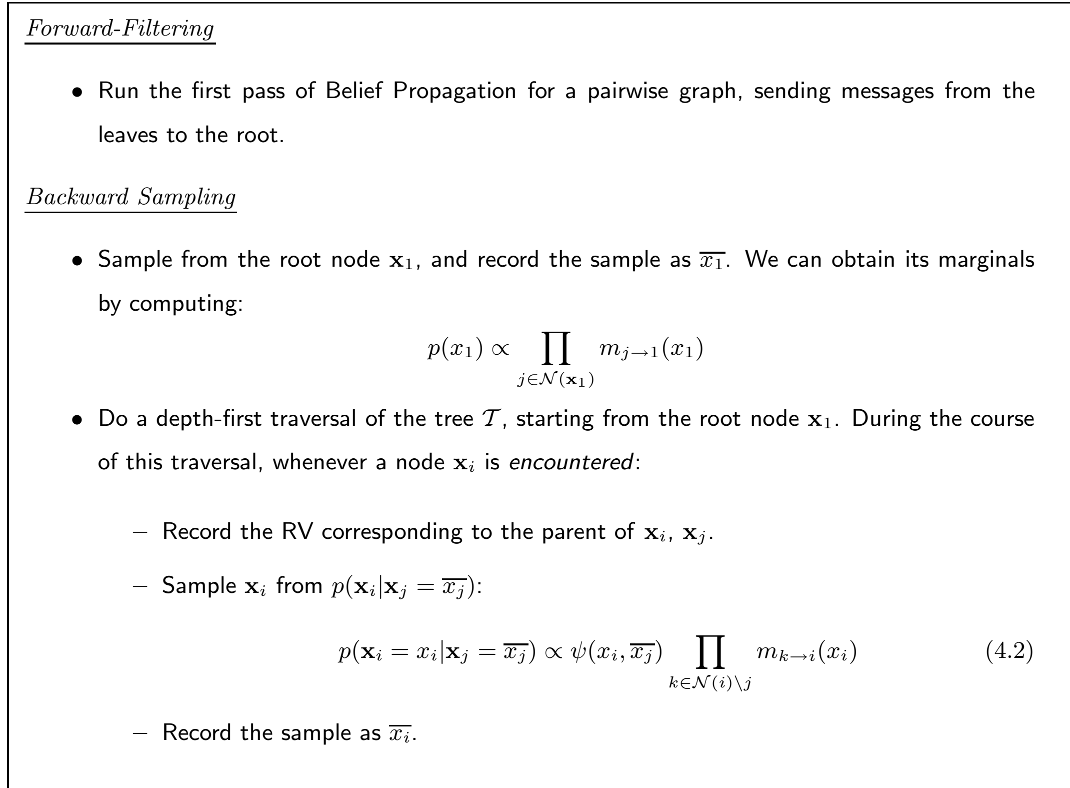
The messages $m_{j \to i}$, when j is not the parent of i, have already been computed in the first phase of BP. Now let us compute $m_{j \to i}$ with $j = \mathcal{P}(i)$. With the explicit notation for evidence potentials of Equation (2.7), we can derive the following expression:

$$\begin{aligned} m_{j \to i}(x_i) &= \sum_{x_j} \phi^E(x_j) \psi(x_i, x_j) \prod_{k \in \mathcal{N}(j) \setminus i} m_{k \to j}(x_j) \\ &= \sum_{x_j} \phi(x_j) \delta(x_j, \overline{x_j}) \psi(x_i, x_j) \prod_{k \in \mathcal{N}(j) \setminus i} m_{k \to j}(x_j) \\ &= \phi(\overline{x_j}) \psi(x_i, \overline{x_j}) \prod_{k \in \mathcal{N}(j) \setminus i} m_{k \to j}(\overline{x_j}) \\ &\propto \psi(x_i, \overline{x_j}) \end{aligned}$$

This finally gives us the following important equation for the conditional probabilities:

$$p(\mathbf{x}_i = x_i | \mathcal{P}(\mathbf{x}_i) = \overline{\mathcal{P}(\mathbf{x}_i)}) \propto \psi(x_i, \overline{\mathcal{P}(\mathbf{x}_i)}) \prod_{j \in \mathcal{N}(i) \setminus \mathcal{P}(i)} m_{j \to i}(x_i) \tag{4.2}$$

With these expressions, we can obtain a sample of the whole tree. We start by sampling the root $\mathbf{x}_1$. We then continue to sample down the tree, from the root to the leaves, using Equation (4.2). Figure 4.2 describes the complete pseudo-code for FFBS on a pairwise undirected graph.



Forward-Filtering

- Run the first pass of Belief Propagation for a pairwise graph, sending messages from the leaves to the root.

Backward Sampling

- Sample from the root node $\mathbf{x}_1$, and record the sample as $\overline{x_1}$. We can obtain its marginals by computing:

$$p(x_1) \propto \prod_{j \in \mathcal{N}(\mathbf{x}_1)} m_{j \to 1}(x_1)$$

- Do a depth-first traversal of the tree $\mathcal{T}$, starting from the root node $\mathbf{x}_1$. During the course of this traversal, whenever a node $\mathbf{x}_i$ is *encountered*:
  - Record the RV corresponding to the parent of $\mathbf{x}_i$, $\mathbf{x}_j$.
  - Sample $\mathbf{x}_i$ from $p(\mathbf{x}_i | \mathbf{x}_j = \overline{x_j})$:

$$p(\mathbf{x}_i = x_i | \mathbf{x}_j = \overline{x_j}) \propto \psi(x_i, \overline{x_j}) \prod_{k \in \mathcal{N}(i) \setminus j} m_{k \to i}(x_i) \tag{4.2}$$

  - Record the sample as $\overline{x_i}$.

Figure 4.2: Forward-Filtering Backward-Sampling for a discrete pairwise tree $\mathcal{T}$.

4.2.2 Factor Trees

Equation (4.1) is no longer valid for factor graphs. A leaf in a factor tree can only correspond to a RV node. This RV no longer depends only on its "parent", which is a potential node. It depends on all the neighbors of the parent potential.

We achieve the decomposition of the full probability $p(X)$ through a more complex independence property. Given the parent $\mathbf{x}$ of a clique ψ, the RVs that are further down the tree, after ψ, are independent of all the other RVs in the graph. Figure 4.3

illustrates this.

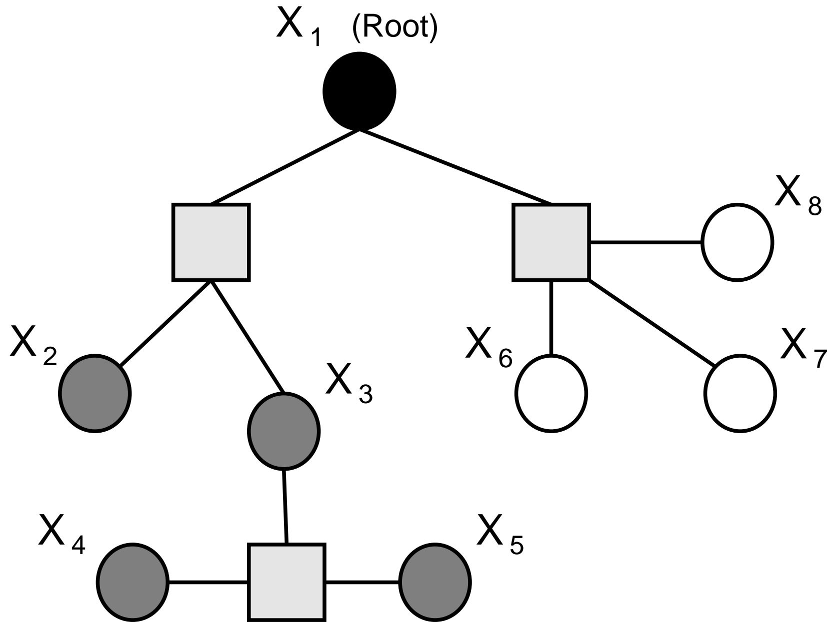


Figure 4.3: An example factor tree. Given the black RV $\mathbf{x}_1$, the gray variables $\mathbf{x}_{2:5}$ are conditionally independent of the white variables $\mathbf{x}_{6:8}$. Similarly, given the parent $\mathbf{x}_3$ of clique $\mathbf{x}_{3:5}$, $\mathbf{x}_4$ and $\mathbf{x}_5$ are independent of all other RVs.

On this example graph, we would write the following derivation:

$$p(X) = p(\mathbf{x}_1)p(\mathbf{x}_{2:8}|\mathbf{x}_1)$$

We now use our independence property. Given $\mathbf{x}_1$, the sets $\mathbf{x}_{2:5}$ and $\mathbf{x}_{6:8}$ are conditionally independent.

$$\begin{aligned} p(X) &= p(\mathbf{x}_1)p(\mathbf{x}_{2:5}|\mathbf{x}_1)p(\mathbf{x}_{6:8}|\mathbf{x}_1) \\ &= p(\mathbf{x}_1)p(\mathbf{x}_2, \mathbf{x}_3|\mathbf{x}_1)p(\mathbf{x}_4, \mathbf{x}_5|\mathbf{x}_{1:3})p(\mathbf{x}_{6:8}|\mathbf{x}_1) \end{aligned}$$

We use the independence property one more time for the clique $\{\mathbf{x}_3, \mathbf{x}_4, \mathbf{x}_5\}$, and finally obtain:

$$p(X) = p(\mathbf{x}_1)p(\mathbf{x}_2, \mathbf{x}_3|\mathbf{x}_1)p(\mathbf{x}_4, \mathbf{x}_5|\mathbf{x}_3)p(\mathbf{x}_{6:8}|\mathbf{x}_1)$$

We can write an equation for a general graph $\mathcal{G}(\mathcal{V}, \mathcal{P})$. We denote $\mathcal{P}(i)$ the index of the parent of clique C_i, and choose $\mathbf{x}_1$ as the root of the factor tree. We obtain this

important decomposition:

$$p(X) = p(\mathbf{x}_1) \prod_{\psi_i \in \mathcal{P}} p(\mathbf{x}_{C_i \setminus \mathcal{P}(i)} | \mathbf{x}_{\mathcal{P}(i)}) \tag{4.3}$$

Once we have factorized the joint distribution as in Equation (4.3), we must decompose each clique individually. For a potential $\psi_i \in \mathcal{P}$, let the clique variables be $C_i = \{\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_k\}$, where $\mathbf{x}_1$ is the parent. We write the following general factorization:

$$p(\mathbf{x}_{2:k}|\mathbf{x}_1) = p(\mathbf{x}_2|\mathbf{x}_1)p(\mathbf{x}_3|\mathbf{x}_{1:2}) \ldots p(\mathbf{x}_{k-1}|\mathbf{x}_{1:k-2})p(\mathbf{x}_k|\mathbf{x}_{1:k-1}) \tag{4.4}$$

Given Equations (4.3) and (4.4), we can obtain a sample of the whole factor tree. We start by sampling the root $\mathbf{x}_1$, and then sample each clique in turn, moving down the tree from the root to the leaves.

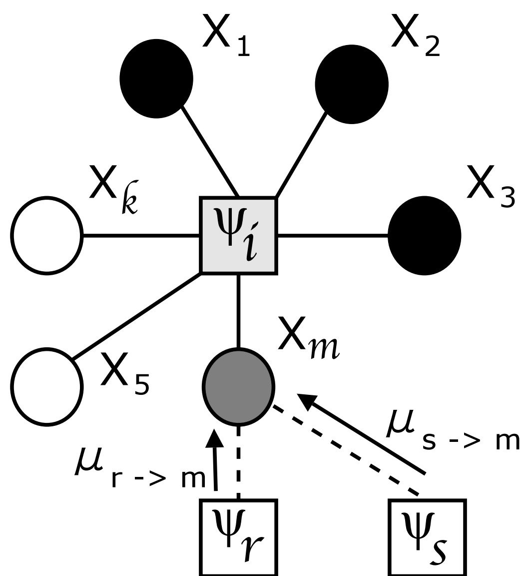


Figure 4.4: The process of sampling from a clique C_i, with $k = 6$. The black RVs $\mathbf{x}_{1:3}$ have already been sampled. We sample the gray variable $\mathbf{x}_m$ ($m = 4$). The last two white RVs $\mathbf{x}_5$ and $\mathbf{x}_6$ remain to be sampled.

To sample from a clique, we must compute the conditional probabilities present in Equation (4.4) with a run of BP. Let us assume that we are sampling from the example

of Figure 4.4. We have a clique C_i with k variables, $\mathbf{x}_{1:k}$. We have already sampled RVs $\mathbf{x}_{1:m-1}$, and the other RVs $\mathbf{x}_{m:k}$ have not been sampled yet. We are going to sample $\mathbf{x}_m$ and thus compute $p(\mathbf{x}_m|\mathbf{x}_{1:m-1})$.

$$p(\mathbf{x}_m|\mathbf{x}_{1:m-1}) \propto \prod_{j\in\mathcal{N}(m)} \mu_{j\to m}(x_m) \tag{2.8}$$

$$\propto \mu_{i\to m}(x_m) \prod_{j\in\mathcal{N}(m)\setminus i} \mu_{j\to m}(x_m) \tag{4.5}$$

The messages $\mu_{j\to m}$, when $j \neq i$ (in Figure 4.4, $\mu_{r\to m}$ and $\mu_{s\to m}$) correspond to cliques other than the one we are currently sampling from. They have already been computed in the first phase of BP. We now compute $\mu_{i\to m}$:

$$\begin{aligned}
\mu_{i\to m}(x_m) &= \sum_{\mathbf{x}_{\mathcal{N}(i)\setminus m}} \psi_i(\mathbf{x}_{1:k}) \prod_{l\in\mathcal{N}(i)\setminus m} \nu_{l\to i}(\mathbf{x}_l) \\
&= \sum_{\mathbf{x}_{\mathcal{N}(i)\setminus m}} \psi_i(\mathbf{x}_{1:k}) \prod_{l=1:m-1} \nu_{l\to i}(\mathbf{x}_l) \prod_{l=m+1:k} \nu_{l\to i}(\mathbf{x}_l) \\
&= \sum_{\mathbf{x}_{\mathcal{N}(i)\setminus m}} \psi_i(\mathbf{x}_{1:k}) \prod_{l=1:m-1} \left(\delta(\mathbf{x}_l, \overline{\mathbf{x}_l})\phi(\mathbf{x}_l) \prod_{j\in\mathcal{N}(l)\setminus i} \mu_{j\to l}(\mathbf{x}_l)\right) \prod_{l=m+1:k} \nu_{l\to i}(\mathbf{x}_l)
\end{aligned}$$

Since RVs $\mathbf{x}_{1:m-1}$ have already been sampled, the messages $\nu_{l\to i,\ l<m}$ use evidence potentials. The sum in the last equation occurs only over RVs $\mathbf{x}_{m+1:k}$. We finally obtain:

$$\mu_{i\to m}(x_m) \propto \sum_{\mathbf{x}_{m+1:k}} \psi_i(\overline{\mathbf{x}_{1:m-1}}, x_m, \mathbf{x}_{m+1:k}) \prod_{l=m+1:k} \nu_{l\to i}(\mathbf{x}_l) \tag{4.6}$$

Figure 4.5 describes the full FFBS pseudo-code for factor trees.

