THE UNIVERSITY OF CHICAGO

ADAPTIVE INFERENCE FOR GRAPHICAL MODELS

A DISSERTATION SUBMITTED TO
THE FACULTY OF THE DIVISION OF THE PHYSICAL SCIENCES
IN CANDIDACY FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

DEPARTMENT OF COMPUTER SCIENCE

BY
ÖZGÜR SÜMER

CHICAGO, ILLINOIS
FEBRUARY 2012

To my dad...

ABSTRACT

Many algorithms and applications involve repeatedly solving a variation of the same statistical inference problem. *Adaptive inference* is a technique where the previous computations are leveraged to speed up the computations after modifying the model parameters. This approach is useful in situations where a slow-to-compute statistical model needs to be re-run after some minor manual changes or in situations where the model is changing over time in minor ways; for example while studying the effects of mutations on proteins, one often constructs models that change slowly as mutations are introduced. Another important application of adaptive inference is in situations where the model is being used iteratively; for example in approximate inference we may want to decompose the problem into simpler inference subproblems that are solved repeatedly and iteratively using adaptive updates.

In this thesis we explore both exact inference and iterative approximate inference approaches using adaptive updates. We first present algorithms for adaptive exact inference on general graphs that can be used to efficiently compute marginals and update MAP configurations under arbitrary changes to the input factor graph and its associated elimination tree. We then apply them to approximate inference using a framework called *dual decomposition*. The key to our approach is a linear time preprocessing step which builds a data structure called a *cluster tree* that can efficiently be maintained when the underlying model is slightly modified. We demonstrate how a cluster tree can be used to compute any marginal in time that is logarithmic in the size of the input model. Moreover, our approach can also be used to update MAP configurations in time that is roughly proportional to the number of updated entries, rather than the size of the input model. This fact enables us to use our framework to speed up the convergence of dual-decomposition methods. Our technique is also amenable to parallelism, and we explore its ability to utilize multi-core parallelism in the context of dual-decomposition approximation methods.

The work in this thesis represents research performed in collaboration with Umut Acar, Alexander Ihler, and Ramgopal Mettu.

ACKNOWLEDGMENTS

First and foremost, I am indebted to the Department of Computer Science for funding me for six years and providing an ideal environment for intellectual development and for collaboration. The Ph.D. process was certainly not as easy as I once thought, and required many more skills other than intellectual prowess. I would like to thank the people who greatly helped me to acquire or advance these skills throughout my Ph.D. life, especially my official advisor László Babai and unofficial advisors Umut Acar, Ramgopal Mettu and Alexander Ihler.

I am honored to have been Laci's student throughout my PhD. He guided my first four years of research and trusted that I came back to the PhD program after a three year leave of absence. He was a phenomenal advisor, mentor and teacher and he always stood up for me.

I started my PhD journey to become a theoretical computer scientist, however after working in the financial industry while on leave, I decided to continue my work in Artificial Intelligence. While searching for a research subject, I met Umut Acar at TTI, who introduced me to the problem of adaptive inference. I am greatly indebted to him for being a great advisor, mentor and friend. His financial and logistic support especially throughout my seventh year enabled me to focus exclusively on the thesis and I am very grateful for this. Ram, the co-inventor of the original idea of adaptive inference with Umut, advised me during the summer quarters as an intern at University of Massachusetts at Amherst. He spent tremendous amount of time teaching me how to write research papers and provided valuable financial support. One of the most important aspects of the thesis, the code in this work was accomplished in collaboration with Ram. Alex provided the first prototype code and his input was crucial in shaping our first paper for publishing. He was a great teacher and mentor, he helped shape my ideas and set a high bar for my work. The way I now approach problems bears his profound influence. I was very fortunate to have great advisors and great co-authors in Alex, Ram and Umut.

University of Chicago provided a great opportunity for me to interact with many professors and fellow students. I thank Yasemin Altun and David McAllester for many helpful discussions in learning theory and Pedro Felzenszwalb being in my thesis committee. I am greateful to Bruno Codennetti for teaching me game theory and working with me on a couple of problems in the subject. I thank my fellow students Duru, Nanda, Irina and Vikas. I especially thank Duru for his support as a lifelong friend, for walking on the same PhD path and sharing many important things in life. Argentine tango and playing music together made the PhD more bearable, not to forget sharing a Nissan Maxima for many years. I am grateful to my friend Aytek who helped me settle in Chicago and guided me through my first years. I also want to thank my friends outside of University: Deniz, Emre, Mehmet, Baran, Jim and Issa.

I finally want to thank my family especially my mom, grandma and Miki. Without their encouragement, motivations and support, this PhD would have been much longer.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

Statistical modeling has proven to be useful in numerous application areas in computer science such as computational biology, computational learning theory and computer vision. Since the models used by these applications usually involve many random variables, we use *graphical models* to represent the conditional dependency relationships among the variables. The graphical models enable us to not only represent the model succinctly but also solve inference tasks more efficiently than using exhaustive search on the joint probability distribution. A graphical model is represented by a factor graph which is an undirected bipartite graph connecting variables and factors. The joint probability distribution for the model can be written as product of factors where each factor represents a local probability distribution over the variables it is connected to. The two commonly studied inference tasks on graphical models are computing the marginal probabilities and finding the maximum *a posteriori*, or MAP, configurations.

Considerable efforts have been made to understand and minimize the computational complexity of these inference tasks. However, in many applications we may need to repeatedly perform inference on variations of essentially the same model. It is therefore desirable to re-use as much information as we can from the previous computation while repeatedly solving the inference tasks. *Adaptive inference* refers to the problem of handling changes to the model more efficiently than performing inference from scratch. In this setting, the changes to the model can be made either to model parameters (factors) or to the dependency structure (adjacencies in the graph).

Adaptive inference arises in various computational biology applications such as mutagenesis. In mutagenesis, it is often desirable to study the effects of mutation on functional or structural properties of a gene or protein. The sequence analysis of a gene is commonly performed using a hidden Markov model while the analysis of proteins usually require a

factor graph defined by the three-dimensional topology of the protein of interest. For both of these types of models, each putative mutation gives rise to a new problem that is nearly identical to the previously solved problem.

Another setting where repeatedly performing inference plays an important role is iterative approximate inference, in particular *dual-decomposition methods*. These methods decompose a complex model into a collection of simpler models (sub-graphs) that are forced to agree on their intersections. By relaxing these constraints with Lagrange multipliers, one obtains a bound on the optimal solution that can be iteratively tightened. Adaptive inference can be used as a subroutine at each iteration which is especially useful if the successive models differ only on a small portion of the model.

The changes to the model described above can, of course, be handled by running known inference algorithms from scratch after incorporating the changes into the model. However, in general we may wish to assess thousands of potential changes to the model – for example, the number of possible mutations in a protein structure grows exponentially with the number of considered sites – and minimize the total amount of work required. It is therefore desirable to handle these changes more efficiently than performing inference from scratch. As a simple example, suppose that we wish to compute the marginal distribution of a leaf node in a Markov chain model with n variables. Using the standard dynamic programming algorithm, upon a change to the conditional probability distribution at one end of the chain, we must perform $\Omega(n)$ computation to compute the marginal distribution of the node at the other end of the chain. In an adaptive setting, however, it might be worth utilizing additional preprocessing time to restructure the underlying model in such a way that changes to the model can be handled in time that is logarithmic, rather than linear, in the size of the model.

1.1 Problems addressed

This thesis primarily provides a technique to solve exact inference tasks efficiently in the adaptive setting. In particular we address the following problems and applications for a given graphical model $G = (X + F, E)$.

- We initially address the problem of adaptive computation of marginals in a graphical model. We use a query/update framework for this problem. Each update operation represents a single modification either to the factors or to the graph structure by inserting or deleting an edge. A query operation is performing marginalization on a variable. Therefore, adaptively computing marginals refers to a collection of ℓ updates operation followed by a single query operation.

- We then focus on finding the maximum *a posteriori* (MAP) configurations. In this setting we again allow multiple changes to be performed before we compute the updated entries in the MAP configuration. This problem, however, differs from the former because the output size is no longer bounded and can be as large as the size of the graph. Therefore, it is essential in this problem to figure out the changed entries in the MAP configuration in a time that is proportional to the output size.

- We finally focus on the application of adaptivity in exact inference to dual-decomposition methods. The problem here is however not as simple as applying the adaptive exact inference methods to solve sub-problems at each iteration. The graph can always be decomposed into small and independent graphs, such as to independent factors and solved independently. This decomposition allows trivial implementation of adaptivity and also provides further advantages such as easy parallelization. The trade-off, however is that this decomposition requires more iterations to converge than other decompositions that use larger sub-graphs. The question here is to investigate if decompositions that use large sub-problems can be made adaptive and parallelizable so

that they run faster than decompositions that use small sub-problems.

We devote one chapter for each of these problems and applications.

1.2 Outline of our approach

Our high-level approach to enabling efficient updates of the model, and recalculation of marginals or a MAP configuration, is to "cluster" parts of the input model by computing partial eliminations, and construct a balanced-tree data structure with depth $O(\log n)$. We use a process based on factor elimination [16] that we call *hierarchical clustering* that takes as input a graph and elimination tree (equivalent to a tree-decomposition of the graphical model), and produces an alternative, balanced elimination sequence. The sufficient statistics of the balanced elimination are re-usable in the sense that they will remain largely unchanged by any small update to the model. In particular, changes to factors and the variables they depend on can be performed in time that is logarithmic in the size of the input model.

1.3 Thesis Organization

This thesis addresses the question of solving various exact inference tasks in the adaptive setting and the application of adaptive techniques to dual-decomposition methods. After putting our contributions into context in the related work chapter, the background chapter presents the necessary materials on which our approach is based. The next chapter is devoted to presenting our algorithm in the standard case without taking the adaptive updates into account. The remaining chapters addresses the three questions described in Section 1.1 one by one and the implementation details.

Related Work. The related work (Chapter 2) presents previous work that either our approach is based on or address adaptive inference problems in graphical models. One way or another, many of the adaptive or parallel algorithms mentioned in this chapter uses a

tree-balancing approach. Hence, we present several tree-balancing techniques with a focus on the parallel tree-contraction that we use in our approach. In the remaining two sections, we first present the first work on adaptive inference [19] that uses tree-contraction and then describe how we improve upon their results using the ideas from self-adjusting framework.

Background. We begin with an overview of an exact inference technique on graphical models called *factor elimination* and how it is related to another well-known exact inference technique called *tree-decomposition*. In this thesis, we use the link between these frameworks by providing algorithms in one framework, factor elimination, and then establishing bounds using the other framework, tree-decomposition. In the context of exact inference, we only review the formulas for marginal computations and leave how these formulas can be modified to compute MAP configurations to Chapter 6. The last section reviews the dual-decomposition methods and the inherit trade-offs among existing subproblem choices.