4.3 MCMC Tree Sampling

The idea of MCMC Tree Sampling [6] is to partition the graph $\mathcal{G}$ into a set of l disjoint trees, $T = \{\mathcal{T}_1, \ldots, \mathcal{T}_l\}$. Then, if all random variables belonging to $\mathcal{T}_2, \ldots, \mathcal{T}_l$ are observed, $\mathcal{T}_1$ can effectively be considered as an *independent* tree. We can then produce a full joint sample of all the variables $\mathbf{x}_i \in \mathcal{T}_1$, using the method described in

Forward-Filtering

- Run the first pass of Belief Propagation, sending μ and ν messages from the leaves to the root. Refer to Figure 2.6.

Backward Sampling

- Sample from the root node $\mathbf{x}_1$, and record the sample as $\overline{x_1}$. We can obtain its marginals by computing:

$$p(x_1) \propto \prod_{p \in \mathcal{N}(\mathbf{x}_1)} \mu_{p \to 1}(x_1)$$

- Do a depth-first traversal of the tree $\mathcal{T}$, starting from the root node $\mathbf{x}_1$. During the course of this traversal, whenever a node corresponding to a potential ψ_i is *encountered*:
  - We note $C_i = \{\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_k\}$ the clique corresponding to ψ_i, with $\mathbf{x}_1$ the parent of C_i.
  - For $m = 2, \ldots, k$:
    * Compute message $\mu_{i \to m}$:

$$\mu_{i \to m}(x_m) \propto \sum_{\mathbf{x}_{m+1:k}} \psi_i(\overline{\mathbf{x}_{1:m-1}}, x_m, \mathbf{x}_{m+1:k}) \prod_{l=m+1:k} \nu_{l \to i}(\mathbf{x}_l) \tag{4.6}$$

    * Sample $\mathbf{x}_m$ from $p(\mathbf{x}_m | \overline{x_{1:m-1}})$:

$$p(\mathbf{x}_m = x_m | \overline{x_{1:m-1}}) \propto \mu_{i \to m}(x_m) \prod_{j \in \mathcal{N}(m) \setminus i} \mu_{j \to m}(x_m) \tag{4.5}$$

    * Record the sample as $\overline{x_m}$.

Figure 4.5: Forward-Filtering Backward-Sampling for a discrete factor tree $\mathcal{T}$.

the previous section. Of course, this is true for every tree $\mathcal{T}$ in the partition T. Given all other trees[3], we can thus obtain a full joint sample for any single tree $\mathcal{T}$.

Using this idea, we construct a blocked Gibbs sampler. We don't sample individual variables one at a time, but entire *trees*. In fact, the "simple" Gibbs sampler can be seen as a special case of our more general tree sampler, where all trees are just in fact single variables.

[3] *Given all other trees* actually means "given all the variables in the other trees".

Before describing the MCMC tree sampler implementation, we enumerate the different rules needed to generate correct tree partitions for MCMC sampling.

4.3.1 Tree partitions for Pairwise Markov Random Fields

For a pairwise MRF $\mathcal{G}(\mathcal{V}, \mathcal{E})$, the problem of choosing a tree partition can be reduced to the mathematical problem of finding a set of l disjoint trees $\mathcal{T}_1(\mathcal{V}_1, \mathcal{E}_1), \ldots, \mathcal{T}_l(\mathcal{V}_l, \mathcal{E}_l)$ covering $\mathcal{G}$. The partition must comply with the following rules:

Rule 1. *Every vertex and edge in a tree $\mathcal{T} \in T$ must also be present in the original graph $\mathcal{G}$.*

Rule 2. *Each vertex must be present in one of the trees of T, and only in one tree. That is, the set of trees is disjoint.*

Rule 3. *For each tree $\mathcal{T}$, and each pair of vertices (i, j) in $\mathcal{T}$, if there exists an edge between i and j in the original graph, then the same edge must be present in $\mathcal{T}$.*

We will delete some edges so that there are no edges between different trees. A deleted edge will correspond to a link between a variable belonging to the tree that we are currently sampling, and another variable in a different tree. This other RV will be considered *observed* while we sample the current tree. The last rule can thus be written as:

Rule 4. *For each edge (i, j) in the original graph $\mathcal{G}$, if vertices i and j belong to different trees in the tree partition, then $(i, j) \notin \mathcal{E}_k$, for $k \in \{1, \ldots, l\}$.*

These four rules can be combined into the following formal definition.

Definition 4.3.1. Given a graph $\mathcal{G}(\mathcal{V}, \mathcal{E})$, an MCMC tree partition is a set of l trees $\{\mathcal{T}_1(\mathcal{V}_1, \mathcal{E}_1), \ldots, \mathcal{T}_l(\mathcal{V}_l, \mathcal{E}_l)\}$ satisfying the following conditions:

$$\forall k \in \{1, \ldots, l\}, \mathcal{V}_k \subset \mathcal{V}, \mathcal{E}_k \subset \mathcal{E} \tag{4.7}$$

$$\bigcup_{i=1}^{l} \mathcal{V}_i = \mathcal{V} \tag{4.8}$$

$$\forall (i,j) \in \{1,\ldots,l\}^2, \mathcal{V}_i \cap \mathcal{V}_j = \emptyset \tag{4.9}$$

$$\forall k \in \{1,\ldots,l\}, \forall (i,j) \in \mathcal{V}_k^2, (i,j) \in \mathcal{E} \Rightarrow (i,j) \in \mathcal{E}_k \tag{4.10}$$

Rule 1 gives (4.7); Rule 2 gives both (4.8) and (4.9). Rule 3 gives (4.10), while Rule 4 does not need to be formulated explicitly, as tree edges can occur only between vertices actually in the tree.

An MCMC tree partition is just a disjoint covering of the graph's vertices by several trees. We are allowed to discard edges between two distinct trees. Figure 4.6 gives an example of a valid MCMC tree partition on a small graph.

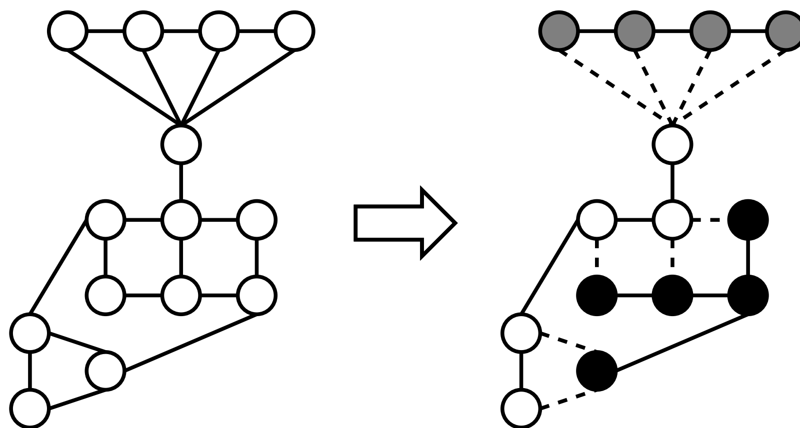


Figure 4.6: The original pairwise 14 nodes graph with several loops (left) has been partitioned into 3 trees (nodes in white, grey, and black) on the right. Edges that link vertices belonging to the same tree appear solid, while edges that are "left-out" appear dotted. Note that it is impossible, for this graph, to obtain an MCMC tree partition consisting of only 2 trees.

It is also interesting to note that a square lattice MRF of an arbitrary size can always be partitioned in two trees using a "comb" partition. In fact, it can even be partitioned in two chains using a spiraling pattern. Figure 4.7 shows these partitions on a 8 by 8 square lattice MRF.

These tree partitions allow us to sample variables in groups. Given all other trees, we can sample an individual tree in one pass according to Section 4.2.1. Potentials corresponding to edges between different trees depend originally on two RVs. While one

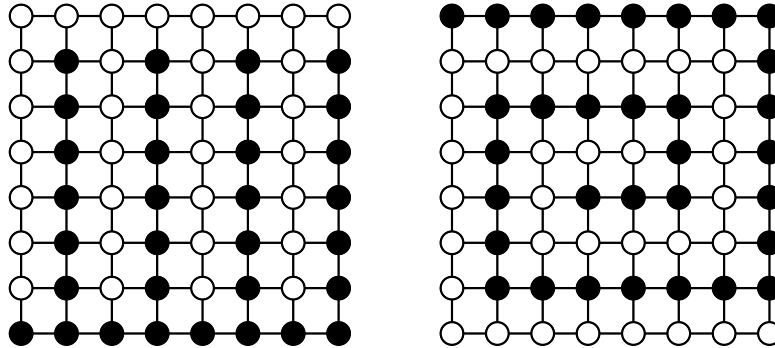


Figure 4.7: Two different MCMC tree partitions of a square lattice MRF, using a comb pattern (left) and a spiraling pattern (right).

tree is conditioned (observed), they become potentials of a single variable. Figure 4.8 illustrates this.

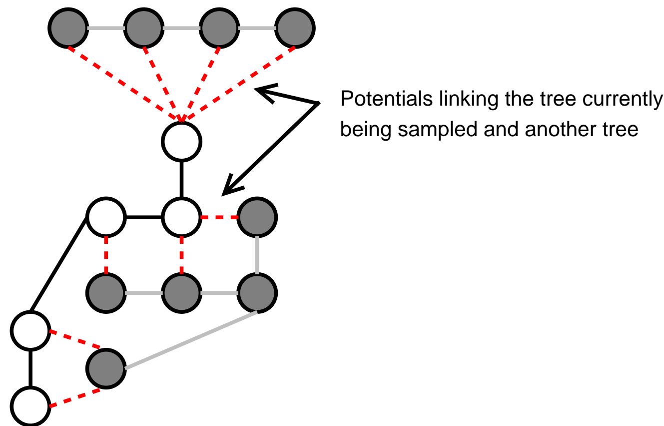


Figure 4.8: The process of sampling from one tree in an MCMC tree partition. The variables in white are currently sampled, while shaded nodes correspond to variables conditioned. The red edges (dotted) correspond to potentials that are linked both to a white and shaded variable. Conditioned on the shaded variable, they depend on a single variable.

For any RV $\mathbf{x}_i$ in the tree $\mathcal{T}$ that we are currently sampling, we look at its edges. If an edge goes to a RV $\mathbf{x}_j$ in a different tree, we incorporate the potential ψ associated with this edge with the local potential ϕ of $\mathbf{x}_i$. We do so by taking the product of ψ,

conditioned on $\mathbf{x}_j$, and ϕ:

$$\phi_{new}(x_i) = \phi(x_i) \prod_{(i,j)\in\mathcal{E}, j\notin\mathcal{T}} \psi(x_i, \overline{x_j}) \tag{4.11}$$

Once we have incorporated all these potentials onto ϕ_{new}, we are then ready to use the tree sampling method of section 4.2.1.

4.3.2 Choosing a tree partition for Factor Graphs

Unfortunately, the rules for obtaining an MCMC tree partition for a Factor Graph are different than those for a pairwise graph. One could naively think of a factor graph as a pairwise graph for the purposes of generating an MCMC tree partition[4] and apply the rules of section 4.3.1. This would not work, because the property that the potentials should depend on a single variable (when conditioned on all trees except the one we are currently sampling) no longer holds. Figure 4.9 will help the reader understand the issue at hand.

In order to avoid this case, we need to add another rule to our existing four rules:

Rule 5. *For each tree $\mathcal{T}$ in the MCMC tree partition T, and each potential ψ in $\mathcal{T}$, if $C = \{\mathbf{x}_1, \ldots, \mathbf{x}_k\}$ is the set of RVs ψ is linked to,* at most *one variable $\mathbf{x}_i \in C$ may be in each single other tree in T.*

This ensures that a potential ψ belonging to a conditioned tree will not pose problems when sampling the current tree. Again, let us write a formal mathematical definition for a MCMC tree partition, in the factor graph case.

Definition 4.3.2. Given a factor graph $\mathcal{G}(\mathcal{V}(X,P),\mathcal{E})$, a MCMC tree partition is a set of l trees $\{\mathcal{T}_1(\mathcal{V}_1(X_1,P_1),\mathcal{E}_1),\ldots,\mathcal{T}_l(\mathcal{V}_l(X_l,P_l),\mathcal{E}_l)\}$ satisfying the following conditions:

$$\forall k \in \{1,\ldots,l\}, \mathcal{V}_k \subset \mathcal{V}, \mathcal{E}_k \subset \mathcal{E} \tag{4.7}$$

$$\bigcup_{i=1}^{l} \mathcal{V}_i = \mathcal{V} \tag{4.8}$$

[4]That is, consider all vertices equal, whether they correspond to RVs or potentials.

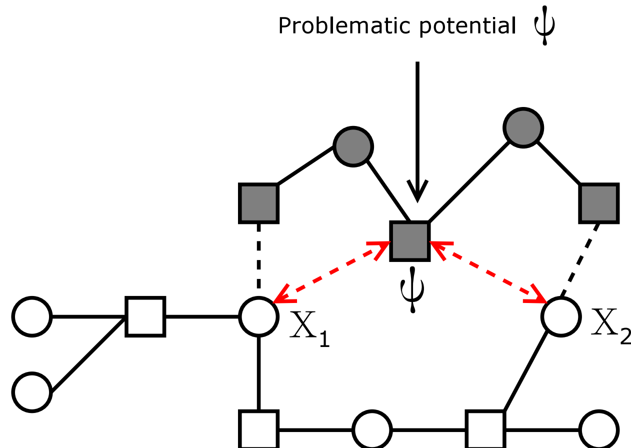


Figure 4.9: A wrong MCMC tree partition for a Factor Graph. While the given partition in two trees (white and shaded) would have been legitimate for a pairwise graph, it is not valid for a factor graph. The problem arises when sampling the white tree. The two red-dashed edges show that while conditioning out the shaded RVs, one of the potentials ψ belonging to the shaded tree *depends on two variables* $\mathbf{x}_1$ and $\mathbf{x}_2$ in the white tree. Thus, ψ can not be considered as a potential depending on a single variable while we sample the white tree. We have a loop, which makes the use of algorithm 4.5 impossible.

$$\forall (i,j) \in \{1,\dots,l\}^2, \mathcal{V}_i \cap \mathcal{V}_j = \emptyset \tag{4.9}$$

$$\forall k \in \{1,\dots,l\}, \forall (i,j) \in \mathcal{V}_k^2, (i,j) \in \mathcal{E} \Rightarrow (i,j) \in \mathcal{E}_k \tag{4.10}$$

$$\forall k \in \{1,\dots,l\}, \forall i \in P_k, \forall m \in \{1,\dots,l\} \setminus k, \left|(i,j) \in \mathcal{E} \mid j \in V_m\right| \leq 1 \tag{4.12}$$

Figure 4.10 gives a corrected tree partition for the factor graph of figure 4.9. In practice, the property expressed by Equation (4.12) makes it much harder to find factor graph MCMC tree partitions.