Cluster Tree Data Structure. Chapter 4 is the main chapter where we explain our method as an extension of factor elimination algorithm. The extensions we make to the ordinary factor elimination algorithm necessitates constructing an additional data structure called *cluster tree*. This data-structure plays a crucial role in making the algorithm adaptive and parallel which will be explored later. In order to keep the exposition clear and simple, in this chapter we only focus on the marginalization problem and present how one can use the cluster tree data structure to compute marginals.

Efficient Updates to Cluster Trees. Having explained how to compute marginals in the previous chapter, Chapter 5 focuses on how modifications to the model can be incorporated into our data structure so that the marginals can be computed in time logarithmic to the size of the graph. Naturally the modifications to the model fall into two categories, the modifications that keep the underlying cluster tree intact and those that modify the cluster tree. The

bulk of the chapter focuses on the latter as they turn out to be important and challenging. The algorithm and proofs we present in this chapter is based on self-adjusting computation [3]. The reader is encouraged to review Chapter 2 to see the differences between our work and [3]. We also explore the practical performance of our methods with extensive synthetic experiments. Part of the experiments in this chapter focus on assesing the practical limits of our techniques, others focus on quantifying the practical performance of our theoretical bounds.

Maintaining MAP Configurations Chapter 6 addresses the second problem in Section 1.1, the computation of MAP configurations. We modify both the factor elimination procedure and the data we store on the cluster tree data structure to maintain MAP configurations under adaptive updates to the underlying model. The experimental section provides two important applications: protein secondary structure prediction and side-chain packing. This chapter also acts as a step-stone to the following chapter because Chapter 7 requires maintenance of MAP configurations as a sub-procedure.

Parallelism for approximate inference. Chapter 7 addresses the final question described in Section 1.1. Here, in addition to adaptivity, we also describe how computations on cluster tree can be performed efficiently in parallel. We apply our adaptive exact inference framework to dual-decomposition and show its effectiveness using experimental data.

Implementation. Chapter 8 provides python code that implements adaptive exact inference to compute marginals and MAP configurations. The implementation closely follows the exposition of the thesis and provides some explanation to a certain degree for the variables and functions defined in the code. This chapter is mainly intended to provide code to clarify the algorithms described in Chapters 4, 5 and 6.

1.4 Contributions

In this thesis, we present a new framework for adaptive exact inference, building upon the work of [19]. Given a factor graph G with n nodes, and domain size d (each variable can take d different values), we require the user to specify an *elimination tree* T on factors. An elimination tree guides the order in which the factors are eliminated and in general it is either expected to be specified by the user or estimated by known greedy algorithms. In our framework, we give the option of supplying elimination tree to the user. The choice of elimination tree determines the *width* w which is a graph theoretical concept measuring the extend the graph resembles a tree.

The remainder of the section lists our contributions under two categories: (i) algorithmic and theoretical contributions and (ii) experimental contributions. In part (ii), we further discuss the scope that our theoretical results are applicable. We performed extensive synthetic experiments to determine the cases where our approach is preferable.

1.4.1 Algorithmic and Theoretical Contributions

For exact inference, our framework for adaptive inference requires a preprocessing step in which we build a cluster tree — a balanced representation of the input elimination tree — in $O(d^{3w} \cdot n)$ time where w is the width of the input elimination tree T. For approximate inference w is always 1, so the preprocessing step takes $O(d^3 \cdot n)$ time.

Given a graphical model with n nodes and elimination tree with width w, we show that

- the cluster tree is essentially equivalent to a tree-decomposition,
- for marginal computations, a change to the model can be processed in $O(d^{3w} \cdot \log n)$ time, and the marginal for particular variable can be computed in $O(d^{2w} \cdot \log n)$ time,
- for a change to the model that induces ℓ changes to a MAP configuration, our approach

can update the MAP configuration in $O(d^{3w} \log n + d^w \ell \log(n/\ell))$ time, without knowing ℓ or the changed entries in the configuration.

- in addition to being adaptive, the cluster tree is also highly parallelizable and can be used to speed-up dual-decomposition solvers.

1.4.2 Experimental Contributions

As in standard approaches for exact inference in general graphs, our algorithm has an exponential dependence on the width of the input model. The dependence in our case, however is stronger: if the input elimination tree has width w, our balanced representation is guaranteed to have width at most $3w$. As a result the running time of our algorithms for building the cluster tree as well as the updates have a $O(d^{3w})$ multiplicative factor; updates to the model and queries however require logarithmic, rather than linear, time in the size of the graph. Our approach is therefore most suitable for settings in which a single build operation is followed by a large number of updates and queries.

To evaluate the practical effectiveness of our approach, we implement the proposed algorithms and present an experimental evaluation by considering both synthetic data (Section 6.3.1) and real data (Sections 6.3.2 and 6.3.3). Our experiments using synthetically generated factor graphs show that even for modestly-sized graphs ($10 - 1000$ nodes) our algorithm provides orders of magnitude speedup over computation from scratch for computing both marginals and MAP configurations. Thus, the overhead observed in practice is negligible compared to the speedup possible using our framework.

In addition, we also show the applicability of our framework to two problems in computational structural biology (Sections 6.3.2 and 6.3.3). First, we apply our algorithm to protein secondary structure prediction using an HMM, showing that secondary structure types can be efficiently updated as mutations are made to the primary sequence. For this application, our algorithm is one to two orders of magnitude faster than computation from scratch. We

also apply our algorithm to protein sidechain packing, in which a (general) factor graph defines energetic interactions in a three-dimensional protein structure and we must find a minimum-energy conformation of the protein. For this problem, our algorithm can be used to maintain a minimum-energy conformation as changes are being made to the underlying protein. In our experiments, we show that for a subset of the SCWRL benchmark [12], our algorithm is nearly 7 times faster than computing minimum-energy conformations from scratch.

We then apply these findings to dual-decomposition methods; we make use of the cluster tree data structure for adaptive inference and show that it can be used in a way that combines the per-iteration advantages of large subproblems while also enabling a high degree of parallelism. We demonstrate that a dual-decomposition solver using our cluster tree approach can improve its time to convergence significantly over other approaches. For random grid-like graphs, we obtain one to two orders of magnitude speedup. We also use our solver for a real-world stereo matching problem, and over a number of data sets show a factor of about 2 improvement over other approaches.

1.5 Acknowledgments

The research in this thesis is done jointly with Umut A. Acar, Alexander T. Ihler and Ramgopal R. Mettu and has also been published in the form of conference and journal papers. Our first published article was specific to tree-structured graphical models and appeared in NIPS [1]. Chapters 4 and 5 presents our algorithm for solving marginalization in general graphical models and is derived from a conference paper [8]. Chapter 6, on maintaining the MAP configurations under changes is first published in SSP [9]. The detailed algorithmic description of how updates are performed (Chapter 5) and the proofs of the theorems in Chapters 4, 5 and 6 are published in a journal paper [47]. Finally, application of our framework to dual-decomposition solvers (Chapter 7) has been published in AAAI [48].

CHAPTER 2

RELATED WORK

The problem of adaptive inference is studied extensively in the artificial intelligence community, in particular in the context of graphical models. Our goal in this chapter is to provide related methods published earlier and put our work in a context. We give special attention to parallel tree-contraction and self-adjusting computation as they provide the main tools that we use in our own approach.

2.1 General Results of Adaptivity in Artificial Intelligence

There are numerous machine learning and artificial intelligence problems, such as path planning problems in robotics, where new information or observations require changing a previously computed solution. As an example, problems solved by heuristic search techniques have benefited greatly from incremental search algorithms [43, 21, 31], in which solutions can be efficiently updated by reusing previously searched parts of the solution space. Incremental search algorithms solve dynamic shortest path problems where shortest paths have to be found repeatedly while the underlying topology of a graph or its edge weights are changing. A robot, as an example, might have to re-plan its shortest path route when it detects a previously unknown obstacle.

Because the literature of incremental search shows similarities to our approach it is helpful to review the approaches previously taken by this community and compare against our approach.

- **Kinds of dynamic changes allowed.** Researchers have approached the dynamic shortest path problems under various assumptions depending on their particular problem domain. In the most general case where arbitrary sequence of edge insertions, deletions and weight changes are allowed, the problem is called the "fully dynamic

shortest path problem" as addressed in [43, 21, 31]. This formulation is similar to ours, we also allow arbitrary sequence of changes to the topology of the graphical model as well as to the factors.

- **Analysis.** The analysis of incremental algorithms differ from classical analysis of batch algorithms because in the worst case, any incremental algorithm requires as much time as running from scratch when the whole input is changed. In order to meaningfully compare the incremental algorithms with each other, [43] suggested to measure the cost of an incremental algorithm as a function of the sum of the size of the "changes" in the input and the output, rather than expressing solely in terms of the "whole" input size. Our running costs are similarly expressed; in particular, Chapter 6 describes the first, to our knowledge, algorithm that dynamically maintains the MAP configurations in graphical models and expresses its running costs in terms of the changes in input and output sizes.

- **Methods.** Finally, incremental search achieves adaptivity by formulating their algorithms as a distributed algorithm where each node computes a value using the information on the neighbor nodes only. When a problem is set up in a distributed manner, both adaptivity and parallelism can be achieved rather trivially. To make it adaptive, we update the information on the nodes where the modification is performed and propagate that change to the neighbors until all nodes are updated. Parallelism can be achieved by assigning a processor to every node and computing the information on a node as soon as one of its neighbors has a new piece of information. Even though our adaptivity and parallelism approach follows the same footsteps, our method differs significantly by also balancing the graph, hence making the change propagation step provably efficient even in the worst case.

2.2 Delcher's Work on Adaptive Inference

In the context of graphical models, the problem of performing adaptive inference was first considered by Delcher et al [19]. In their work, they introduced a logarithmic time method for updating marginals under changes to observed variables in the model. At a high level their approach is similar to our own, in that they also use a linear time preprocessing step to transform the input tree-structured model into a balanced tree representation using tree-contraction. They use query/update model like our model, where a number of update operations is followed by a query operation. Like ours, their query operation finds the marginal distribution for a single variable.

Our work significantly extends their work and explores the method in depth in the following ways.

- While their algorithm relies on the input model being tree-structured, our algorithm has no such restriction and can work on any graphical model.
- Their algorithm is designed to compute marginals only. In addition to marginals, our algorithm can also compute and maintain MAP configurations.
- Their algorithm can only handle changes to observations which are the factors that depend on a single variable and cannot update dependencies in the input model. Our algorithm allows arbitrary changes to any of the factors in the factor graph. Moreover, we allow modifications to the graphical model and to the elimination tree.
- They set up the problem using directed Bayesian networks with conditional probability tables whereas we use relatively modern factor graphs to represent a graphical model.

While their algorithm can be applied to general graphs by performing a tree decomposition, it is not clear whether the tree decomposition itself can be easily updated, as is necessary to remain efficient when modifying the input model.

2.3 Parallel Balancing Approaches and Tree-Contraction

The data structures required for adaptive computation has been closely related to those required for parallel processing. The approach in both cases is to split-up the problem instance into independent sub-pieces. For parallelism, we exploit the ability of computing many sub-pieces simultaneously, whereas in adaptivity, a small change usually requires us to update the information within a single piece, saving significant computation time compared to computing from scratch. The dynamic programming approaches to solve the exact inference problems can naturally be divided into independent sub-pieces, making them a good candidate for parallelization. In fact, both Pearl's original Belief Propagation algorithm [41] and the junction-tree algorithm of Lauritzen [35] were conceived as distributed algorithms where each variable or clique could be associated with a separate processor and perform the elimination on that node as soon as the inputs from neighbor nodes are available. In addition to the available parallelism in the topology of the graph, [15] and [33] further exploited the parallelism on a single node during the elimination phase.

None of these early approaches however, attempted to map the unbalanced graphs into a balanced structure to improve parallelization. Efficient parallel implementations of exact inference even for unbalanced graphs were proposed by [42, 40, 55]. They use a pointer jumping technique to balance the computations on the chains. They first map each chain (connected nodes of degree two) in a tree-structured graph into a balanced binary tree, hence making the whole tree balanced by making its depth logarithmic in the size of the graph. They then apply dynamic programming in parallel in the balanced graph. The tree-balancing technique that both Delcher [19] and we use, however is inspired by another method known as *parallel tree contraction*, devised by Miller and Reif [39] to evaluate expressions on parallel architectures. Parallel tree contraction in essence is similar to pointer jumping, however its formulation is cleaner and more flexible, making it easier to use especially for tree structured problems. In the original paper of [39], parallel tree contraction is used to evaluate

a given expression tree, where internal nodes are arithmetic operations and leaves are input values. Their parallel algorithm works by "contracting" both leaves and internal nodes of the tree in rounds. At each round, the nodes to eliminate are chosen in a random fashion and it can be shown that, in expectation, a constant fraction of the nodes are eliminated in each round. By performing contractions in parallel, the expression tree can be evaluated in logarithmic time and linear total work. Parallel tree contraction can be applied to any semi-ring, including sum-product (marginalization) and max-product (maximization) operators, making it directly applicable to inference problems.

2.4 Self-adjusting Computation

The algorithms and techniques that we use in this thesis builds on previous work on self-adjusting computation [2, 3, 5]. Self-adjusting computation is a general-purpose technique for allowing computation to respond automatically to changes to their input data. At a high-level, the idea behind self-adjusting computation is to construct a trace of a computation and use a change propagation algorithm to update the trace and the output when the input data changes. The change-propation algorithm uses the trace to identify the parts of the computation that are affected by a change and rebuilds the affected pieces by selectively re-executing pieces of the computation to update the output. If self-adjusting computation is applied to algorithms that generate stable traces, where making a small change to the input affects a relatively small (polylogarithmic in general) number of elements of the trace, then change propagation algorithm can update the trace and the output efficiently.

In previous work, Acar et al [3] showed that the tree-contraction algorithm for computing various properties of trees is stable and thus properties of dynamically changing tree can be computed efficiently via self-adjusting computation. Acar et al [5] also show that the trace of the execution of a tree-contraction algorithm can be concretely represented by a rake-and-compress tree (RC-tree) data structure, which can be viewed as a balanced decomposition of

the tree itself. Acar et al shows that the approach can be used to solve a number of problems on trees.

Our approach to performing adaptive inference builds on these ideas on computing properties of dynamic trees via self-adjusting computation. Specifically, our cluster trees are similar to RC-trees. However, none of the previous work on self-adjusting computation considers statistical inference. Similarly, there is no work on showing how self-adjusting computation can be used to compute and update inferences efficiently on graphs (as we need for inference on loopy graphical models). In addition to these differences, there are also several more technical differences, which we describe below.

- The RC-Tree data structure is built and maintained by a randomized algorithm where our cluster-tree data structure uses a deterministic algorithm. Therefore, our algorithm creates a balanced cluster-tree with depth deterministically logarithmic in the size of the graph whereas the RC-Tree data structure has logarithmic depth only in expectation. Besides having theoretical value, our deterministic algorithm also has a practical advantage of producing lower depth data structures than the randomized procedure used by the self-adjusting framework. This feature enables us to establish deterministic worst-case bounds for our algorithms and makes sure that they have competitive practical running times.

- The randomized procedure used by the self-adjusting computation framework ensures that the RC-Tree data structure is history independent whereas our deterministic procedure has no such property. History independence - the ability to dynamically maintain a data structure so that it does not reveal the history of insertions and deletions to the factor graph - is important in certain problem domains where privacy is a concern. Even though our algorithm is not theoretically history independent, it is a very difficult task for an adversary to recover even a small portion of the history of insertions and deletions from our representation.

- While describing the change propagation algorithm in the context of the tree-contraction, [3] coins a new term "affected nodes" for the nodes that have different information from one run to the next run after the input changes. These nodes are the ones where the information on them has to be re-computed in order to maintain the RC-Tree data structure, hence the proof of the theoretical correctness and efficiency of the update algorithm depends on how these affected nodes are defined and analyzed. The description given in [3] for this set is defined more as an abstract mathematical object rather than an implementable algorithmic concept. In this thesis, we give a new definition of "affected set of nodes" that can be efficiently implemented. However, our concept of affected nodes is not as tight as the affectedness defined in self-adjusting computation framework, hence our definition of affectedness can possibly yield inferior theoretical bounds. Despite this disadvantage, we prove the same efficiency bound on the size of the affected nodes after a modification is made to the input. Moreover, due to the deterministic nature of our approach, our theoretical bounds have better constant terms than their bounds have.

In recent years, there has been a lot of progress on developing a language-based appraoch to self-adjusting computation and applying to problems from a several domains. Ley-Wild et al [37] extended Standard ML language and Hammer et al [24] extended the C language to enable writing self-adjusting programs. The technique has also been applied to problems in geometry, specifically in motion simulation [4], and in meshing [6, 7], as well as to large-scale processing of data [11].

CHAPTER 3

BACKGROUND

3.1 Factor Graphs

Factor graphs [34] describe the factorization structure of the function $g(X)$ using a bipartite graph consisting of *variable* nodes and *factor* nodes. Specifically, suppose such a graph $G = (X, F)$ consists of variable nodes $X = \{x_1, \ldots, x_n\}$ and factor nodes $F = \{f_1, \ldots, f_m\}$ (see Figure 3.1a). Let $X_{f_j} = \{x_i \in X : x_i \text{ is adjacent to } f_j \text{ in } G\}$ be the set of variables that the factor f_j depends on. For example, in Figure 3.1a, $X_{f_5} = \{x, v\}$. G is said to be consistent with a function $g(\cdot)$ if and only if

$$g(x_1, \ldots, x_n) = \prod_j f_j$$

for some functions f_j whose arguments are the variable sets X_{f_j}. We omit the arguments X_{f_j} of each factor f_j from our formulas. In a common abuse of notation, we use the same symbol to denote a variable (resp., factor) node and its associated variable x_i (resp., factor f_j). We assume that each variable x_i takes on a finite set of values.

In this thesis we first study the problem of marginalization of the function $g(X)$. Specifically, for any x_i we are interested in computing the marginal function

$$g^i(x_i) = \sum_{X \setminus x_i} g(X).$$

Once we establish the basic results for performing adaptive inference, we will also show how our methods can be applied to another commonly studied inference problem, that of finding

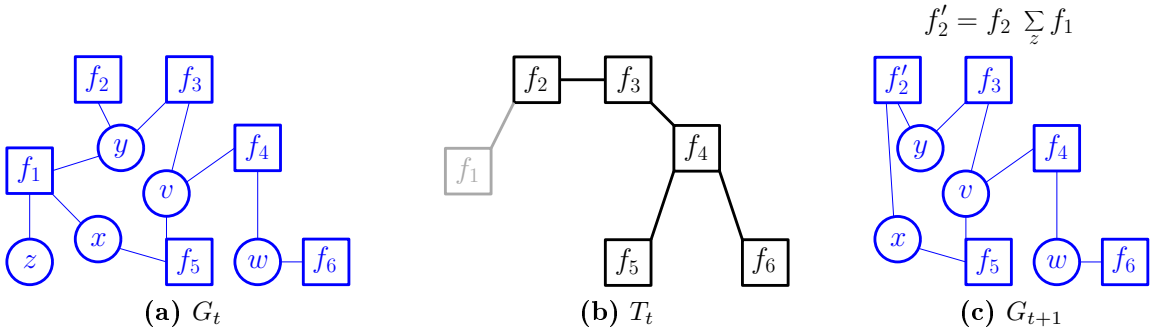


Figure 3.1: Factor elimination. Factor elimination takes a factor graph G_1 and an elimination tree T_1 as input and sequentially eliminates the leaf factors in the elimination tree. As an example, to eliminate f_1 in iteration t, we first marginalize out any variables that are only adjacent to the eliminated factor, and then propagate this information to the unique neighbor in T_t, i.e. $f_2' = f_2 \sum_z f_1$.

the configuration of the variables that maximizes g, that is,

$$X^* = \arg\max_X g(X)$$

In this thesis, we call the vector X^* the *maximum a posteriori* (MAP) configuration of X.

3.2 Exact inference

It is well-known that both marginalization and finding MAP configurations is NP-hard if no assumptions are made about the structure of the underlying graphical model [14]. It is however possible to solve it exactly in certain important speacial cases such as for tree-structured graphs. Many essentially equivalent algorithms are proposed for solving tree-structured, including belief propagation [41] or sum-product [34], and for general graphs, bucket elimination [18], recursive conditioning [17], junction-trees [35] and factor elimination [16].

The basic structure of these algorithms is iterative; in each iteration partial marginalizations are computed by eliminating variables and factors from the graph. The set of variables

and factors that are eliminated at each iteration is typically guided by some sort of auxiliary structure on either variables or factors. For example, the sum-product algorithm simply eliminates variables starting at leaves of the input factor graph. In contrast, factor elimination uses an *elimination tree* T on the factors and eliminates factors starting at leaves of T; an example elimination tree is shown in Figure 3.1b.