4.4 MCMC Tree Sampling Implementation

In this last section, we provide the full pseudo-code for the MCMC tree sampler. The overall structure is the one of a Gibbs Sampler. Variables are replaced with trees. To

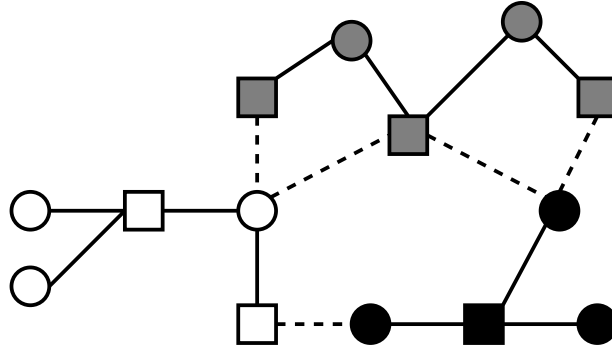


Figure 4.10: A legitimate MCMC tree partition, with the addition of a third tree (in black). Now the potential linked to four variables has 2 variables in its own tree, and only one variable in each of the other trees. Property (4.12) is respected.

sample from these trees, we will use the methods described in Figures 4.2 and 4.5.

We have not discussed so far the output of the results. If we interpreted MCMC tree sampling merely as an improvement over Gibbs sampling, we could just *count* the samples obtained for each variable. We would use the Monte Carlo histogram as the approximation to the marginals. Per sample, MCMC tree sampling would be more efficient that Gibbs sampling, since we sample from a much larger state space with tree sampling. It turns out that with MCMC tree sampling, we can adopt *Rao-Blackwellized* estimates instead of the simpler Monte Carlo histogram that corresponds to counting [6].

4.4.1 Rao-Blackwellization

For Rao-Blackwellization, we add directly the conditional probabilities we computed for variable $\mathbf{x}_i$ at each step t. These probabilities are conditioned on the RVs not on the tree containing $\mathbf{x}_i$, denoted as $\mathcal{T}(i)$. We denote this set of RVs $\mathbf{x}_\Delta$. These values can be computed by a simple application of Belief Propagation (see Section 2.2) to the tree $\mathcal{T}(i)$, conditionned on all the other trees Δ.

For T total samples, the expression of both estimates is given by Equations (4.13)

and (4.14).

Monte Carlo estimates:
$$\widetilde{p}(\mathbf{x}_i = x_i) = \frac{1}{T+1} \sum_{t=0}^{T} \delta(\mathbf{x}_i^{(t)} = x_i) \qquad (4.13)$$

Rao-Blackwellized estimates:
$$\widetilde{p}(\mathbf{x}_i = x_i) = \frac{1}{T+1} \sum_{t=0}^{T} p(\mathbf{x}_i = x_i \mid \mathbf{x}_{\Delta}^{(t)}) \qquad (4.14)$$

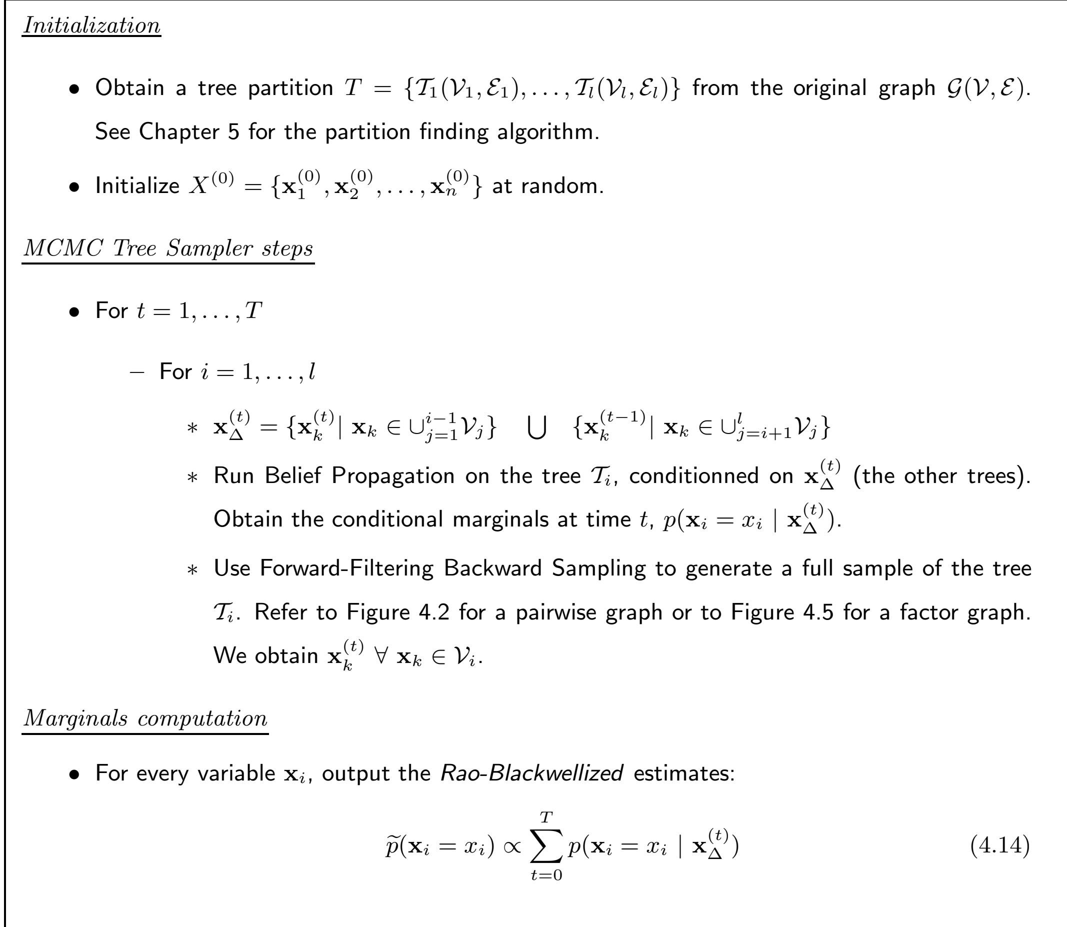
It can be proven [6] that Rao-Blackwellized estimates have lower variance than Monte Carlo estimates. The computation of RB estimates is more expensive, as it requires us to compute the exact conditional marginals via BP at each step. However, since we already have to do half the work of Belief Propagation to sample from a tree, Rao-Blackwellized estimates only cost the other half of BP's computational work. In practice, the reduction in variance is worth this extra cost.

4.4.2 Pseudo-code and remarks

Figure 4.11 contains the pseudo-code for the MCMC tree sampler. Note the similarity of this code with the Gibbs Sampler of Figure 3.1.

The MCMC tree sampler is a general Monte Carlo algorithm that builds on the Gibbs sampler and improves it through intelligent blocking. We presented this algorithm in its "one-partition" version, but we can also use MCMC tree sampling with several tree partitions. At each sampling step, we choose a different tree partition rather than a static one. That is, we use a mixture of MCMC kernels [20].

This variation on the basic MCMC tree sampler once more emphasizes the need for generating many MCMC tree partitions. Obtaining a correct MCMC tree partition is a hard problem. We attack this task in the next chapter.



Initialization

- Obtain a tree partition $T = \{\mathcal{T}_1(\mathcal{V}_1, \mathcal{E}_1), \ldots, \mathcal{T}_l(\mathcal{V}_l, \mathcal{E}_l)\}$ from the original graph $\mathcal{G}(\mathcal{V}, \mathcal{E})$. See Chapter 5 for the partition finding algorithm.
- Initialize $X^{(0)} = \{\mathbf{x}_1^{(0)}, \mathbf{x}_2^{(0)}, \ldots, \mathbf{x}_n^{(0)}\}$ at random.

MCMC Tree Sampler steps

- For $t = 1, \ldots, T$
  - For $i = 1, \ldots, l$
    * $\mathbf{x}_\Delta^{(t)} = \{\mathbf{x}_k^{(t)} \mid \mathbf{x}_k \in \cup_{j=1}^{i-1} \mathcal{V}_j\} \quad \bigcup \quad \{\mathbf{x}_k^{(t-1)} \mid \mathbf{x}_k \in \cup_{j=i+1}^{l} \mathcal{V}_j\}$
    * Run Belief Propagation on the tree $\mathcal{T}_i$, conditionned on $\mathbf{x}_\Delta^{(t)}$ (the other trees). Obtain the conditional marginals at time t, $p(\mathbf{x}_i = x_i \mid \mathbf{x}_\Delta^{(t)})$.
    * Use Forward-Filtering Backward Sampling to generate a full sample of the tree $\mathcal{T}_i$. Refer to Figure 4.2 for a pairwise graph or to Figure 4.5 for a factor graph. We obtain $\mathbf{x}_k^{(t)} \ \forall \ \mathbf{x}_k \in \mathcal{V}_i$.

Marginals computation

- For every variable $\mathbf{x}_i$, output the *Rao-Blackwellized* estimates:

$$\widetilde{p}(\mathbf{x}_i = x_i) \propto \sum_{t=0}^{T} p(\mathbf{x}_i = x_i \mid \mathbf{x}_\Delta^{(t)}) \tag{4.14}$$

Figure 4.11: The MCMC Tree Sampler algorithm, for T steps and a graph $\mathcal{G}(\mathcal{V}, \mathcal{E})$.

Chapter 5

Under her song the saplings
grew and became fair and tail.

The Silmarillion J.R.R. TOLKIEN

Finding MCMC Tree Partitions

The MCMC tree sampling algorithm introduced in Chapter 4 represents a considerable improvement over simple Gibbs sampling. However, it requires an MCMC tree partition of the graph. This chapter provides solutions to the problem of finding this partition automatically.

5.1 Introductory remarks

5.1.1 Definition of the problem

Given any pairwise or factor graph, we want to partition it according to the definitions 4.3.1 and 4.3.2. The difficulty arises not from the mathematical definitions, but from the *optimality* of such a partition.

Finding a correct tree partition is actually trivial. It is sufficient to take every variable node in the graph and declare that this single node forms a tree. We will thus obtain as many trees as there are variables. This partition corresponds exactly to the simple Gibbs sampler, so we lose the benefits of tree sampling.

The problem is to find an MCMC tree partition that will give good sampling performance. Ideally, we would like to have a partition with as few trees as possible, and large trees. This intuition is based on the theoretical results in [6].

5.1.2 Search heuristics to minimize the number of trees in an MCMC tree partition

While it is possible to evaluate the performance of various MCMC tree partitions empirically, understanding why a given partition gives better performance is extremely

hard. We focus on designing an algorithm to minimize only one factor: *the total number of trees* in the partition.

Our partition-finding algorithms are based on search heuristics, not mathematical properties. In fact, our specific problem does not appear to have been tackled in the graph theory literature. There are many results on partitioning a graph into trees. The most famous one was obtained by Nash-Williams in [18]; more recently, see also [2]. Unfortunately, all these results are not applicable to our case, since the partitions obtained respect radically different properties.

5.2 The pairwise case

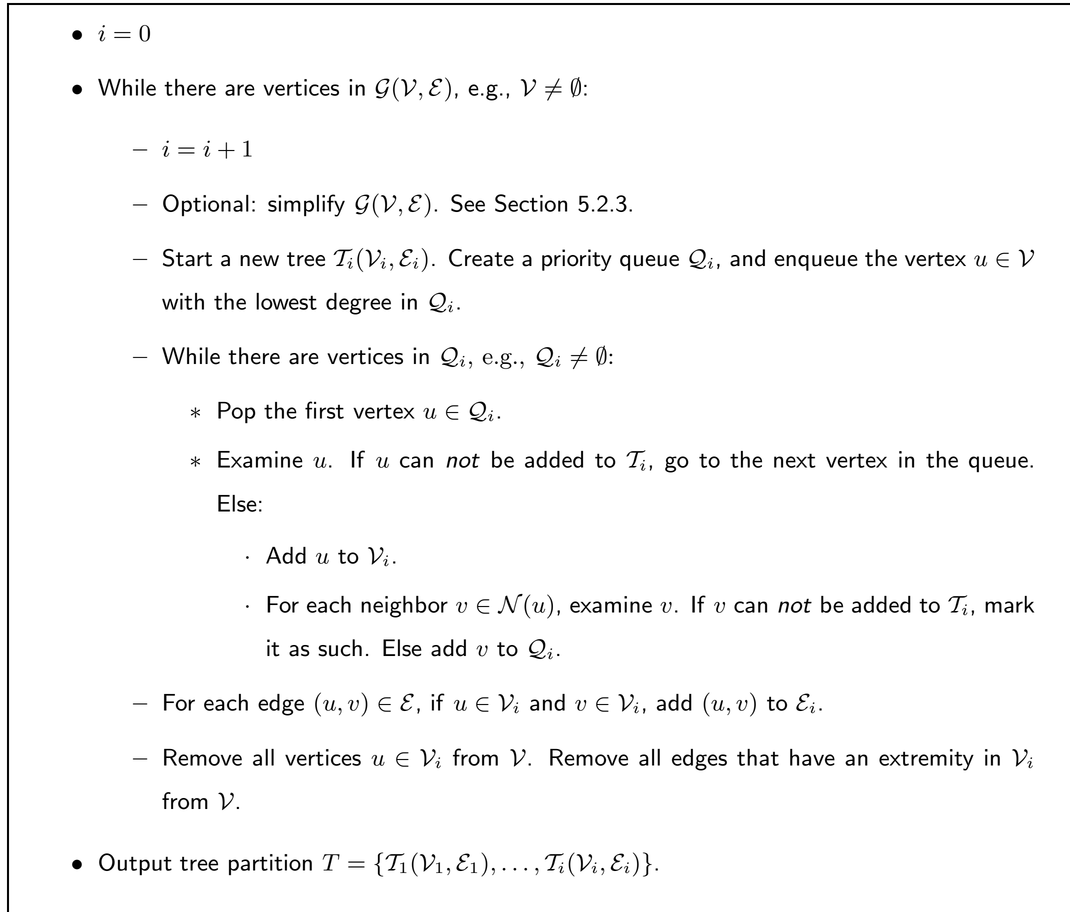
5.2.1 Essential choices

Our exploration mechanism for partitioning a graph is based on a vertex search. This means we continuously add vertices to a current tree. Once we cannot add more vertices to the current tree, we remove it from the graph and start with a new (empty) tree on the rest of the graph. In order to ensure that this process will terminate, we need to assert that the trees created will respect Definition 4.3.1 and that we can always add at least one vertex to a new tree. Adding one vertex is in fact always possible in an obvious manner, since we pointed out that we can make a partition consisting of one-variable trees. To respect Definition 4.3.1, we need to introduce an exploration mechanism.

It should be noted that this vertex search choice is not a requirement. A completely different approach could have been taken. For example, one could try to focus on *removing* some edges (perhaps every time there is a loop). The strategy of creating trees one at a time is also questionable; trying to partition a graph "globally" can make more sense. We make no claims that our approach is the best. However, it is easy to implement, and its computational complexity is low. Other approaches seemed more complex, and thus more risky.

5.2.2 General overview

While the next sections will describe in more detail our partitioning algorithm, we present an overview first in Figure 5.1.



- $i = 0$
- While there are vertices in $\mathcal{G}(\mathcal{V}, \mathcal{E})$, e.g., $\mathcal{V} \neq \emptyset$:
  - $i = i + 1$
  - Optional: simplify $\mathcal{G}(\mathcal{V}, \mathcal{E})$. See Section 5.2.3.
  - Start a new tree $\mathcal{T}_i(\mathcal{V}_i, \mathcal{E}_i)$. Create a priority queue $\mathcal{Q}_i$, and enqueue the vertex $u \in \mathcal{V}$ with the lowest degree in $\mathcal{Q}_i$.
  - While there are vertices in $\mathcal{Q}_i$, e.g., $\mathcal{Q}_i \neq \emptyset$:
    - Pop the first vertex $u \in \mathcal{Q}_i$.
    - Examine u. If u can *not* be added to $\mathcal{T}_i$, go to the next vertex in the queue. Else:
      - Add u to $\mathcal{V}_i$.
      - For each neighbor $v \in \mathcal{N}(u)$, examine v. If v can *not* be added to $\mathcal{T}_i$, mark it as such. Else add v to $\mathcal{Q}_i$.
  - For each edge $(u, v) \in \mathcal{E}$, if $u \in \mathcal{V}_i$ and $v \in \mathcal{V}_i$, add (u, v) to $\mathcal{E}_i$.
  - Remove all vertices $u \in \mathcal{V}_i$ from $\mathcal{V}$. Remove all edges that have an extremity in $\mathcal{V}_i$ from $\mathcal{V}$.
- Output tree partition $T = \{\mathcal{T}_1(\mathcal{V}_1, \mathcal{E}_1), \ldots, \mathcal{T}_i(\mathcal{V}_i, \mathcal{E}_i)\}$.

Figure 5.1: Structure of the partitioning algorithm for a pairwise graph $\mathcal{G}(\mathcal{V}, \mathcal{E})$.

We have been deliberately vague in this pseudo-code. We have not revealed how we examine the vertices, and thus how we decide if they can, or cannot, be added to the current tree. The exploration algorithm works by moving from vertex to vertex, adding them (or not) to a current tree. We move from a vertex to another only along the edges of the graph, because we only enqueue vertices that are neighbors of the current one.

Visually, this algorithm is simple to imagine. Starting from the vertex of lowest degree in the original graph, it progressively "builds" a tree by exploring around this first vertex, ensuring that the tree will respect definition 4.3.1. When the tree can not grow larger, we entirely remove it from the graph and start again the process with the remaining graph.

5.2.3 Simplifying the initial graph

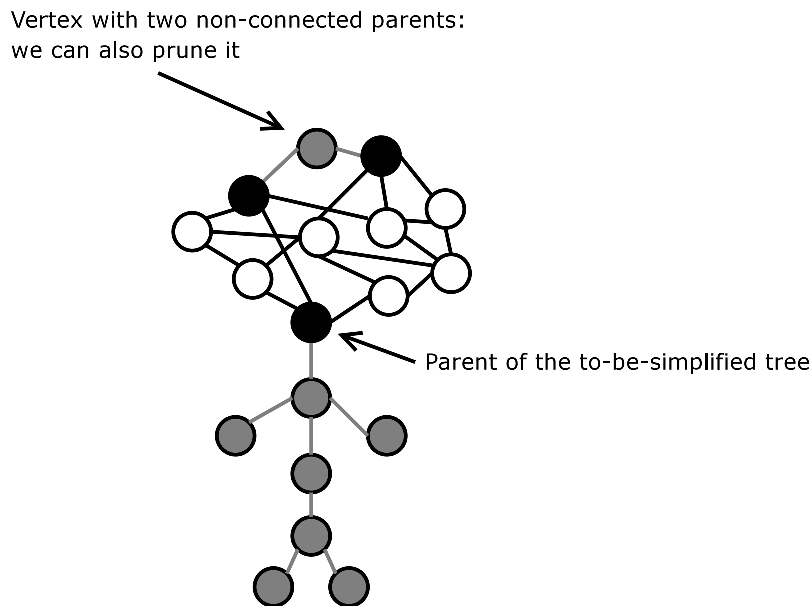


Figure 5.2: Simplification of a graph before partitioning occurs. The vertices that will be pruned appear as shaded nodes. The gray edges will be removed too.

In pairwise graphs, we can optionally perform some simplifications first. These simplifications rely on the fact that trees "attached" to the original graph can automatically be included in the partition tree containing their parent. They don't need to be explicitly visited by the exploration mechanism.

In practice, we execute the simplification code prior to each run of the exploration algorithm. Vertices with a degree[1] of 1 can be recursively pruned. Vertices with a

[1]The degree of a vertex is its number of neighbors in the graph.

degree of 2 can also be pruned if their neighbors are not themselves linked. Figure 5.2 shows an example of a simplification.

5.2.4 Exploring and coloring the graph

We use the exploration mechanism each time we need to add a new tree to the partition (starting with an empty partition). During exploration, we label the encountered vertices with different colors. We start with all the vertices initially set to a *white* color. When we add a vertex to the current tree, we mark it as *red*. We use *gray* to indicate that a vertex is in the priority queue, and *black* to indicate the vertices that cannot be included in the current tree.[2]

At the end of an exploration phase, all vertices will be either labeled as red (inclusion in the current tree) or as black (inclusion in another future tree). We will definitely remove the red vertices from the original graph $\mathcal{G}$. The black vertices are those that remain to be partitioned.

We change the color of vertices only at two points in the algorithm of Figure 5.1. When we add a vertex to the current tree, we label it in red. When we examine each neighbor of a current vertex, we color it according to the following four rules:

Rule 6. *If the neighbor has color* **red***, do not change its color.*

Rule 7. *If the neighbor has color* **white***, change it to* **gray***.*

Rule 8. *If the neighbor has color* **gray***, change it to* **black***.*

Rule 9. *If the neighbor has color* **black***, do not change its color.*

If the neighbor is red, we are looking at the vertex that was encountered prior to the current one in the same exploration phase. We do not do anything in this case as this vertex is already in the current tree, and should not be further modified.

[2]In the figures of this thesis, we used a light gray and a strong red. This allows the figures to make sense if the thesis is printed in black and white: red appears as a dark gray while gray appears as a light gray.

Encountering a white neighbor means that we have reached a new vertex never seen before. It should be marked as gray, because we will include it in the priority queue. It becomes a candidate for the next vertex to be chosen by the exploration method.

A gray neighbor is a vertex that we could reach from *another* of the vertices already in our current tree. This vertex, if we added it to the tree being built, would create a loop. It should thus *not* be added. We mark it as black to remember that it will not be part of the current tree.

When encountering a black vertex, we obviously leave it black.

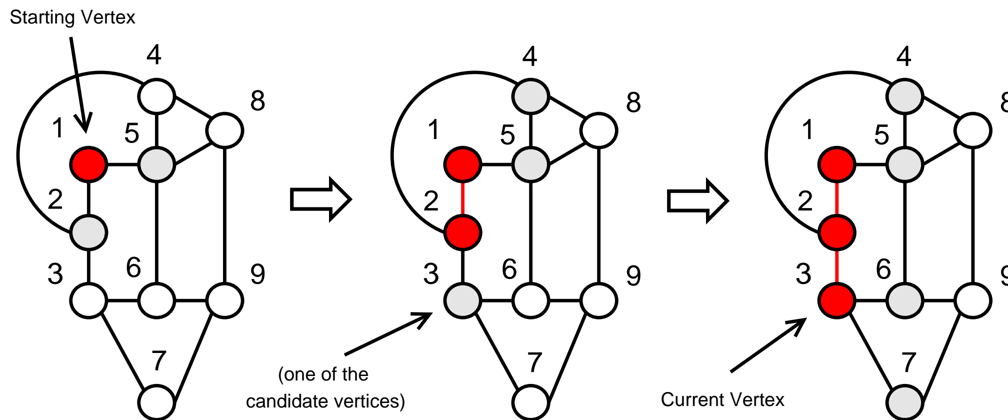


Figure 5.3: The process of coloring a graph. We start with vertex 1. It has neighbors 2 and 5, which we select as next candidates and label in gray. We then move to vertex 2, and then to vertex 3, adding them to the tree and coloring them in red. At the end of the exploration of the third vertex, 4 vertices are in the priority queue: 4, 5, 6, and 7.

When we have finished exploring and labeling all the neighboring vertices, we enqueue all the vertices marked in gray. The priority queue $\mathcal{Q}_i$ corresponds to the set of vertices that are the "next candidates" for exploration. We use the term "priority queue" because there is an ordering on these vertices, as we will explain on the next section.

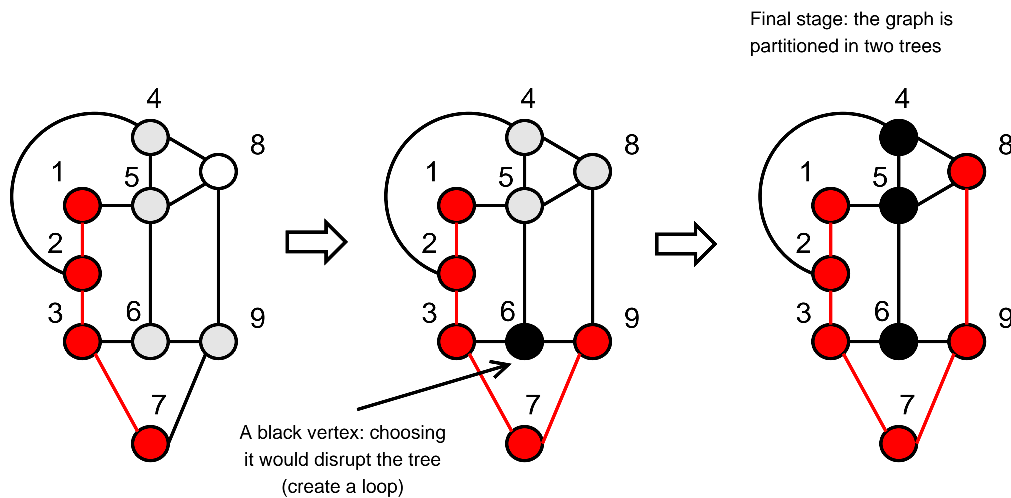


Figure 5.4: Second part of the coloring process. Jumping from vertex 3 to vertex 7, we encounter our first black vertex (6) when we visit vertex 9. Vertex 6 was already a neighbor of vertex 3. It is colored in gray. Since it is also a neighbor of 9, it becomes a black vertex. The last vertex to be included in the graph is vertex 8. Note that adding vertex 8 terminates the process of labeling all the vertices as either red or black.

Each time we retrieve a vertex u from the queue $\mathcal{Q}_i$, we test its color first. If it is not black, we do add it to the current tree and thus change its color to red. If it is black, we just skip this vertex as written on 5.1). This is important, because while its initial color was gray when we originally added it to the queue, its color may have changed to black by the time we retrieve it from the queue.

Figures 5.3 and 5.4 present a full example of graph coloring. Starting a new exploration with the 3 remaining black vertices (4, 5, and 6) will just lead to a new tree containing these 3 vertices. So we finally obtain a partition consisting of two trees.

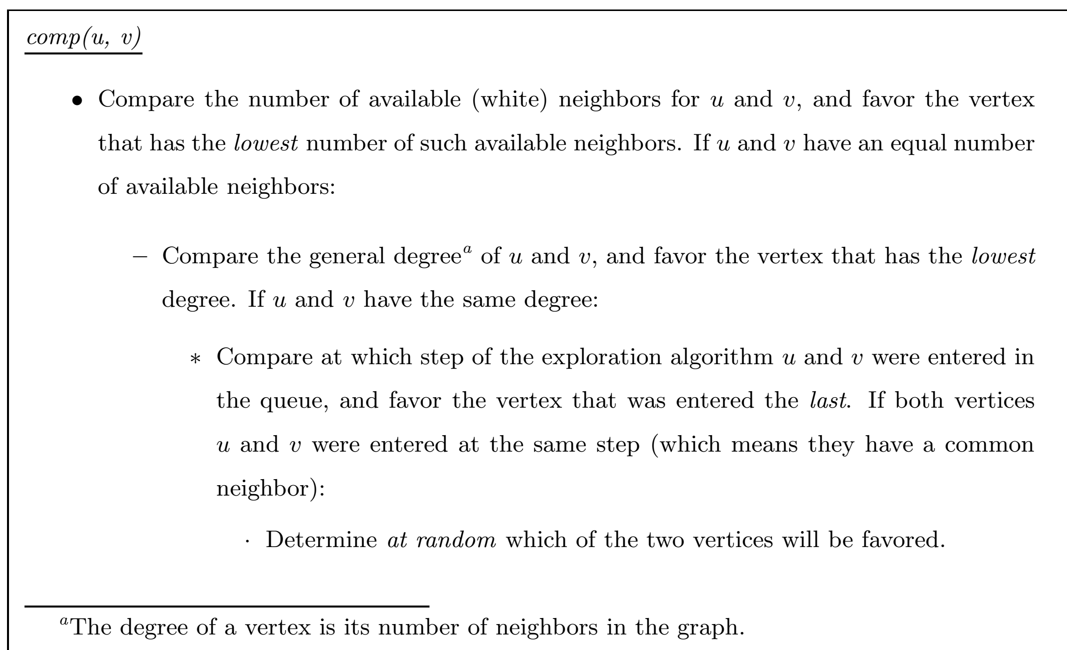
5.2.5 Choosing an ordering on the priority queue

This section addresses the important issue of the ordering on the priority queue $\mathcal{Q}_i$. The exploration algorithm should favor some vertices over others when it selects its next vertex. All gray vertices should not be considered equal, but “ordered”.

To specify an ordering on a queue, we just need to supply a comparison function

comp() between two vertices u and v. If $comp(u, v) = true$, then by convention u will be out of the queue before v.

The priority queue ordering has a dramatic effect on the partitions that will be computed. There are many orderings that could be proposed. We tried a number of different methods, evaluated their performance, and chose the best. The "best method" is however dependent of the particular structure of the graph. Our proposal is not always optimal, but pragmatic.



comp(u, v)

- Compare the number of available (white) neighbors for u and v, and favor the vertex that has the *lowest* number of such available neighbors. If u and v have an equal number of available neighbors:
  - Compare the general degree[a] of u and v, and favor the vertex that has the *lowest* degree. If u and v have the same degree:
    - Compare at which step of the exploration algorithm u and v were entered in the queue, and favor the vertex that was entered the *last*. If both vertices u and v were entered at the same step (which means they have a common neighbor):
      - Determine *at random* which of the two vertices will be favored.