3.2.1 Factor Elimination

For a particular factor f_j, the basic operation of *factor elimination* eliminates f_j in the given model and then propagates information associated with f_j to neighboring factors. At iteration t, we pick a leaf factor f_j in T_t and eliminate it from the elimination tree forming T_{t+1}. We also remove f_j along with all the variables $\mathcal{V}_j \subseteq X$ that appear only in factor f_j from G_t forming G_{t+1}. Let f_k be f_j's unique neighbor in T_t. We then partially marginalize f_j, and update the value of f_k in G_{t+1} and T_{t+1} with

$$\lambda_j = \sum_{\mathcal{V}_j} f_j \qquad\qquad f'_k = f_k \lambda_j. \tag{3.1}$$

For reasons that will be explained in Section 4.1, we use the notation λ_i to represent the partially marginalized functions; for standard factor elimination these operations are typically combined into a single update to f_k. Finally, since multiplying by λ_j may make f'_k depend on additional variables, we expand the argument set of f'_k by making the arguments of λ_j adjacent to f'_k in G_{t+1}, i.e. $X_{f'_k} := X_{f_k} \cup X_{f_j} \setminus \mathcal{V}_j$. Figure 3.1 gives an example where we apply factor elimination to a leaf factor f_1 in the elimination tree. We marginalize out the variables that are only adjacent to f_1 (i.e., $\mathcal{V}_1 = \{z\}$) and update f_1's neighbor f_2 in the elimination tree with $f'_2 = f_2 \sum_{\mathcal{V}_1} f_1$. Finally, we add an edge between the remaining variables $X_{f_1} \setminus \mathcal{V}_1 = \{x\}$ and the updated factor f'_2.

Suppose we wish to compute a particular marginal $g^i(x_i)$. We root the elimination tree

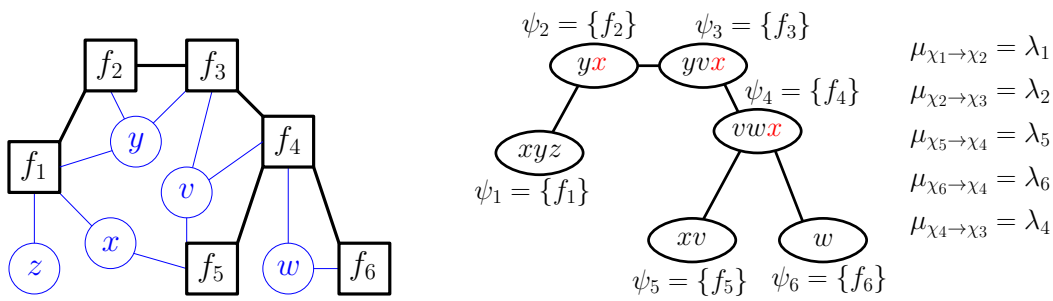


Figure 3.2: Factor trees and tree decompositions. A tree-decomposition (right) that is equivalent to a given elimination tree (left) can be obtained by first replacing each factor with a hyper-node that contains the variables adjacent to that factor node and then adding variables to the hyper-nodes so that the running intersection property is satisfied.

at a factor f_j such that x_i is adjacent to f_j in G, then eliminate leaves of the elimination tree one at a time, until only one factor remains. By definition the remaining factor f'_j corresponds to f_j multiplied by the results of the elimination steps. Then, we have that $g^i(x_i) = \sum_{X \setminus x_i} f'_j$. All of the marginals in the factor graph can be efficiently computed by re-rooting the tree and reusing the values propagated during the previous eliminations.

Factor elimination is equivalent to bucket (or variable) elimination [30, 16] in the sense that we can identify a correspondence between the computations performed in each algorithm. In particular, the factor elimination algorithm marginalizes out a variable x_i when there is no factor left in the factor graph that is adjacent to x_i. Therefore, if we consider the operations from the variables' point of view, this sequence is also a valid bucket (variable) elimination procedure. With a similar argument, one can also interpret any bucket elimination procedure as a factor elimination sequence. In all of these algorithms, while marginal calculations are guaranteed to be correct, the particular auxiliary structure or ordering determines the worst-case running time. In the following section, we analyze the performance consequences of imposing a particular elimination tree.

3.2.2 Viewing elimination trees as tree-decompositions

For tree-structured factor graphs, the typical choice for the elimination tree is based on the factor graph itself. However, when the input factor graph is not tree-structured, we must choose an elimination ordering that ensures that the propagation of variables over the course of elimination is not too costly. In this section, we outline how a particular elimination tree can be related to a tree decomposition on the input graph (e.g., as in [17, 30]), thereby allowing us to use the quality of the associated tree decomposition as a measure of quality for elimination trees. In subsequent sections, this relationship will enable us to compare the constant-factor overhead associated with our algorithm against that of the original input elimination tree.

Let $G = (X, F)$ be a factor graph. A *tree-decomposition* for G is a triplet $(\chi, \psi, \mathcal{D})$ where $\chi = \{\chi_1, \chi_2, \ldots, \chi_m\}$ is a family of subsets of X and $\psi = \{\psi_1, \psi_2, \ldots, \psi_m\}$ is a family of subsets of F such that $\cup_{f \in \psi_i} X_f \subseteq \chi_i$ for all $i = 1, 2, \ldots, m$ and $\mathcal{D}$ is a tree whose nodes are the subsets χ_i satisfying the following properties:

1. *Cover property:* Each variable x_i is contained in some subset belonging to χ and each factor $f_j \in F$ is contained in exactly one subset belonging to ψ.
2. *Running Intersection property:* If $\chi_s, \chi_t \in \chi$ both contain a variable x_i, then all nodes χ_u of the tree in the (unique) path between χ_s and χ_t contain x_i as well. That is, the nodes associated with vertex x_i form a connected sub-tree of $\mathcal{D}$.

Any factor elimination algorithm can be viewed in terms of a message-passing algorithm in a tree-decomposition. For a factor graph G, we can construct a tree decomposition $(\chi, \psi, \mathcal{D})$ that corresponds to an elimination tree $T = (F, E)$ on G. First, we set $\psi_i = \{f_i\}$ and $\mathcal{D} = (\chi, E')$ where $(\chi_i, \chi_j) \in E'$ is an edge in the tree-decomposition if and only if $(f_i, f_j) \in E$ is an edge in the elimination tree T. We then initialize $\chi = \{X_{f_1}, X_{f_2}, \ldots, X_{f_m}\}$ and add the minimal number of variables to each set χ_j so that the running intersection property is satisfied. By construction, the final triplet $(\chi, \psi, \mathcal{D})$ satisfies all the conditions of

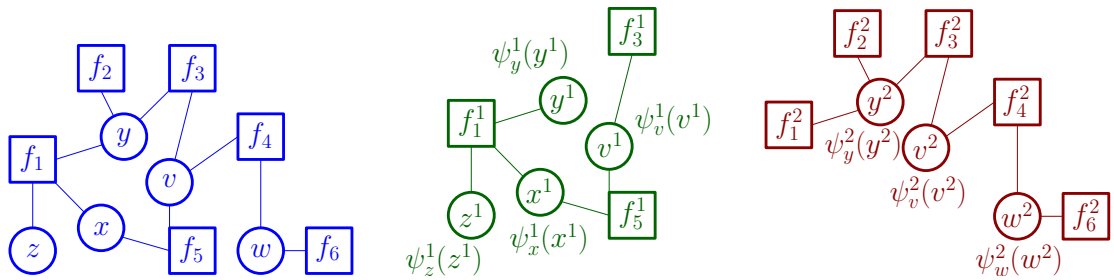
a tree-decomposition. This procedure is illustrated in Figure 3.2. The factor graph (light blue edges) and its elimination tree (bold edges) on the left is equivalent to the tree-decomposition on the right. We first initialize $\chi_j = X_{f_j}$ for each $j = 1, \ldots, 6$ and add necessary variables to sets χ_j to satisfy the running intersection property: x is added to χ_2, χ_3 and χ_4. Finally, we set $\psi_j = \{f_j\}$ for each $j = 1, \ldots, 6$.

Using a similar procedure, it is also possible to obtain an elimination tree equivalent to the messages passed on a given tree-decomposition. We define two messages for each edge (χ_i, χ_j) in the tree decomposition: the message $\mu_{\chi_i \to \chi_j}$ from χ_i to χ_j is the partial marginalization of the factors on the χ_i side of $\mathcal{D}$, and the message $\mu_{\chi_j \to \chi_i}$ from χ_j to χ_i is the partial marginalization of the factors on the χ_j side of $\mathcal{D}$. The outgoing message $\mu_{\chi_i \to \chi_j}$ from χ_i can be computed recursively using the incoming messages $\mu_{\chi_k \to \chi_i}$ except for $k = j$, that is,

$$\mu_{\chi_i \to \chi_j} = \sum_{\chi_j \setminus \chi_i} f_i \prod_{(\chi_k, \chi_i) \in E' \setminus \{(\chi_j, \chi_i)\}} \mu_{\chi_k \to \chi_i}. \tag{3.2}$$

The factor elimination process can then be interpreted as passing messages from leaves to parents in the corresponding tree-decomposition. The partial marginalization function λ_i computed during the elimination of f_i is identical to the message $\mu_{\chi_i \to \chi_j}$ where f_j is the parent of f_i in the elimination tree. This equivalence is illustrated in Figure 3.2 where each partial marginalization function λ_j is equal to a sum-product message $\mu_{\chi_j \to \chi_k}$ for some k. This example assumes that f_3 is eliminated last.

For an elimination tree T, suppose that the corresponding tree decomposition is $(\chi, \psi, \mathcal{D})$. For the remainder of this thesis, we will define the *width* of T to be the size of the largest set contained in χ minus 1. Inference performed using T incurs a constant-factor overhead that is exponential in its width; for example, computing marginals using an elimination tree T of width w takes $O(d^{w+1} \cdot n)$ time and space where n is the number of variables and d is the domain size. The width of a graph G without any reference to an elimination tree is the smallest width among all elimination trees.



(a) G **(b)** G^1 **(c)** G^2

Figure 3.3: Dual-decomposition. The factor graph G is decomposed into two subgraphs G^1 and G^2 by creating copies of each variable for each subgraph. The factor copies must agree with the original factor: $f_1(x, y, z) = f_1^1(x, y, z)f_1^2(y)$ and $f_3(v, y) = f_3^1(v)f_3^2(v, y)$. Similarly, the multiplication of the unary functions that belong to the same variable copy must be 1: $\psi_v^1(v)\psi_v^2(v) = 1$ and $\psi_y^1(y)\psi_y^2(y) = 1$.