[a]The degree of a vertex is its number of neighbors in the graph.

Figure 5.5: A general comparison function between two vertices u and v, used to define an ordering on the queues $\mathcal{Q}$.

Figure 5.5 describes this proposal. We define the *number of available neighbors* for a vertex u as the number of its neighbors that are not colored yet (white). When we enter a vertex into the queue, we compute this number. It can decrease during the course of the exploration algorithm, but we do not update it later, as it would be too costly for large graphs.

On our ordering, we use a random choice in last resort. The introduction of

randomness in the process is actually interesting. It means that several runs of the partitioning algorithm will produce different partitions. One could then try to run the algorithm several tries in order to get a "lucky run". If we had chosen a completely random comparison function, we would be sure to get the optimal partition by running the partitioning algorithm an infinite number of times. In practice, this is infeasible.

However, the partition of the graph can be done as an *offline process*. It may be worth to spend a lot of computational time to get a "good" partition first. Once a MCMC tree partition is found, it is valid for any probabilistic problem that has the same underlying graph. It can be reused even if the potentials and variables in the graph change.

Let us see on an example how our ordering works. In fact, the exploration process of Figures 5.3 and 5.4 would have been different if we had used our comparison function (rather than some seemingly random one). Figure 5.6 shows the partitioning process with our new ordering.

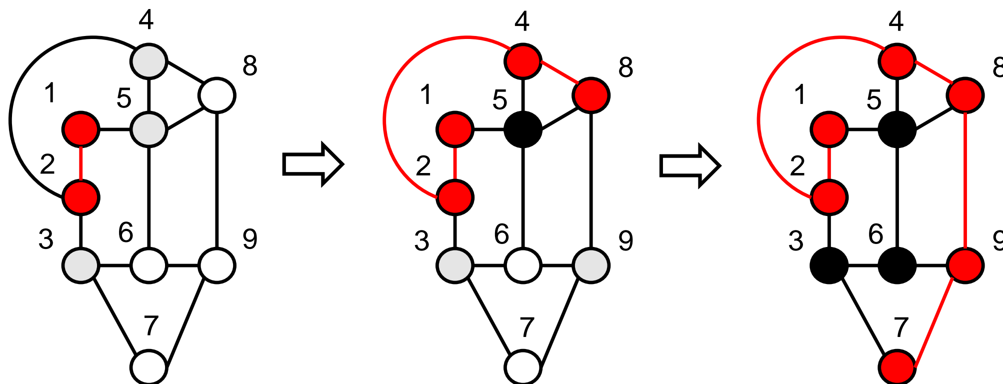


Figure 5.6: Partition of a graph, using an ordering on the queue.

We still start at vertex 1, because it has the lowest degree in the graph. Between its two neighbors 5 and 2, we favor 2 because it only has two available neighbors (3 and 4), whereas 5 has three (4, 6, 8). We thus move to 2 which neighbors are 3 and 4. We choose 4 because it has only one available vertex (8, since 5 is gray and 2 is red). 5 becomes black and our next choice can only be 8 that has only one available

neighbor.

At this point vertices 3 and 9 are in the queue. They both have the same number of available neighbors (two) and overall degree (three), so we use the third rule in our comparison function. We favor 9 because we encountered it more recently than 3. We can then choose between 3, 6 and 7. They all have no available neighbors, but we select 7 because its overall degree is lower. We thus obtain the tree (1-2-4-8-9-7) and (3-6-5). This is a different partition than the one obtained in figure 5.4.

5.2.6 A useful improvement

A traditional technique while solving constraint based problems is called *backtracking*. It involves "moving back" in the algorithm. If at some point we realize our choices were not optimal, we move back to the wrong choice and correct it. Backtracking is obviously costly in terms of computational power. We did not include a fully developed backtracking system in our algorithm. However, there is a simple improvement that can be implemented with a backtracking size of one. This allows to "forget" the current vertex and choose another one if needed.

With one-step backtracking, we can avoid the kind of situation described in Figure 5.7. Let us pretend that we have explored the graph, starting from vertex 1. We explored[3] 2, 3, 4, 5 and finally 6. At this point, vertices 7 and 9 are labeled in black, and we cannot choose them. Vertex 8 is in gray and is the next candidate for exploration.

If we choose vertex 8, it gets added to the current tree, and this leads to a sub optimal result later. This is because adding 8 to the current tree, while legitimate, will "cut off" the only path between the remaining vertices 7 and 9. If vertex 8 was available for the next tree, the next tree would obviously contain 7, 8 and 9. The tree partition would thus have 2 trees and be optimal. But if vertex 8 is added to the first tree, 7 and 9 will constitute one node trees later, and the partition will have three trees: (1-2-3-4-5-6-8), (7), and (9). Our improvement consists in making the algorithm aware

[3]This example does not use any ordering on the queue. The exploration sequence shown here is however likely to appear in practice.

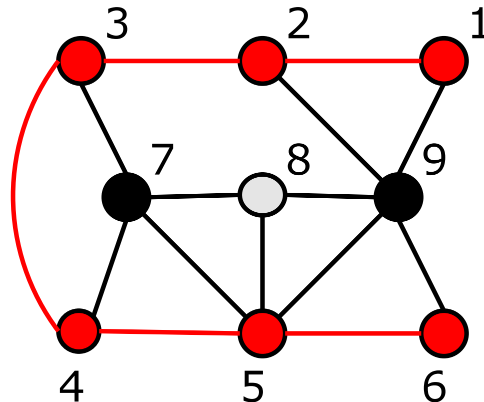


Figure 5.7: Exploration of a graph resulting in a sub optimal partition. Normally, the algorithm would now explore vertex 8, adding it to the current tree. This would result in a partition with three trees. We can obtain the optimal partition if we do not explore 8 and mark it as black.

that choosing 8 is sub optimal. It is better to stop at the step described in Figure 5.7.

The implementation of this improvement is easy. We keep a record of available edges[4] for every node. This number is initially equal to the number of neighbors of the node, and decreases by one each time one of the neighbor is encountered. We then write the following rule:

Rule 10. *Whenever the number of available edges for a vertex reaches 0, we backtrack.*

In our example, there are initially four available neighbors for vertex 7. When we explore 3, 4, and 5 this number drops to one. When we explore 8, that number drops to zero. According to Rule 10, we backtrack. We do not label 8 in red, and instead mark it in black. When a vertex is the last "escape route" for another vertex already labeled in black, we can *always* be sure that it is better to backtrack. If we added the vertex to the current tree, we would need another tree for the trapped vertex. If we leave the vertex for later, we may avoid the need for another tree, or we may not. When backtracking, the situation is not automatically better. But it cannot be worse.

[4]The number of available edges is the same as the number of available neighbors. This number is also used for the ordering of the queue in Section 5.2.5.

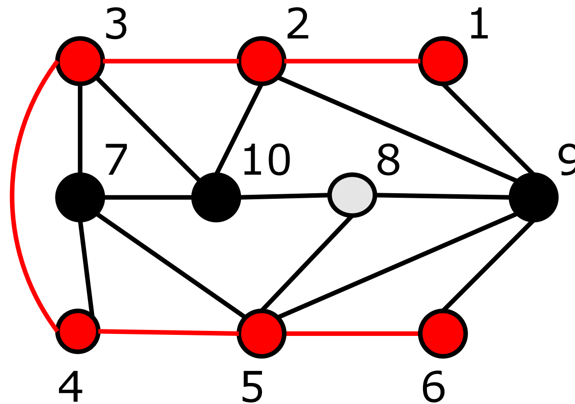


Figure 5.8: Another example of "trapped vertices". 7 and 10 can only be linked to 9 via 8. As in Figure 5.7, we should not add vertex 8 to the current tree. With a generalized propagation of information about available neighbors, 7 will contact 10 when it understands that 10 is its only remaining neighbor. It will happen when we will have visited vertices 3, 4 and 5.

Improvements with *trapped vertices*, or *isolated vertices*[5], can go even further than the simple example of Figure 5.7. Figure 5.8 shows a slightly modified graph, where the only change is the addition of vertex 10 between vertices 7 and 8. On this graph, we would like the same backtracking to happen. Adding vertex 8 to the current tree is worse than leaving it for the future. With the implementation we just described, vertices 7 and 10 would never reach zero available neighbors, as they are linked together. Since we never explore 7 or 10, both of these vertices keep believing that they have an escape route available. No backtracking takes place.

The solution is to use a propagation system. Whenever a vertex has only *one* available neighbor and is black, we propagate this information to the only remaining available neighbor. One vertex u essentially says to another v that its only escape path is through v. The vertex v no longer counts u as an available neighbor. This propagation can go on as many times as needed. If at the time v receives this information, it

[5]These terms refer to black vertices that have only one possible escape path to link them to the rest of the graph. Vertices become trapped during the exploration of the graph.

understands that its number of available neighbors has dropped to a single neighbor w, it can contact w, and so on.

In our example, vertex 10 will have a number of available neighbors equal to zero when vertex 8 is explored. The backtrack will happen and vertex 8 won't be labeled in red, but in black.

5.2.7 Full Pseudo Code for our partitioning implementation

In this section we give a more detailed pseudo code version of the algorithm (Figures 5.9 and 5.10), and we discuss its computational complexity.

In the worst case, the computational complexity of our partitioning method is $O(N^2)$ for a fully-connected graph with N vertices. We visit each vertex once, and on each vertex, we explore all its neighbors. We obtain a complexity of $O(N^2)$ for a fully-connected graph[6] . For graphs where the average number of edges per vertex is low compared to N, the computational complexity is just $O(N)$. This is often the case in practice.

For the naive Gibbs sampler, the complexity *for a single step* is the product of the number of vertices and the number of edges per vertex. The time needed to find a tree partition is thus comparable to the time needed to obtain a sample of the whole graph. Since many samples are needed to perform inference efficiently, the inference computations are responsible for almost all the computational time. We can safely ignore the cost of finding partitions when using tree sampling.

5.3 The general case

We will now discuss our partition finding algorithm for factor graphs. Good tree partitions on such graphs are harder to obtain. Whereas in the pairwise case large trees can be found by our automatic algorithm, usually trees will be much smaller for factor graphs.

[6]This result is not immediate, since the number of neighbors per edge decreases with each exploration phase. Some work is required to prove that the complexity is actually $O(N^2)$.

- $i = 0$
- While there are vertices in $\mathcal{G}(\mathcal{V}, \mathcal{E})$, e.g., $\mathcal{V} \neq \emptyset$:
  - $i = i + 1$
  - Optionally, simplify $\mathcal{G}(\mathcal{V}, \mathcal{E})$ according to the rules in Figure 5.10. For each vertex in $\mathcal{V}$, initialize the variable representing the available number of neighbors to the actual number of neighbors. Label each vertex in $\mathcal{V}$ in white.
  - Start a new tree $\mathcal{T}_i(\mathcal{V}_i, \mathcal{E}_i)$. Create a queue $\mathcal{Q}_i$, and enqueue the vertex $u \in \mathcal{V}$ with the lowest degree in $\mathcal{Q}_i$.
  - While there are vertices in $\mathcal{Q}_i$, e.g., $\mathcal{Q}_i \neq \emptyset$:
    - Pop the first vertex $u \in \mathcal{Q}_i$.
    - Look at the color of u. If u is black, go to the next vertex in the queue. Else, label u in red. Perform the following for each neighbor $v \in \mathcal{N}(u)$:
      - If v only has one remaining available neighbor (which is u), stop and *back-track* (e.g., undo all operations done in this loop) to another vertex in the queue. Mark u as black.
      - Look at the color of v and change it according to the rules in Figure 5.10.
      - Decrease the number of available neighbors in v. If that number reaches 1, v has a single available neighbor w. Decrease the number of available neighbors for w, and continue to *propagate* these changes until we encounter a vertex having more than one available neighbor.
    - Enqueue all gray vertices in $\mathcal{Q}_i$, according to the ordering described in Figure 5.5.
  - For each vertex $u \in \mathcal{V}$ that is red, add u to $\mathcal{V}_i$.
  - For each edge $(u, v) \in \mathcal{E}$, if $u \in \mathcal{V}_i$ and $v \in \mathcal{V}_i$, add (u, v) to $\mathcal{E}_i$.
  - Remove all vertices $u \in \mathcal{V}_i$ from $\mathcal{V}$. Remove all edges that have an extremity in $\mathcal{V}_i$ from $\mathcal{V}$.
- Output tree partition $T = \{\mathcal{T}_1(\mathcal{V}_1, \mathcal{E}_1), \ldots, \mathcal{T}_i(\mathcal{V}_i, \mathcal{E}_i)\}$.

Figure 5.9: Full pseudo-code of the partitioning algorithm for a pairwise graph $\mathcal{G}(\mathcal{V}, \mathcal{E})$.

Simplification Rules

- Prune out vertices with degree equal to 1. As long as the following loop removed one vertex, do:
  - For each vertex u in $\mathcal{V}$:
    - If $degree(u) = 1$, remove u and the edge linking u to its parent v from the graph. We will later add u to whatever tree v is in.
- Prune out vertices with degree equal to 2. Do the following loop once:
  - For each vertex u in $\mathcal{V}$:
    - If $degree(u) = 2$, look at its two neighbors v and w. If v and w are not linked, remove u and the edges linking u to its neighbors. Add an edge from v to w. We will later add u to whatever tree v is in.

Color Changing Rules for a vertex u

- If v has color **red**, do not change its color.
- If v has color **white**, change it to **gray**.
- If v has color **gray**, change it to **black**.
- If v has color **black**, do not change its color.

Figure 5.10: Auxiliary functions for the partitioning algorithm code.

Most of the algorithm is identical to the pairwise case. Generally, we consider all nodes to be "equal", whether they correspond to random variables or potentials. We distinguish between potentials and variables *only when needed.* The coloring part of section 5.2.4 will be used without any changes. We use the same colors as before (white, red, gray, black). The ordering on the queue can also remain the same.

5.3.1 Changes in the factor graph case

A minor modification is the absence of simplifications. Pruning the vertices of degree 2 is no longer possible with factor graphs. In our implementation, we removed *all* the simplifications, even if technically pruning the vertices of degree 1 is still feasible.

The main change is due to Rule 5. It introduces a new backtracking condition:

Rule 11. *During the exploration of a* variable node *u, we backtrack if we encounter a* gray *neighbor v.*

This vertex v corresponds to a potential. Since it is already labeled in gray, it is linked to a RV w in the current tree. If we do not backtrack, we include u in the current tree and set v to black. We then have, in the current tree, two random variables linked to the same potential v. The potential v belongs to another tree, since it has been marked as black. This is a violation of Rule 5.

Backtracking, and thus not choosing u for inclusion in the current tree, is necessary to respect Rule 5. When we backtrack due to this new condition, the gray potential stays in gray. There is no need to label it in black. Indeed, this potential can be included later by the exploration mechanism.

This new backtracking condition is almost sufficient in itself to enforce Rule 5. The only additional requirement is that we keep all potentials still linked to two black variables, at the end of each exploration phase. We won't add these potentials to future trees. But they must be present in future graphs so that Rule 11 may apply. In terms of implementation, they will obey the normal coloring rules. They cannot be entered in the priority queue for inclusion in the current tree.

5.3.2 A Factor Graph partitioning example

The following example describes a complete run of the partitioning algorithm on a factor graph.

We start at vertex 1, which has one of the lowest degree on the graph. Vertex 3 has an available number of neighbors lower than 2, so we select it next. We then have vertex 2, 7, and 8 in the queue. 7 and 8 both had one available neighbor when entered into the queue. We choose 7 since its overall degree is lower.

At this point, a backtrack occurs. Normally 8 should now be selected out of the queue. However, when examining the neighbors of vertex 8, we encounter a gray

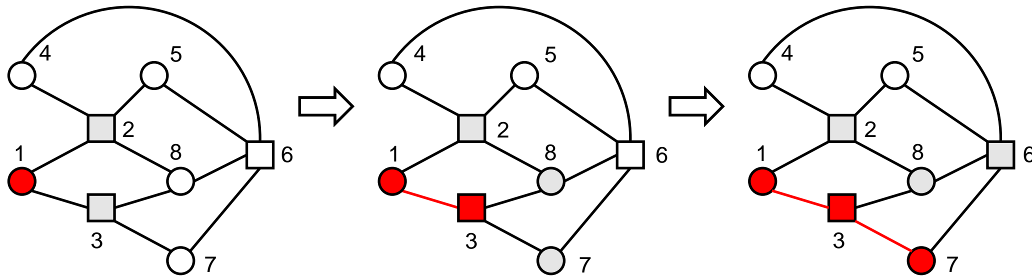


Figure 5.11: An example run of the partitioning algorithm on a Factor Graph.

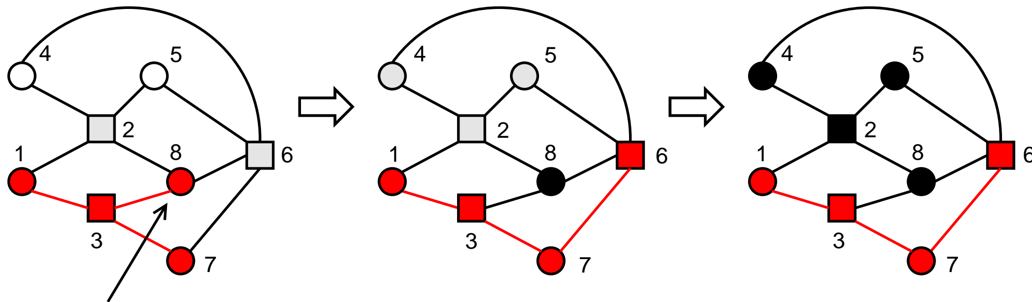


Figure 5.12: The second part of the partitioning algorithm run.

potential (actually, two: 2 and 6). According to the mechanism described in 5.3.1, we backtrack. We pop the next vertex out of the queue. Everything else is unchanged, as if we never explored 8.

The next vertex happens to be 6. After 6, we cannot add any more vertex to the current tree. We would encounter backtracks if we were to explore 4 and 5. It would be legitimate to add the potential 2 to the current tree, but this would trap vertices 4, 5, and 8. According to Section 5.2.6 and Rule 10, we do not add 2 to the first tree.

Our first tree contains 1, 3, 6, and 7. We can safely forget potential 3. Potential 6 has more than one variable in black, so we keep it for the next round. Because of this potential, variables 4, 5 and 8 must be in separate trees. In the end, we finish with a partition of four trees.

This example demonstrates that good partitions are much harder to obtain on factor graphs. If the same graph was pairwise, we could have partitioned it in only

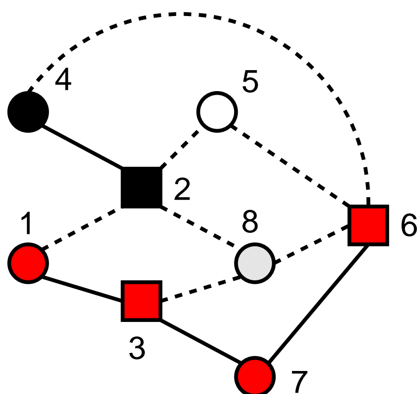


Figure 5.13: The final partition obtained, with four trees.

two trees. With FGs, we often obtain partition consisting of many small trees (see Section 6.4.3). On such partitions, tree sampling amounts to Gibbs sampling.

Chapter 6

And there arose a multitude of
growing things great and small.

The Silmarillion J.R.R. Tolkien

Experimental Results

This chapter regroups all our experimental results: a comparison of inference algorithms for pairwise graphs, for factor graphs, and the performance of our partition finding algorithm.

6.1 Experimental Setup

We used the distance between computed and true marginals as an error benchmark for our inference comparisons. If we write $p_C(\mathbf{x}_i = x_j)$ and $p_T(\mathbf{x}_i = x_j)$ respectively the computed and true marginals of $\mathbf{x}_i$, the following equation gives the error E:

$$E = \sum_{\mathbf{x}_i} \sum_{x_j} \left(p_C(\mathbf{x}_i = x_j) - p_T(\mathbf{x}_i = x_j)\right)^2 \tag{6.1}$$

For the same experiment, we used several runs in order to assess the variance of the inference algorithms. The same model is used throughout the experiment, but the parameters change and are generated for each run. The results (errors) are plotted as box plots against adjusted computational time. Adjusted computational time only means actual time, which allows for a fair comparison. We let the algorithms run for the same number of seconds before recording their errors E via Equation (6.1).

We present comparison plots in pair. On the same page, the bottom plot is a zoom of the top plot. This is necessary, since some errors bars are often invisible (too small) in the unzoomed plot.

6.2 Inference on Pairwise Graphs

We embedded a classical Potts model on our experimental pairwise graphs. Each RV could take three values. We chose uniform local potentials ϕ, so they did not matter. Between RVs $\mathbf{x}_i$ and $\mathbf{x}_j$ we used the following interaction potential $\psi(\mathbf{x}_i, \mathbf{x}_j)$:

$$\psi(\mathbf{x}_i = i, \mathbf{x}_j = j) = e^{\frac{1}{T} M_{i,j}} \tag{6.2}$$

where $M_{i,j}$ is a three-by-three diagonal matrix. The parameters $M_{i,i}$ were drawn from a standard Gaussian distribution[1] with mean 0 and variance 1. We used the same matrix M for each interaction potential.

Each RV in the graph had a probability $P_{obs} = 0.2$ to be observed. Between an observed RV $\overline{\mathbf{x}_i}$ and a non observed $\mathbf{x}_j$, there was a different interaction potential $\psi_{obs}(\overline{\mathbf{x}_i}, \mathbf{x}_j)$.

$$\psi(\overline{\mathbf{x}_i} = i, \mathbf{x}_j = j) = e^{\frac{1}{T} N_{i,j}} \tag{6.3}$$

$N_{i,j}$ is different from $M_{i,j}$, but is generated in the same way. T represents the temperature and was set to 0.5 for all our experiments. 10 runs were made for each experiment.

We computed the reference marginals with one long run of the Gibbs sampler, and one long run of the tree sampler. If both runs converged to the same marginals (e.g., the error E between the marginals was very small), then these marginals were used as "true marginals". Else, this particular run was discarded along with the generated parameters.

We used three different colors[2] and outlier symbols in the plots: tree sampling appears in black with '+' outliers, Gibbs sampling in red ('*') and LBP in green ('x').

6.2.1 Fully Connected Graph

Figure 6.1 presents the comparison results on fully connected graphs with 20 RVs. LBP often fails to converge. Tree sampling, despite the fact that a fully connected

[1] $M_{i,i}$ were allowed to be negative.

[2] If this thesis is printed in grayscale, tree sampling is black, Gibbs sampling is gray, and LBP is very light gray. This thesis should be printed in color, as it is much easier to distinguish the plots.

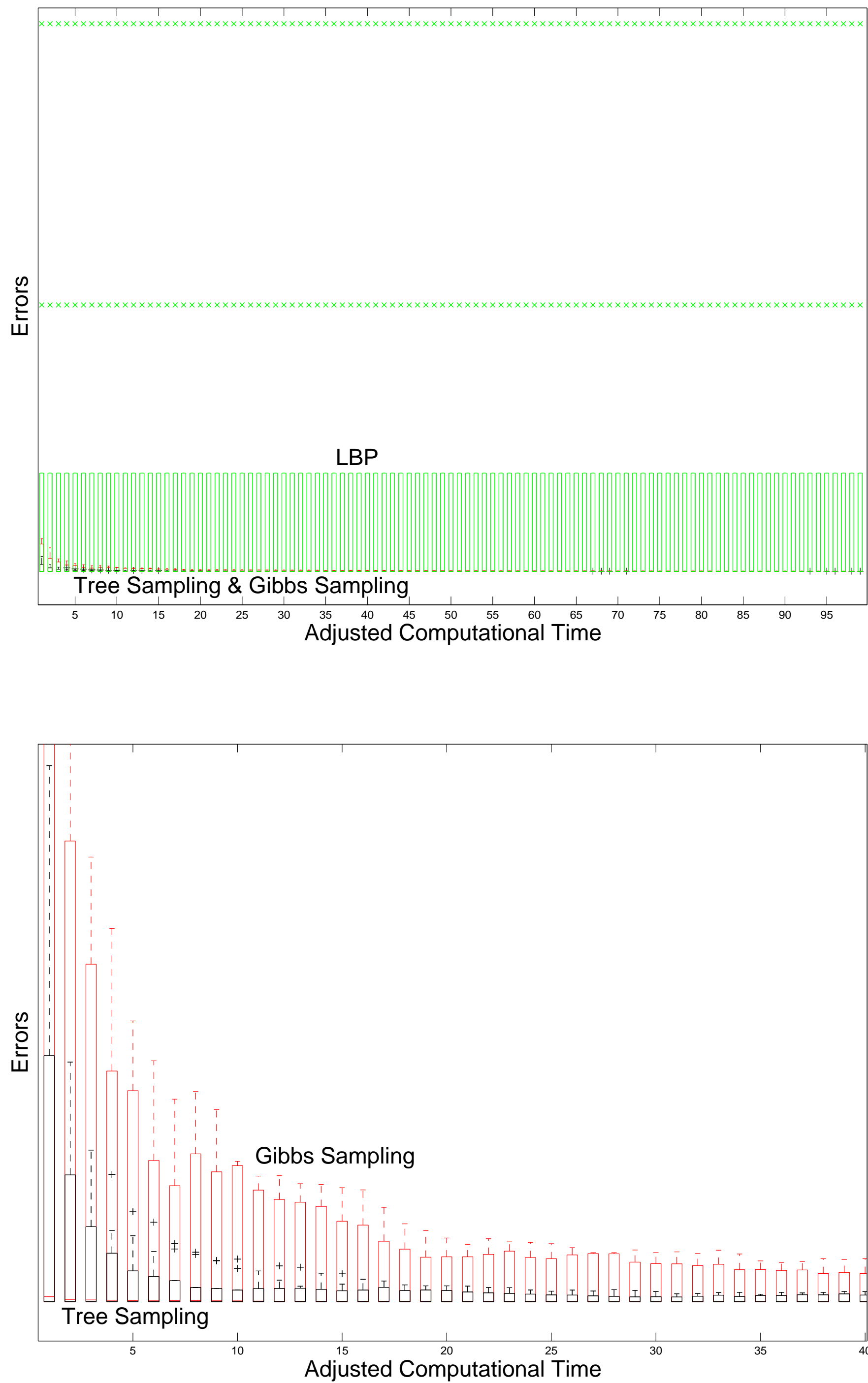


Figure 6.1: Performance comparison on a fully connected 20 nodes graph. The top graph shows that LBP's performance is very poor. These green lines of dots correspond to the errors obtained by various LBP runs. These errors are constant, since LBP converge almost instantly, and high. The bottom plot demonstrates the superiority of tree sampling over Gibbs sampling.

graph forces it to create trees with only two nodes, is the best inference algorithm here.

6.2.2 Square Lattice MRF

On a 25 by 25 MRF, tree sampling is again the best method. LBP fails to converge on one of the 10 runs. The bottom plots of Figure 6.2 show that the tree sampler has a slight performance advantage over Gibbs, but has an outlier run.

6.2.3 Random Graph

We generated a 1000 nodes random graph with a density of 0.01. The density corresponds to the probability that any two given RVs are linked together.

Gibbs sampling completely diverged in one of the ten runs (not shown on Figure 6.3), and otherwise obtains correct results, but takes longer than tree sampling. On random graphs, LBP tends to perform well. On our experiments, we found this to be true when LBP converges. But some random graphs still cause LBP to misconverge. On the graph of Figure 6.3, LBP slightly diverged on two runs. While tree sampling also diverged by a moderate amount on one of the runs (and slightly on another one), it is generally more reliable than LBP on random graphs[3].

6.2.4 Remarks

On pairwise graphs, tree sampling performs remarkably well. We found Gibbs sampling to be an efficient algorithm too. The true benefits of tree sampling, as compared to simple Gibbs sampling, are probably more visible with much larger graphs.

However, conducting experiments with large graphs is challenging, as finding ground truth is impossible. On a 1200 nodes random graph, the long Gibbs runs used for computing the true marginals diverged by a significant amount on some cases. Gibbs sampling then gives different results than tree sampling (and LBP does not converge at all).

[3] Several other random graphs experiments, not shown, support this conclusion.