3.3 Approximate Inference

For some graphical models with high tree-width, exact inference is intractable and often the marginals or the MAP configurations need to be approximated. In the context of approximate inference, we focus only on the MAP estimation problem and use the technique of dual-decomposition, based on Lagrangian relaxation, to solve these inference problems by decomposing them into simpler components that are repeatedly solved and combined into a global solution.

3.3.1 Dual-decomposition

For notational simplicity of the formulas related to dual-decomposition methods, we assume that there is a unary factor $\psi_i(x_i) = 1$ for every variable x_i. Therefore, the $g(X)$ function can be rewritten as

$$g(X) = \prod_{x_i \in X} \psi_i(x_i) \cdot \prod_{f_j \in F} f_j.$$

Dual-decomposition methods further split the function $g(X)$ into a multiplicative set of

"subproblems"

$$g_{DD}(\cup_t X^t) = \prod_t \left[\prod_{x_i \in X} \psi_i^t(x_i^t) \cdot \prod_{f_j \in F} f_j^t(X_{f_j^t}) \right]$$

where each subproblem indexed by t depends on a new set of copied variables $X^t = \{x_1^t, \ldots, x_n^t\}$ and the decomposition satisfies

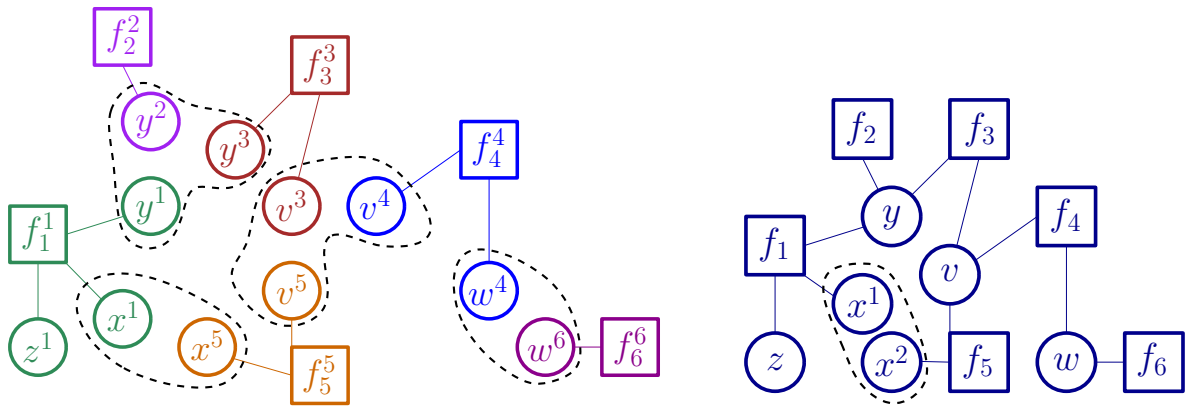
$$\psi_i(x_i) = \prod_t \psi_i^t(x_i) \quad \text{and} \quad f_j = \prod_t f_j^t(X_{f_j}) \quad \text{for all } x_i \in X \text{ and } f_j \in F. \tag{3.3}$$

The new factors f_j^t depend on $X_{f_j^t} = \left\{ x_i^t : x_i \in X_{f_j} \right\}$ and are chosen to be non-unit on a strictly simpler graph than G (for example, a tree), so that each subproblem t can be solved easily by exact inference. For example, Figure 3.3 illustrates the decomposition of the factor graph G into two tree-structured subgraphs G^1 and G^2 where the conditions of Equation (3.3) are satisfied. For stability of arithmetic operations, the computations are usually carried out by taking the negative logarithm of the factors. We define "dual energy" of a decomposition as $-\log(g_{DD}(\cup_t X^t))$.

If all copies x_i^t of each variable x_i are forced to be equal, both problems are equivalent. However, by relaxing this constraint and enforcing it with Lagrange multipliers, we obtain a collection of simpler problems that when solved individually, upper bounds the original function $g(X)$. Typical dual-decomposition solvers minimize this dual upper bound (or maximize the dual energy) using a projected subgradient update. Suppose that $\{x_i^t\}$ are the copies of variable x_i, with optimal assignments $\{a_i^t\}$ for $t = 1 \ldots T$. Then we modify ψ_i^t as

$$\psi_i^t(x_i) = \psi_i^t(x_i) \left(\frac{e^{\delta(x_i = a_i^t)}}{\left(\prod_{u=1}^T e^{\delta(x_i = a_i^u)} \right)^{\frac{1}{T}}} \right)^{\gamma} \tag{3.4}$$

where $\delta(\cdot)$ is the Kronecker delta function and γ is a step-size constant. It is easy to see that this update maintains the constraints on the ψ_i^t. If at any point a solution is found in which



(a) Independent Factors **(b)** Cover Tree

Figure 3.4: Independent Factors vs. Cover Tree. Decomposing graph into independent factors (a) leads to a relaxation that is amenable to parallelization and can be easily made adaptive. Even though the cover tree (b) is not easily parallelizable and does not naturally support adaptivity, it is better at improving the dual bound at each iteration.

all variable copies share the same value, this configuration must be the MAP configuration.

Dual decomposition solvers are closely related to LP-based loopy message passing algorithms [52, 23], which solve the same dual using a coordinate ascent fixed point update. However, these algorithms can have sub-optimal fixed points, so gradient and "accelerated" gradient methods [26, 27] are often preferred. In this thesis we focus on the standard projected subgradient method.

3.3.2 Subproblem choice: Independent Factors vs. Cover Tree

Dual-decomposition methods leave the choice of subproblems to the user. All problem collections that include equivalent sets of cliques (for example, any collection of trees that covers all factors of G) can be shown to have the same dual-optimal bound, but the actual choice of problems can significantly affect convergence speed. For example, one simple option is to include one subproblem per factor in the original graph; this leads to a large number of simple problems, which can then be easily parallelized. However, as observed in [32], choosing larger subproblems can often improve the convergence rate (i.e., the decrease in the upper bound per iteration) at the possible expense of parallelization. For example,

single-subproblem models [26, 57] create a single "covering" tree over several copies of each variable. This approach provides good convergence properties but is not easily amenable to parallelization. This trade-off can be observed in Figure 3.4. When individual factors are chosen as subproblems, they can be solved independently in parallel as shown in Figure 3.4a. However, because there are high number of variable copies, the copies are less likely to agree after a single iteration, which in turn slows down the overall convergence rate. The cover tree in Figure 3.4b on the other hand, has only a single variable x with multiple copies x^1 and x^2. They are much more likely to agree after a single iteration compared to independent factors, which speeds up the convergence.

Another advantage we propose for small subproblems is their ability to be *adaptive*, or more specifically to re-use previous iteration's solution. The subgradient update Equation (3.4) depends only on the solution a^t of each subproblem t; if all parameters of a subproblem are unchanged, its solution remains valid and we need not re-solve it. We show in Sections 7.1 and 7.4 that this can lead to considerable computational advantages. However, although this is common in very small subproblems (such as individual factors), for larger problems with better convergence rates it becomes less likely that the problem will not be modified. For example, in Figure 3.4a, if x^1 and x^5 are the only variable copies that disagree when the MAP configuration is computed independently for each subproblem. In the next round, we only need to re-solve the subproblems corresponding to factors f_1 and f_5. For cover tree in Figure 3.4b, as long as there is disagreement among copies, the MAP configuration for the whole tree has to be re-computed in the next round.

Thus, collections of small problems have significant speed (time per iteration) advantages, but larger problems have typically better convergence rates, or fewer iterations required. The focus of Chapter 7 is to present a new framework that captures both the convergence properties of single-subproblem approaches, and the update speed of many, small subproblems.

CHAPTER 4

CLUSTER TREE DATA STRUCTURE

When performing inference with factor elimination, one typically attempts to select an elimination tree to minimize its associated width. However, such an elimination ordering may not be optimal for repeated inference tasks. For example, an HMM typically used for sequence analysis yields a chain-structured factor graph as shown in Figure 4.1a. The obvious elimination tree for this graph is also chain-structured (Figure 4.1b). While this elimination tree is optimal for a single computation, suppose that we now modify the leaf factor f_1. Then, recomputing the marginal for the leaf factor f_7 requires time that is linear in the size in the model, even though only a single factor has changed. However, if we use the *balanced* elimination tree shown in Figure 4.1c, we can compute the marginalization for f_7 in time that is logarithmic in the size of the model. While the latter elimination tree increases the width by one (increasing the dependence on d), for fixed d and as n grows large we can achieve a significant speedup over the unbalanced ordering if we wish to make changes to the model.

In this section we present an algorithm that generates a logarithmic-depth representation of a given elimination tree. Our primary technique, which we call *deferred factor elimination*, generalizes factor elimination so that it can be applied to non-leaf nodes in the input elimination tree. Deferred factor elimination introduces ambiguity, however, since we cannot determine the "direction" that a factor should be propagated until one of its neighbors is also eliminated. We refer to the local information resulting from each deferred factor elimination as a *cluster function* (or, more succinctly, as a *cluster*), and store this information along with the balanced elimination tree. We use the resulting data structure, which we call a *cluster tree*, to perform marginalization and efficiently manage structural and parameter updates.

For our algorithm, we assume that the user provides both an input factor graph G and an associated elimination tree T. While the elimination tree is traditionally computed from

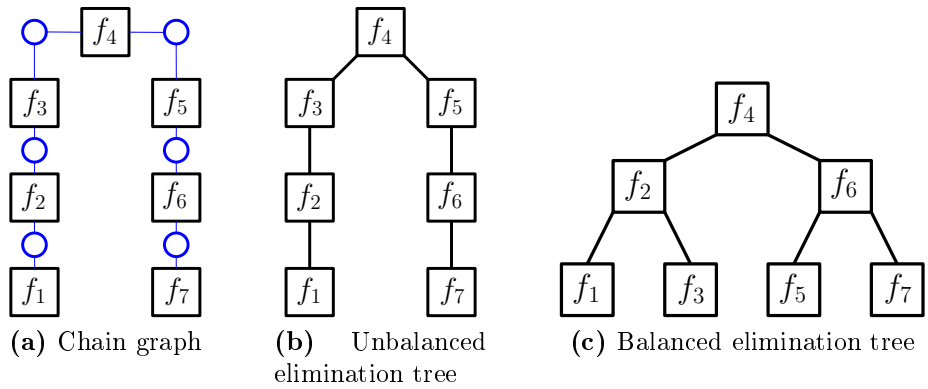


Figure 4.1: Balanced and unbalanced elimination trees. For the chain factor graph in (a), the elimination tree in (b) has width 1 but requires $O(n)$ steps to propagate information from leaves to the root. The balanced elimination tree in (c), for the same factor graph, has width 2 but takes only $O(\log n)$ steps to propagate information from a leaf to the root, since f_3 and f_5 are eliminated earlier. If f_1 is modified, then using a balanced elimination tree, we only need to update $O(\log n)$ elimination steps, while an unbalanced tree requires potentially $O(n)$ updates.

an input model, in an adaptive setting it may be desirable to change the elimination tree to take advantage of changes made to the factors (see Figure 5.1 for an example). Furthermore, domain-specific knowledge of the changes being made to the model may also inform how the elimination tree should be chosen and updated. Thus, in the remainder of the paper we separate the discussion of updates applied to the input model from updates that are applied to the input elimination tree. As we will see in Chapter 5, the former prove to be relatively easy to deal with, while the latter require a reorganization of the cluster tree data structure.

4.1 Deferred factor elimination and cluster functions

Consider the elimination of a degree-two factor f_j, with neighbors f_i and f_k in the given elimination tree. We can perform a partial marginalization for f_j to obtain λ_k, but cannot yet choose whether to update f_i or f_k – whichever is eliminated first will need λ_k for its computation. To address this, we define *deferred factor elimination*, which removes the factor f_j and saves the partial marginalization λ_j as a *cluster*, leaving the propagation step

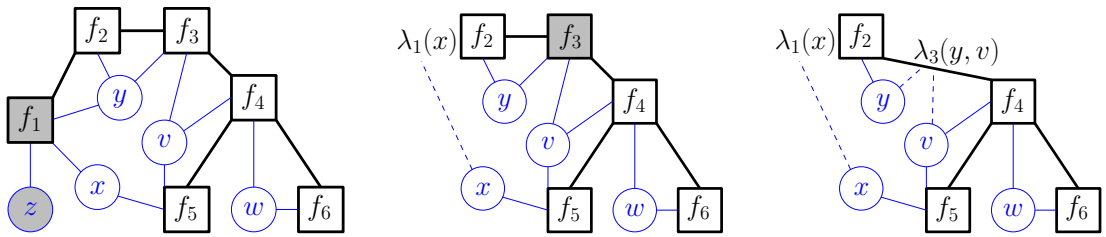
```
DeferredFactorElimination(G, T, f_j)

Compute cluster λ_j using Equation (3)
if f_j is a leaf in elimination tree T
    Let f_k be f_j's unique neighbor in T
    Attach λ_j to f_k in T
end if
if f_j is a degree-2 node in T
    Let f_i and f_k be f_j's neighbors in T
    Create a new edge (f_i, f_k) in T
    Attach λ_j to the newly created edge (f_i, f_k)
endif
Remove factor f_j from factor graph G and T
for each variable x_i that is connected to only f_j in G
    Remove x_i from G
endfor
```

Figure 4.2: Deferred factor elimination. In addition to eliminating leaves, deferred factor elimination also eliminates degree-two nodes. This operation can be simultaneously applied to an independent set of leaves and degree two nodes.

to be decided at a later time. In this section, we show how deferred factor elimination can be performed on the elimination tree, and how the intermediate cluster information can be saved and also used to efficiently compute marginals.

For convenience, we will segregate the process of deferred factor elimination on the input model into rounds. In a particular round t ($1 \leq t \leq n$), we begin with a factor graph G_t and an elimination tree T_t, and after performing some set of deferred factor eliminations, we obtain a resulting factor graph G_{t+1} and elimination tree T_{t+1} for the next round. For the first round, we let $G_1 = G$ and $T_1 = T$. Note that since each factor is eliminated exactly once, the number of total rounds depends on the number of the factors eliminated in each round.

To construct T_{t+1} from T_t, we modify the elimination tree as follows. When we eliminate a degree-1 (leaf) factor f_j, we attach λ_j to the neighbor vertex f_k. When a degree-2 factor f_j is removed, we attach λ_j to a newly created edge (f_i, f_k) where f_i and f_k are f_j's neighbors in elimination tree T. We define $\mathcal{C}_T(f_j)$ to be the set of clusters that are attached either directly to f_j or to an edge incident to f_j. In the factor graph G_{t+1}, we remove all $\lambda_k \in \mathcal{C}_{T_t}(f_j)$ and



(a) T_1 and G_1 **(b)** T_2 and G_2 **(c)** T_3 and G_3

Figure 4.3: Deferred factor elimination. (a) An elimination tree T_1 (in black), with variable dependencies shown in blue for reference. To eliminate a leaf node f_1, we sum out variables that are not attached to any other factors (shaded), resulting in the cluster function λ_1 and new elimination tree T_2 in (b).To eliminate a degree-2 node f_3, we replace it with λ_3 attached to the edge (f_2, f_4), giving tree T_3 shown in (c).

variables $\mathcal{V}_j \subset X$ that do not depend on any factors other than f_j or $\lambda_k \in \mathcal{C}_{T_t}(f_j)$. Finally, we replace f_j with λ_j, given by

$$\lambda_j = \sum_{\mathcal{V}_j} f_j \prod_{\lambda_k \in \mathcal{C}_{T_t}(f_j)} \lambda_k. \tag{4.1}$$

The cluster λ_j is referred to as a *root cluster* if $\deg_{T_t}(f_j) = 0$, a *degree-1 cluster* if $\deg_{T_t}(f_j) = 1$, and a *degree-2 cluster* if $\deg_{T_t}(f_j) = 2$. Figure 4.3 illustrates the creation of degree-1 and degree-2 clusters, and the associated changes to the elimination tree and factor graph. We first eliminate f_1 by replacing it with degree-1 cluster $\lambda_1(x) = \sum_z f_1(x)$. Cluster λ_1 is attached to factor f_2 and the set of clusters around f_2 is $\mathcal{C}_{T_2}(f_2) = \{\lambda_1, \lambda_3\}$. We then eliminate a degree-2 factor f_3 by replacing it with degree-2 cluster $\lambda_3(y, v) = f_3(y, v)$. This connects f_2 to f_4 in the elimination tree, and places λ_3 on the newly created edge.

We note that the correctness of deferred factor elimination follows from the correctness of standard factor elimination. To perform marginalization for any particular variable, we can simply instantiate a series of propagations, at each step using a cluster function that has already been computed in one of the aforementioned rounds.

To establish the overall running time of deferred factor elimination we first explain how the clusters we compute can be interpreted in the tree-decomposition framework. Recall

that in Section 3.2.2, we established an equivalence between clusters and messages in the tree-decomposition in the case where only leaf factors in the elimination tree are eliminated. We can generalize this relationship to the case where degree-2 factors are also eliminated. As discussed earlier in Section 3.2.2, the equivalent tree-decomposition $(\chi, \psi, \mathcal{D})$ of an elimination tree $T = (F, E)$ consists of a tree $\mathcal{D}$ on hyper-nodes $\chi = \{\chi_1, \ldots, \chi_m\}$ with the same adjacency relationship with the factors $\{f_1, \ldots, f_m\}$ in T.

A degree-1 cluster λ_j produced after eliminating a leaf f_j factor in T is a partial marginalization of the factors on a sub-tree of T. Let f_k be f_j's unique neighbor in the elimination tree when it is eliminated. This implies $\lambda_j = \mu_{\chi_t \to \chi_k}$ for some t as previously discussed in Section 3.2.2. Note that the index t may not equal j, since there may be a cluster attached to the edge (f_j, f_k) (for example in Figure 4.3, $\lambda_1(x) = \mu_{\chi_1 \to \chi_2}(x)$).

A degree-2 cluster λ_j produced after eliminating a degree-2 factor f_j in T is a partial marginalization of the factors in a connected subgraph $S \subset T$ such that S and $T \setminus S$ are connected by exactly two edges. Let (f_i, f_c) and (f_d, f_k) be these edges, where f_c and f_d belong to S and f_i and f_k are outside of S (we will show how these "boundary" edges can be efficiently computed in Section 4.2). We interpret λ_j as an intermediary function that enables us to compute an outgoing message $\mu_{\chi_d \to \chi_k}$ by using only λ_j and the incoming message $\mu_{\chi_j \to \chi_c}$, i.e. $\mu_{\chi_d \to \chi_k} = \sum_{\chi_k \setminus \chi_j} \lambda_i \mu_{\chi_j \to \chi_c}$. These intermediate functions are in fact the mechanism that allows us avoid long sequences of message passing. For example in Figure 4.3, λ_3 can be used to compute the message $\mu_{\chi_3 \to \chi_4}$ using only $\mu_{\chi_2 \to \chi_3}$, i.e. $\mu_{\chi_3 \to \chi_4}(x, v) = \sum_y \mu_{\chi_2 \to \chi_3}(x, y) \lambda_3(y, v)$.

Finally, we note that we have a single root cluster that is just a marginalization of all of the factors in the factor graph. Using the relationships established above between cluster functions and messages in a tree decomposition, we give the running time of deferred factor elimination on a given elimination tree and input factor graph.

Lemma 4.1.1 *For an elimination tree with width w, the elimination of leaf factors takes*

$\Theta(d^{2w})$ time and produces a cluster of size $\Theta(d^w)$, where d is the domain size of the variables in the input factor graph. The elimination of degree-2 vertices takes $\Theta(d^{3w})$ time and produces a cluster of size $\Theta(d^{2w})$.

Proof: Each degree-1 cluster has size $O(d^w)$ because it is equal to a sum-product message in the equivalent tree-decomposition. For a degree-2 vertex f_j, the cluster λ_j can be interpreted as an intermediary function that enables us to compute the outgoing messages $\mu_{\chi_c \to \chi_i}$ and $\mu_{\chi_d \to \chi_k}$ using the incoming messages $\mu_{\chi_k \to \chi_d}$ and $\mu_{\chi_i \to \chi_c}$ for some χ_c, χ_d, χ_i and χ_k where f_i and f_k are neighbors of f_j in the elimination tree during its elimination. The set of variables involved in these computations is $(\chi_i \cap \chi_c) \cup (\chi_k \cap \chi_d)$ which is bounded by $2w$. Hence, the cluster f_i that computes the partial marginalization of the factors that are between (f_d, f_k) and (f_i, f_c) has size $O(d^{2w})$. Moreover, these bounds are achieved if $\chi_i \cap \chi_c$ and $\chi_k \cap \chi_k$ are disjoint and each has w variables.

We now establish the running times of calculating cluster functions, by bounding the number of variables involved in computing a cluster. We first show that when a leaf node f_j is eliminated, the set of variables involved in the computation is $\chi_j \cup \chi_k$ where f_k is f_j's neighbor. For all the degree-1 clusters of f_j, their argument set is a subset of χ_j, so the product in Equation (4.1) can be computed in $O(d^w)$ time. There can be a cluster λ_c on the edge (f_j, f_k) whose argument set has to be subset of $\chi_j \cup \chi_k$. If there is such a cluster, the cost of computing the product in Equation (4.1) becomes $O(d^{2w})$. This bound is achieved when there is a degree-2 cluster and χ_j and χ_k are disjoint.