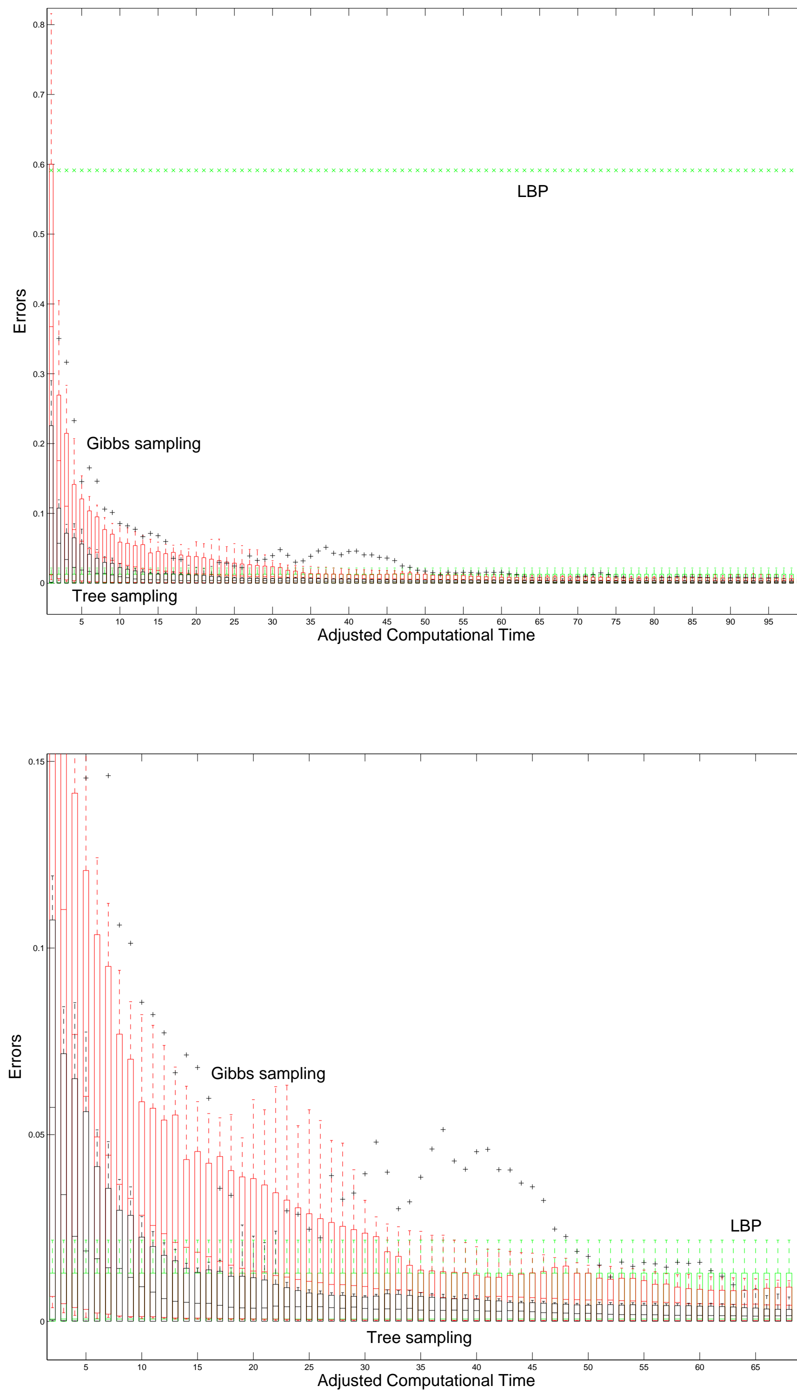
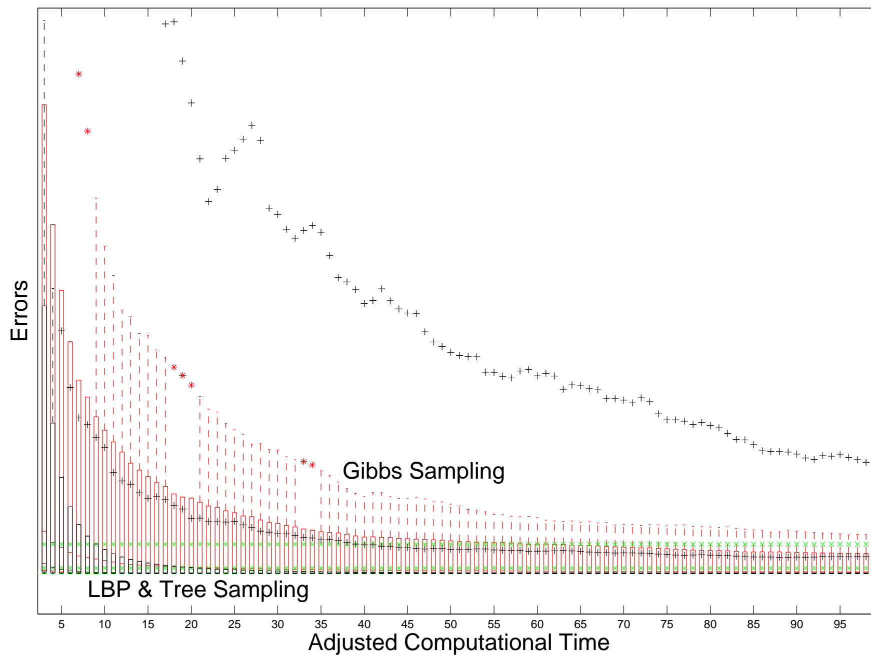


Figure 6.2: Performance comparison on a 25 by 25 square-lattice MRF. LBP obtains better results but the top graph still shows the presence of a very bad LBP run (high error). On the bottom graph, it can be seen that tree sampling has a slight advantage over Gibbs sampling. However, one of the tree sampling runs is an outlier.



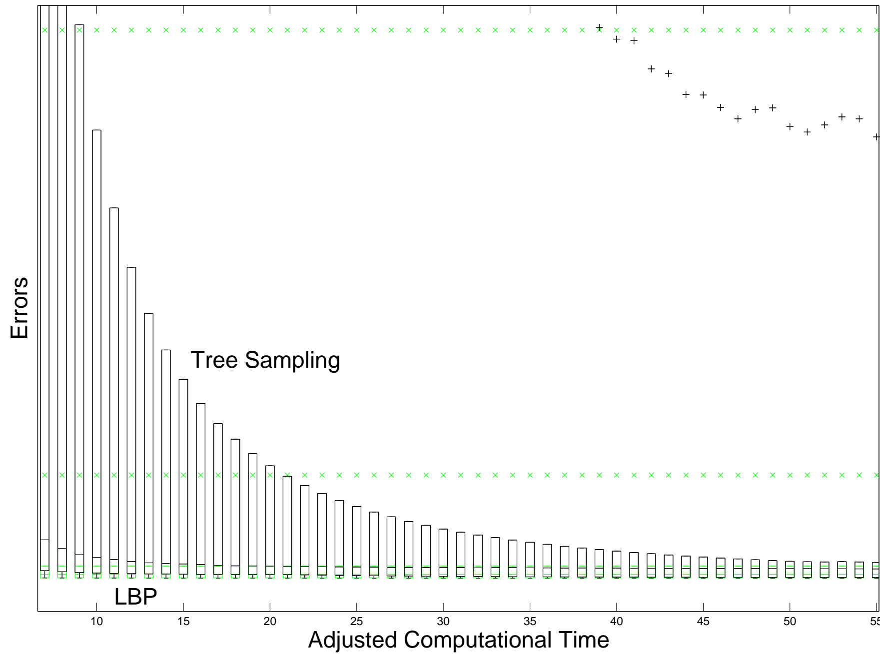


Figure 6.3: Performance comparison on a 1000 nodes random graph. Gibbs sampling has a poor performance here (it even totally diverged on one run, not shown in the plots). LBP has two outlier runs. Tree sampling is efficient, except on one run when it takes longer to converge.

6.3 Inference on Factor Graphs

We ran experiments on QMR (Quick Medical Reference) factor graphs. Jaakkola and Jordan [10] describe QMRs in more detail, along with variational inference methods for them. The joint probability of a QMR graph $\mathcal{G}(\mathcal{V}, \mathcal{P})$ can be written in Equation (6.4):

$$P(X) = \frac{1}{Z} \prod_{i \in \mathcal{P}} \psi_i(\mathbf{x}_{C_i}) \prod_{j \in \mathcal{V}} \phi(\mathbf{x}_j) \tag{6.4}$$

where ϕ represent the priors over the binary random variables (corresponding to diseases), and ψ represent the potentials associated with positive findings[4]. An expression for ψ is:

$$\psi_i(\mathbf{x}_{C_i}) = 1 - \left((1 - q_{i0}) \prod_{j \in C_i} (1 - q_{ij})^{x_j} \right) \tag{6.5}$$

q_{i0} and the q_{ij} are the parameters of the model, and need to be chosen between 0 and 1. q_{i0} represents the *leak* probability for the finding i. This is the probability that the finding is caused by other diseases than the ones included in the model. q_{ij} is the probability than the disease j, if present, will cause a positive finding i.

For each experiment, the leak probability q_{i0} was fixed at a different value. This parameter drastically influences the performance of our inference algorithms: a low leak makes computing approximate inference harder. We set the number of RVs (diseases) to 40 and the number of potentials (findings) to 14. A Bernouilli prior was chosen for every disease, with a parameter of 0.01 for the presence of the disease.

We used 15 runs in each experiment. For each run, we chose the q_{ij} uniformly at random between 0 and 1, and generated the QMR graph randomly. The density d, representing the probability that a given RV will be linked to a given potential, was set to 0.15. This is quite low, but necessary for us to be interesting. If d is high, it is just impossible to create a worthy tree partition, and tree sampling becomes Gibbs sampling (see Section 6.4.3).

[4] We only included positive findings in our experiments. Unobserved findings marginalize out and have no effects on the posteriors of the RVs, while negative findings can be factorized into the RV priors. Only positive findings make inference hard (see [10]).

We obtained the true marginals (posteriors) of the variables via the junction tree algorithm in Kevin Murphy's BNT Toolbox [16]. The errors were computed according to Equation (6.1).

6.3.1 First QMR graph: low leak

A low leak corresponds to the hardest of our QMR experiments. Loopy belief propagation fails to converge, and cycles through false beliefs. As shown on Figure 6.4, both Gibbs sampling and tree sampling converge. Gibbs has the best performance, as it produces much more samples than tree sampling in the same amount of time.

6.3.2 Second QMR graph: medium leak

With a medium leak, all three inference algorithms converge. Figure 6.5 shows that tree sampling achieves the best performance, followed by LBP and Gibbs sampling.

6.3.3 Third QMR graph: high leak

A high leak corresponds to an easy situation. All algorithms converge (Figure 6.6). Tree sampling (thanks to Rao-Blackwellization) and LBP both outperform simple Gibbs sampling. LBP is the best algorithm here.

6.3.4 Remarks

A low leak corresponds to the most interesting case, as the graph is more constrained. This causes LBP to fail on the first experiment. Tree sampling and Gibbs sampling both converge every time. Gibbs sampling produces much more samples per second than tree sampling, and outperforms our inference algorithm on these small test graphs.

As the leak increases, LBP and tree sampling start to perform better. On the high leak case, LBP is extremely efficient. Tree sampling is the best on the medium leak case.

The QMR experiments demonstrate that tree sampling is a compromise between LBP and Gibbs sampling. Contrary to LBP, it never failed to converge. It is able

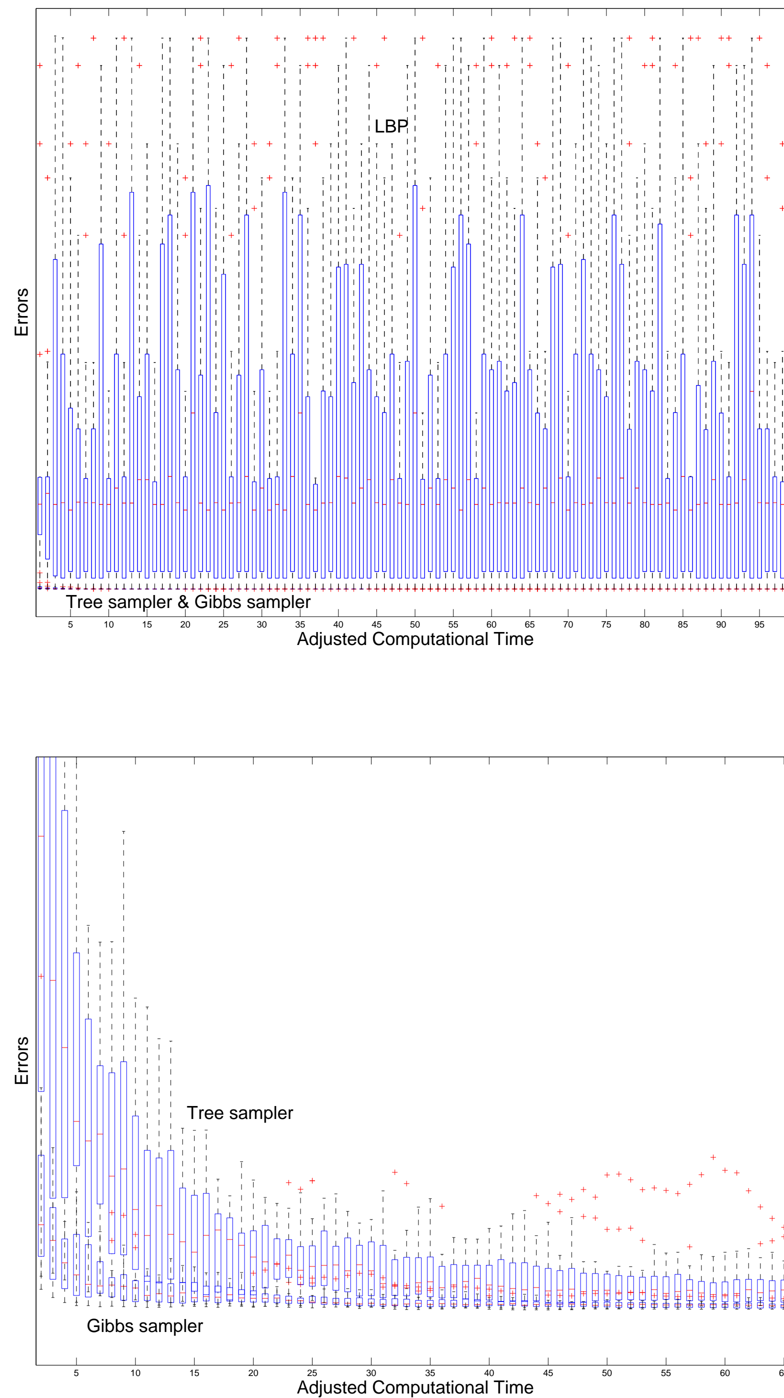


Figure 6.4: Performance comparison on a low-leak QMR network. The top plot shows LBP cycling through different false beliefs. The bottom plot shows the two other algorithms converging: Gibbs sampler converge faster.