When a degree-2 factor f_j is eliminated, the set of variables involved in the computation is $\chi_i \cup \chi_j \cup \chi_k$ where f_i and f_k are neighbors of f_j. As shown above, the argument set of degree-1 clusters is a subset of χ_j. This cluster can have degree-2 clusters on edges (f_i, f_j) and (f_j, f_k), and in this case, computation of a degree-2 cluster takes $O(d^{3w})$ time. This upper bound is achieved when the sets χ_i, χ_j and χ_k are disjoint.

We note that in the above discussion we assumed that the number of operands in Equa-

```
BuildClusterTree(G, T)

G_0 := G, T_0 := T
Initialize H as an empty rooted tree
for round t = 1 up to k
    G_t := G_{t-1}, T_t := T_{t-1}
    S := A maximal independent set of leaves and degree two nodes in T_t
    for each factor f_j in S
        call DeferredFactorElimination(G_t, T_t, f_j)
        for each cluster λ_i that is used to compute λ_j
            Add edge (λ_i, λ_j) in H where λ_j is the parent.
        endfor
        for each variable x_i eliminated along with f_j
            Add edge (x_i, λ_j) in H where λ_j is the parent
        endfor
    endfor
endfor
return H as the cluster tree
```

Figure 4.4: Hierarchical clustering. Using deferred factor elimination, we can construct a balanced cluster tree data structure that can be used for subsequent marginal queries.

tion (4.1) is bounded, that is, for any factor f, $|\mathcal{C}_T(f)| = O(1)$. This assumption is valid because for any given elimination tree, we can construct an equivalent elimination tree with degree 3 by adding dummy factors. For example, suppose the input elimination tree has degree $n-1$ (i.e., it is star-shaped); then Equation (4.1) has n multiplication operands hence requires $O(nd^w)$ time to compute. By adding dummy factors in the shape of a complete binary tree between the center factor and the leaf factors, we can bring the complexity of computing Equation (4.1) down to $O(d^w)$ for each factor. ■

4.2 Constructing a balanced cluster tree

In this section, we show how performing deferred factor elimination in rounds can be used to create a data structure we call a *cluster tree*. As variables and factors are eliminated through deferred factor elimination, we build the cluster tree using the dependency relationships among clusters (see Figure 4.4). The cluster tree can then be used to compute marginals

efficiently, and as we will see, it can also be used to efficiently update the original factor graph or elimination tree.

For a factor graph $G = (X, F)$ and an elimination tree T, a cluster tree $\mathcal{H} = (X \cup C, E)$ is a rooted tree on variables and clusters $X \cup C$ where C is the set of clusters. The edges E represent the dependency relationships among the quantities computed while performing deferred factor elimination. When a factor f_j is eliminated, cluster λ_j is produced by Equation (4.1). All the variables $\mathcal{V}_j$ and clusters $\mathcal{C}(f_j)$ removed in this computation become λ_j's children. For a cluster λ_j, the *boundary* ∂_j is the set of edges in T that separates the collection of factors that is contracted into λ_j from the rest of the factors.

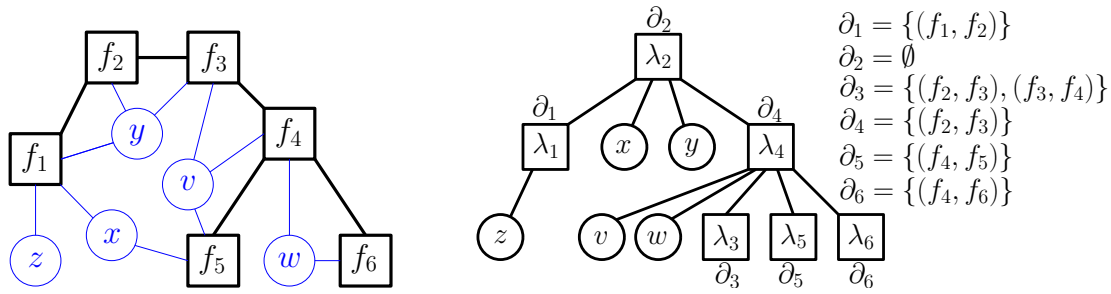
In Equation (4.1), we gave a recursive formula to compute λ_j in terms of its children in the cluster tree. In order to use the cluster tree in our computations, we need to derive a similar recursive formula for the boundary ∂_j for each cluster λ_j. Let clusters $\lambda_1, \lambda_2, \ldots, \lambda_k$ and variables $x_1, x_2, \ldots, x_t$ be λ_j's children in the cluster tree. Let $E(f_j)$ be the set of edges incident to f_j in T. Then the boundary of λ_j can be computed by

$$\partial_j = E(f_j) \triangle \partial_1 \triangle \partial_2 \triangle \ldots \triangle \partial_k$$

where ∂_i is the boundary of cluster λ_i and $\triangle$ is the symmetric set difference operator. An example cluster tree, along with explicitly computed boundaries, is given in Figure 4.5b. For example the boundary of the cluster λ_4 is computed by $\partial_4 = E(f_4) \triangle \partial_3 \triangle \partial_5 \triangle \partial_6$ where $E(f_4) = \{(f_2, f_4), (f_4, f_5), (f_4, f_6)\}$.

Theorem 4.2.1 *Let $G = (X, F)$ be a factor graph with n nodes and T be an elimination tree on G with width w. Constructing a cluster tree takes $\Theta(d^{3w} \cdot n)$ time.*

Proof: During the construction of the cluster tree, every factor is eliminated once. By Lemma 4.1.1, each such elimination takes $O(d^{3w})$ time. ■



(a) Factor Graph G **(b)** Cluster Tree $\mathcal{H}$

Figure 4.5: Cluster Tree Construction. To obtain the cluster tree in (b), eliminations are performed in the factor graph G (a) in the following order: f_1, f_3, f_5 and f_6 in round 1, f_4 in round 2 and f_2 in round 3. The cluster-tree (b) representing this elimination is annotated by boundaries.

For our purposes it is desirable to perform deferred factor elimination so that we obtain a cluster tree with logarithmic depth. We call this process *hierarchical clustering* and define it as follows. We start with $T_1 = T$ and at each round i we identify a set K of degree-1 or -2 factors in T_i and apply deferred factor elimination to this independent set of factors to construct T_{i+1}. This procedure ends once we eliminate the last factor, say f_r. We make λ_r the root of the cluster tree. At each round, the set $K \subset F$ is chosen to be a maximal independent set, that is, for $f_i, f_j \in K$, $f_i \not\sim f_j$ in T, and no other factor f_k can be added to K without violating independence. The sequence of elimination trees created during the hiearchical clustering process will prove to be useful in Chapter 5, when we show how to perform structural updates to the elimination tree. As an example, a factor graph G, along with its associated elimination tree $T = T_1$, is given in Figure 4.5a. In round 1, we eliminate a maximal independent set $\{f_1, f_3, f_5, f_6\}$ and obtain T_2. In round 2 we eliminate f_4, and finally in round 3 we eliminate f_2. This gives us the cluster tree shown in Figure 4.5b.

As we show with the following lemma, the cluster tree that results from hiearchical clustering has logarithmic depth. We will make use of this property throughout the remainder of the paper to establish the running times for updating and computing marginals and MAP configurations.

```
QueryMarginal(H, x_i)
Let x_i, λ_1, ..., λ_k be the path from x_i to the root λ_k of cluster tree H
for j = k down to 1
    Let f_j be the factor associated with cluster λ_j
    Compute downward marginalization function M_{f_j} using Equation (4)
endfor
Compute the marginal at x_i using Equation (5)
```

Figure 4.6: Performing Marginalization with a Cluster Tree. Computing any particular marginal in the input factor graph corresponds to a root-to-leaf path in the cluster tree.

Lemma 4.2.2 *For any factor graph $G = (X, F)$ with n nodes and any elimination tree T, the cluster tree obtained by hierarchical clustering has depth $O(\log n)$.*

Proof: Let the elimination tree $T = (F, E)$ have a leaves, b degree-2 nodes and c degree-3 or more nodes, i.e. $m = a + b + c$ where m is the number of factors. Using the fact that the sum of the degrees of the vertices is twice the number of edges, we get $2\,|E| \geq a + 2b + 3c$. Since a tree with m vertices have $m - 1$ edges, we get $2a + b - 2 \geq m$. On the other hand, a maximal independent set of degree-1 and degree-2 vertices must have size at least $a - 1 + (b - a)/3 \geq m/3$, since we can eliminate at least a third of the degree-2 vertices that are not adjacent to leaves. Therefore at each round, we eliminate at least a third of the vertices, which in turn guarantees that the depth of the cluster tree is $O(\log n)$. ■

4.3 Computing marginals

Once a balanced cluster tree $\mathcal{H}$ has been constructed from the input factor graph and elimination tree, as in standard approaches we can compute the marginal distribution of any variable x_i by propagating information (i.e., partial marginalizations) through the cluster tree. For any fixed variable x_i, let $\lambda_1, \lambda_2, \ldots, \lambda_k$ be the sequence from x_i to the root λ_k in the cluster tree $\mathcal{H}$. We now describe how to compute the marginal for x_i (see Figure 4.6 for

pseudocode). For each factor f_j, let ∂_j contain neighbors f_a and f_b of f_j (i.e., neighboring factors at the time f_j is eliminated). This information can be obtained easily, since f_a and f_b are ancestors of f_j in the cluster tree, that is, $f_a, f_b \in \{f_{j+1}, f_{j+2}, \ldots, f_k\}$. For convenience we state our formulas as if there are two neighbors in the boundary; in the case of degree-1 clusters, terms associated with one of the neighbors, say f_b, can be ignored in the statements below. First, we compute a downward pass of marginalization functions from λ_k to λ_1 given by

$$M_{f_j} = \sum_{Y \setminus X_{\lambda_j}} f_j M_{f_a} M_{f_b} \prod_{f \in \mathcal{C}_j \setminus \{f_{j-1}\}} f, \tag{4.2}$$

where Y is the set of variables that appear in the summands and X_{λ_j} is the set of variables that cluster λ_j depends on. Therefore each marginalization function M_j from parent λ_j is computed using only information in the path above λ_j. Then, the marginal for variable x_i is

$$g^i(x_i) = \sum_{Y \setminus \{x_i\}} M_{f_1} \prod_{f \in \mathcal{C}_1} f \tag{4.3}$$

where Y is the set of variables that appear in the summands. Combining this approach with Lemmas 4.1.1 and 4.2.2, we have the following theorem.