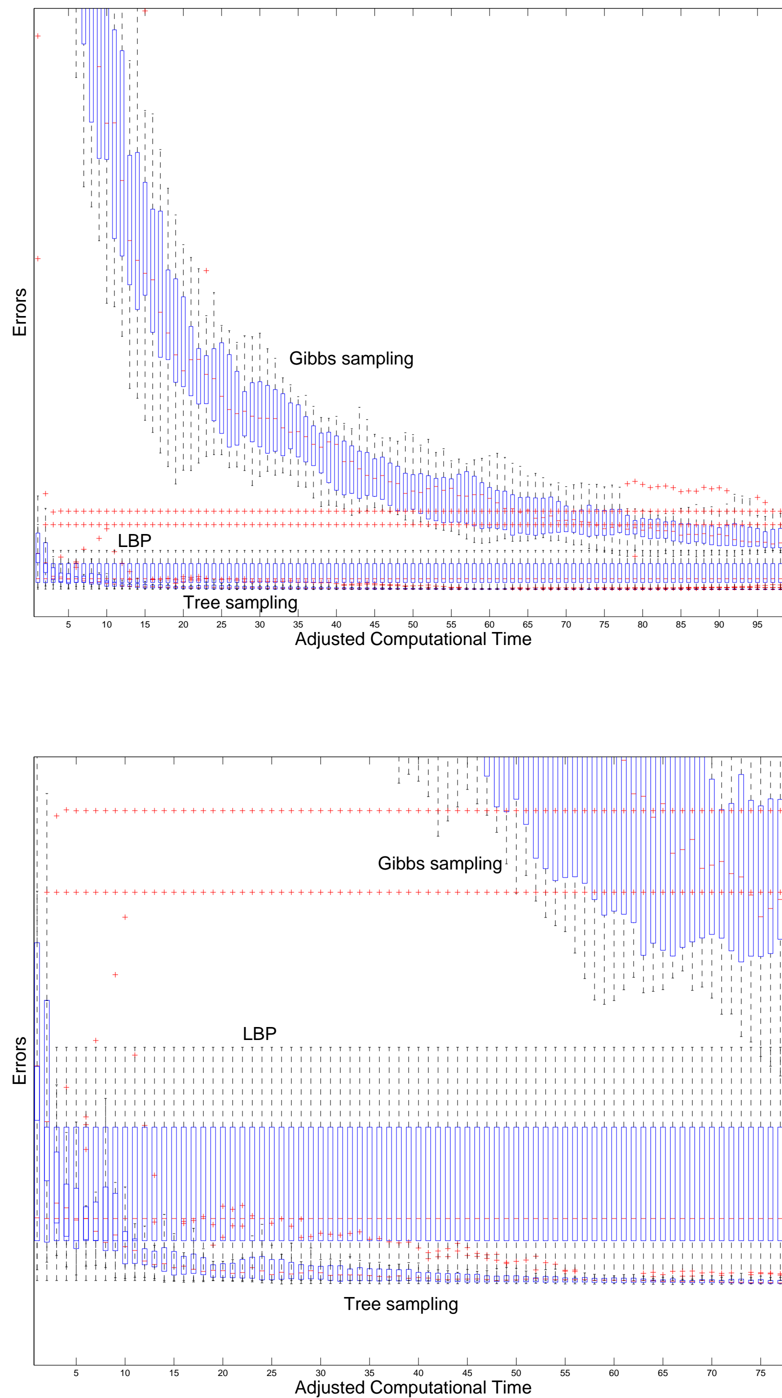


Figure 6.5: Performance comparison on a medium-leak QMR network. All algorithms obtain the correct marginals (even if LBP and Gibbs have much higher errors than tree sampling). Tree sampler is clearly the fastest and most robust method here.

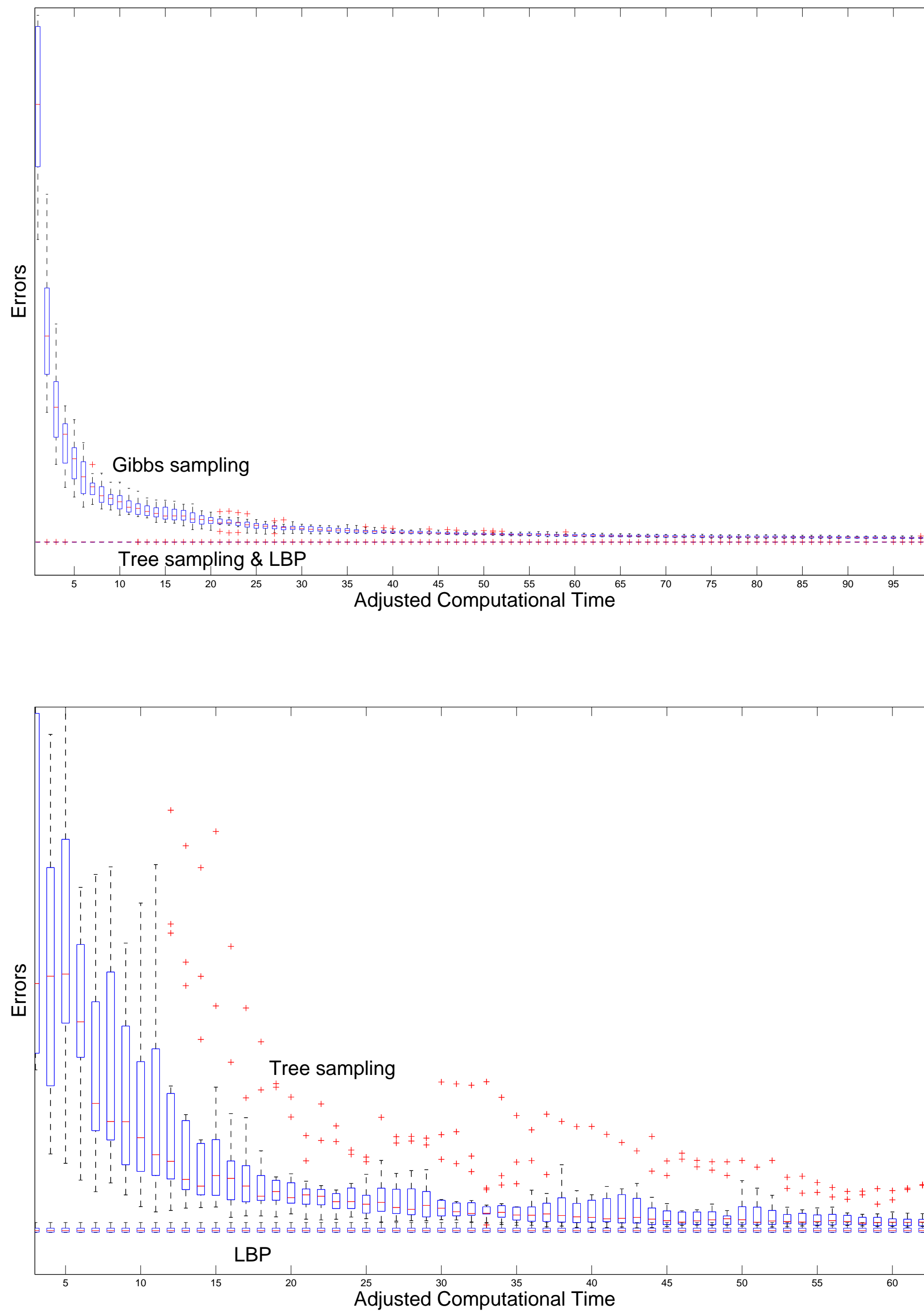


Figure 6.6: Performance comparison on a high-leak QMR graph. On this easy case, LBP outperforms the MCMC algorithms. The top plot shows that Gibbs performance is poor compared to the others. The bottom plot demonstrates the superiority of LBP over tree sampling.

to take advantage of belief propagation when it helps. On these cases, it outperforms Gibbs sampling.

6.4 Tree Partitioning Algorithm Results

All reported results count the number of trees in the resulting partition. On each experiment, we ran the partitioning algorithm 20 times. We provide the mean result (rounded), along with the best and worst run results.

6.4.1 Square Lattices MRF

Square-lattices MRFs are hard graphs to partition automatically, since they contain so many loops. We present the partitioning results for MRFs in Table 6.1.

MRF size	Queue Ordering	Mean	Best	Worst
5*5	normal	2	2	2
5*5	random	3	2	5
10*10	normal	5	3	7
10*10	random	8	5	10
20*20	normal	26	17	36
20*20	random	25	17	31
50*50	normal	148	105	241
50*50	random	152	127	166
100*100	normal	365	273	639
100*100	random	584	563	624

Table 6.1: Partitioning Algorithm applied to pairwise square-lattices MRF.

Roughly, there are twenty times less trees than the number of vertices[5]. We can see that the queue ordering does matter. Choosing the "normal one" (defined in

[5]This does *not* mean, however, that the average length of a tree is 20 vertices. On the MRF case, generally a single tree accounts for more than half of the total vertices in the graph.

Figure 5.5) has clear benefits compared to a totally random one (except, surprisingly, in the 20*20 case, but even there their performance is almost equal).

6.4.2 Pairwise Random Graphs

Random Graph size	Density	Backtrack	Mean	Best	Worst
100	0.1	yes	5	5	6
100	0.1	no	6	5	7
100	0.5	yes	14	14	15
100	0.5	no	14	14	15
1000	0.01	yes	7	6	9
1000	0.01	no	17	10	22
1000	0.25	yes	41	40	42
1000	0.25	no	41	40	42
10000	0.01	yes	22	21	24
10000	0.01	no	31	25	37

Table 6.2: Partitioning Algorithm applied to pairwise random graphs.

Our algorithm provides good results for random graphs. Even with a high density (0.25), we find a correct partition in 41 trees for a 1000-nodes graph, where each vertex was in average linked with 250 others. It is interesting to note that for random graphs, the variance across different runs is much lower than for square-lattices. While for MRFs there was a difference of 366 trees between the best and worst run for 10000 vertices, here this difference is only 3.

Table 6.2 makes a comparison between the backtrack-enabled algorithm and the simpler one. It is clear that backtracking is very useful for random graphs with low density. If the graph has a high density, it is however unable to help. These results confirm the claims of Section 5.2.6. Backtracking can only help, and never worsens the results.

6.4.3 Random Factor Graphs

We created random factor graphs by specifying the number of RVs and the number of potentials. The density represents the maximal number of RV a potential can be linked to. We chose the edges randomly.

Number of Variables	Number of Potentials	Density	Mean	Best	Worst
50	30	3	6	6	6
50	30	5	14	13	14
250	100	4	22	19	27
250	100	8	42	41	43
1000	700	4	163	150	179
1000	1500	4	139	125	150
4000	1000	5	261	261	262
4000	1000	10	1073	1060	1075
100	75	100	100	100	100
100	75	50	96	95	96
1000	100	100	865	853	874

Table 6.3: Partitioning Algorithm applied to random factor graphs.

As expected from the example of Figure 5.13, the factor graph case is harder to partition. For a given number of random variables, the pairwise case produces generally less trees.

For factor graph with low densities, the algorithm produces partitions with relatively few trees. This is encouraging. Increasing the number of links does not *always* increase the number of trees, since it also provides more "escape routes" for trapped groups of vertices. By raising the number of potentials from 700 to 1500 with 1000 RVs, Table 6.3 shows a decrease in the number of trees found.

In cases of high density, it is impossible to obtain good partitions. The partitions for the last three lines of Table 6.3 contain trees of only one RV. Tree sampling is useless in such situations.

Chapter 7

Conclusion

From all its branches there spilled a golden dew upon the barren earth.

The Silmarillion J.R.R. TOLKIEN

This thesis extended the tree sampling framework [6] to factor graphs (Chapter 4) and, with automatic partitioning algorithms (Chapter 5), to arbitrary pairwise graphs.

Tree sampling is a hybrid algorithm, using elements of Monte Carlo simulation and Belief Propagation. The experimental results of Chapter 6 showed this heritage. Tree sampling is able to benefit from BP to outperform the simple Gibbs sampler in many test cases, albeit not all. But, contrary to LBP, tree sampling never failed to converge in our experiments.

Many aspects of tree sampling remain to be investigated. First, using multiple tree partitions (a mixture of MCMC kernels) could dramatically improve the performance of tree sampling.

Adapting tree sampling to the continuous case is a hard challenge. We tried a variant of tree sampling, called Sequential Monte Carlo (SMC) tree sampling, on a MRF with a non-parametric model. Instead of using BP to sample from a tree, we used a SMC technique [8]. The mixing of MCMC and SMC did not work well, and the results were disappointing. However, tree sampling is probably adaptable to Gaussian models.

Finally, inference algorithms such as tree sampling are not yet fully understood from a theoretical point of view. On very large graphs[1], Gibbs sampling usually has a poor performance. Our hope is that tree sampling would scale much better.

[1]It is hard to conduct experiments on such graphs, as it is impossible to get ground truth.

Bibliography

[1] C. Andrieu, N. de Freitas, A. Doucet, and M. Jordan. An introduction to MCMC for machine learning. *Machine Learning*, 50:5–43, 2003.

[2] József Balogh, Martin Kochol, András Pluhár, and Xingxing Yu. Covering planar graphs with forests. *Journal of Combinatorial Theory, Series B*, 94(1):147–158, 2005.

[3] B. Bidyuk and R. Dechter. Cycle-cutset sampling for bayesian networks, 2003. Sixteenth Canadian Conf. on AI.

[4] C.K. Carter and R. Kohn. On Gibbs sampling for state space models. *Biometrika*, 81(3):541–553, 1994.

[5] G. Casella and C. P. Robert. *Monte Carlo Statistical Methods*. Springer, 1999.

[6] Nando de Freitas and Firas Hamze. From Fields to Trees. In *Proceedings of the 20th conference on Uncertainty in Artificial Intelligence*, pages 243–250. ACM International Conference Proceeding Series, Vol. 70, 2004.

[7] W. Gilks, S. Richardson, and D. Spiegelhalter. *Markov chain Monte Carlo in practice*. Chapman and Hall, 1996.

[8] S. J. Godsill, A. Doucet, and M. West. Monte Carlo Smoothing for Nonlinear Time Series. *Journal of the American Statistical Association*, 99(465):156–168, March 2004.

[9] D. Heckerman. A tutorial on learning with Bayesian networks. Technical report, Microsoft Research, Redmond, Washington, 1995. Revised June 96.

[10] Tommi Jaakkola and Michael I. Jordan. Variational probabilistic inference and the QMR-DT network. *Journal of Artificial Intelligence Research*, 10:291–322, 1999.

[11] C. Jensen, A. Kong, and U. Kjaerulff. Blocking Gibbs sampling in very large probabilistic expert systems. *International Journal of Human-Computer Studies*, 42:647–666, 1995.

[12] M. Jordan. *Probabilistic Graphical Models*. To be published.

[13] Kschischang, Frey, and Loeliger. Factor graphs and the sum-product algorithm. *IEEETIT: IEEE Transactions on Information Theory*, 47, 2001.

[14] S. Z. Li. *Markov Random Field modeling in image analysis*. Springer-Verlag, 2001.

[15] N. Metropolis and S. Ulam. The Monte Carlo method. *JASA*, 44(247):335–341, 1949.

[16] Kevin P. Murphy. The Bayes Net Toolbox for MATLAB. *Computing Science and Statistics*, 33, 2001.

[17] Kevin P. Murphy, Yair Weiss, and Michael I. Jordan. Loopy belief propagation for approximate inference: An empirical study. In *Proceedings of Uncertainty in AI*, pages 467–475, 1999.

[18] C.St.J.A. Nash-Williams. Edge-disjoint spanning-trees of finite graph. *J. London Math. Soc.*, 36:445–450, 1961.

[19] J. Pearl. Evidential reasoning using stochastic simulation. *Artificial Intelligence*, 32:245–257, 1987.

[20] L. Tierney. Markov chains for exploring posterior distributions. *Annals of Statistics*, 4(22):1701–1762, 1994.

[21] D. J. Wilkinson and S. K. H. Yeung. Conditional simulation from highly structured gaussian systems, with application to blocking-MCMC for the Bayesian analysis of very large linear models. *Statistics and Computing*, 12(3):287–300, 2002.

[22] J. S. Yedidia, W. T. Freeman, and Y. Weiss. Understanding belief propagation and its generalizations. *Exploring Artificial Intelligence in the New Millennium*, pages 239–269, 2003.