Theorem 4.3.1 *Consider a factor graph G with n nodes and let T be an elimination tree with width w. Then, Equation (4.3) holds for any variable x_i and can be computed in $O(d^{2w} \log n)$ time.*

Proof: The correctness of Equation (4.3) follows when each marginalization function M_{f_j} is viewed as a sum-product message in the equivalent tree-decomposition. To prove the latter, we will show that for $\partial_j = \{(f_c, f_a), (f_d, f_b)\}$, M_{f_a} and M_{f_b} are equal to the tree-decomposition messages $\mu_{\chi_a \to \chi_c}$ and $\mu_{\chi_b \to \chi_d}$, respectively. This can be proven inductively starting with M_{f_k}. First, note that the base case holds trivially. Then, using the inductive hypothesis, we assume that $M_{f_a} = \mu_{\chi_a \to \chi_c}$ and $M_{f_b} = \mu_{\chi_b \to \chi_d}$. Now, there has to be a

descendant λ_ℓ of λ_j such that $(f_e, f_j) \in \partial_\ell$. By multiplying with the degree-2 clusters in $\mathcal{C}_j \setminus \{f_{j-1}\}$, we can convert the messages $\mu_{\chi_a \to \chi_c}$ and $\mu_{\chi_b \to \chi_d}$ to the messages into f_j. Applying Equation (3.2) then gives $M_{f_j} = \mu_{\chi_j \to \chi_e}$ as desired.

For the running time, we observe that each message computation is essentially the same procedure as eliminating a leaf factor, therefore each message has size $O(d^w)$ and takes $O(d^{2w})$ time to compute by Lemma 4.1.1. ■

We note that it is also possible to speed-up successive marginal queries by caching the downward marginalization functions in Equation (4.2). For example, if we query all variables as described above, we compute $O(n \log n)$ many downward marginalization messages. However, by caching the downward marignalization functions in the cluster tree, we can compute all marginals in $O(d^{2w} \cdot n)$ time, which is optimal given the elimination ordering. As we will see in Section 5.1, the balanced nature of the cluster tree allows us to perform batch operations efficiently. In particular, for marginal computation, using the caching strategy above, any set of ℓ marginals can be computed in $O(d^{2w} \ell \log(n/\ell))$ time.

CHAPTER 5

EFFICIENT UPDATES TO CLUSTER TREES

The preceding chapter described the process of constructing a balanced cluster tree elimination ordering from a given elimination tree, and how to use the resulting cluster tree to compute marginal distributions. However, the primary advantage of a balanced ordering lies in its ability to adapt to changes and incorporate updates to the model. In this chapter, we describe how to efficiently update the cluster tree data structure after changes are made to the input factor graph or elimination tree.

We divide our update process into two algorithmic components. We first describe how to make changes to the factors, whether changing the parameters of the factor or its arguments (and thus the structure of the factor graph), but leaving the original elimination tree (and thus the cluster tree) fixed. We then describe how to make changes to the elimination tree and efficiently update the cluster tree. In practice these two operations may be combined; for example when modifying a tree-structured graph such that it remains a tree we are likely to change the elimination tree to reflect the new structure. Similarly, for a general input factor graph we may also wish to change the elimination tree upon changes to factors. Figure 5.1 illustrates such an example, in which changing a dependency in the factor graph makes it possible to reduce the width of the elimination tree.

5.1 Updating factors with a fixed elimination tree

For a fixed elimination tree, suppose that we change the parameters of a factor f_j (but not its arguments), and consider the new cluster tree created for the resulting graph. As suggested in the discussion in Chapter 4, the first change in the clustering process occurs when computing λ_j; a change to λ_j changes its parent, and so on upwards to the root. Thus, the number of affected functions that need to be recalculated is at most the depth of the

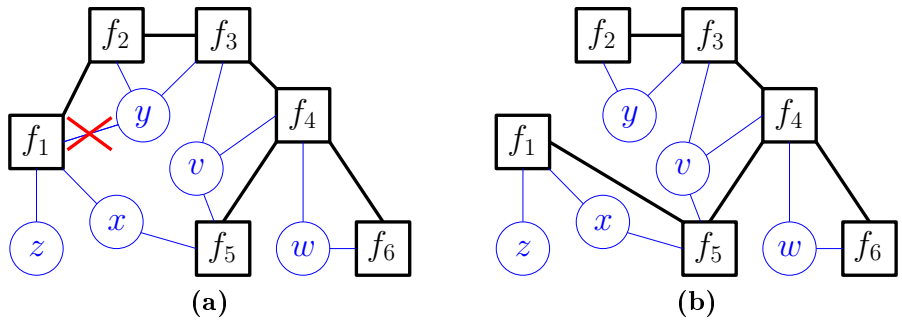


Figure 5.1: Modifying the Elimination Tree. If the factor graph in (a) is modified by removing the edge (y, f_1), we can reduce the width of the elimination tree (from 3 to 2) by replacing the edge (f_1, f_2) by (f_1, f_5).

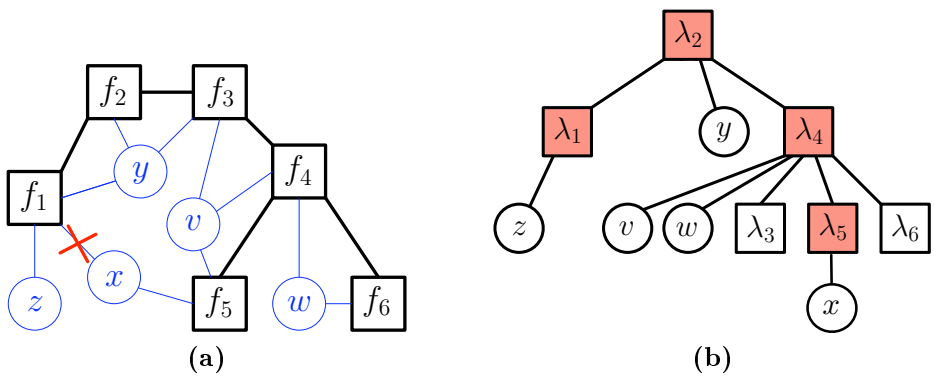


Figure 5.2: Modifying the arguments of factors. If the factor graph in (a) is modified by removing the edge (x, f_1), we update two paths in the cluster tree, as shown in (b), from both x and λ_1 to the root. The position in which x is eliminated is found by bottom-up traversing of the factors adjacent to x.

cluster tree. Since the cluster tree is of depth $O(\log n)$ by Lemma 4.2.2, and each operation takes at most $O(d^{3w})$, the total recomputation is at most $O(d^{3w} \log n)$.

If we change the structure of graph G by modifying the arguments of a factor f_j by adding or removing some variable x_i, then the point at which x_i is removed from the factor graph may also change. Since x_i is eliminated (eg., summed out) once every factor that depends on it has been eliminated, adding an edge may postpone elimination, while removing an edge may lead to an earlier elimination. To update the cluster tree as a result of this change, we must update all clusters affected by the change to f_j, and we must also identify and update the clusters affected by earlier, or later, removal of x_i from the factor graph. In both edge

addition and removal, we can update clusters from λ_j to the root in $O(d^{3w} \log n)$ time.

We describe how to identify the new elimination point for x_i in $O(\log n)$ time. Observe that the original cluster λ_k at which x_i is eliminated is the topmost cluster in the cluster tree with the property that either f_k, the associated factor, depends on x_i, or λ_k has two children clusters that both depend on x_i. The procedure to find the new point of elimination differs for edge insertion and edge removal. First, suppose we add edge (x_i, f_j) to the factor graph. We must traverse upward in the cluster tree until we find the cluster satisfying the above condition. For edge removal, suppose that we remove the dependency (x_i, f_j). Then, x_i can only need to be removed earlier in the clustering process, and so we traverse downwards from the cluster where x_i was originally eliminated. At any cluster λ_k during the traversal, if the above condition is not satisfied then λ_k must have one or no children clusters that depend on x_i. If λ_k has a single child that depends on x_i, we continue traversing in that direction. If λ_k has no children that depend on x_i, then we continue traversing towards λ_j. Note that this latter case occurs only when the paths of x_i and λ_j to the root overlap, and thus is always possible to traverse toward λ_j.

Once we have identified the new cluster at which x_i is eliminated, we can recalculate cluster functions upwards in $O(d^{3w} \log n)$ time. Therefore the total cost of performing an edge insertion or removal $O(d^{3w} \log n)$. Figure 5.2 illustrates how the cluster tree is updated after deleting an edge in a factor graph keeping the elimination tree fixed. After deleting (x, f_1) we first update the clusters upwards starting from λ_1. Then traverse downwards to find the point at which x_i is eliminated, which is λ_5 because f_5 depends on x. Finally, we update λ_5 and its ancestors.

We can also extend the above arguments to handle multiple, simultaneous updates to the factor graph. Suppose that we make ℓ changes to the model, either to the definition of a factor or its dependencies. Each change results in a set of affected nodes that must be recomputed; these nodes are the ancestors of the changed factor, and thus form a path

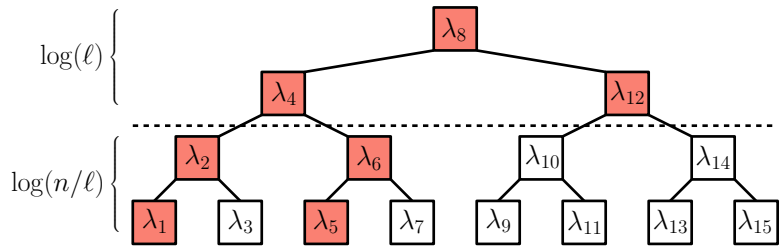


Figure 5.3: Batch updates. After modifying $\ell = 3$ factors, f_1, f_5 and f_{12}, we update the corresponding clusters and their ancestors in a bottom-up fashion. The total number of nodes visited is $O(\ell \log(\frac{n}{\ell}) + 2^{\log(\ell)}) = O(\ell \log(\frac{n}{\ell}))$.

upwards through the cluster tree. This situation is illustrated in Figure 5.3. Now, we count the number of affected nodes by grouping them into two sets. If our cluster tree has branching factor b, level $\log_b(\ell)$ has ℓ nodes; above this point, paths *must* merge, and all clusters may need to be recalculated. Below level $\log_b(\ell)$, each path may be separate. Thus the total number of affected clusters is $\ell + \ell \log_b(n/\ell)$.

Note that for edge modifications, we must also address how to find new elimination points efficiently. As stated earlier, any elimination point λ_k for x_i satisfies the condition that it is the topmost cluster in the cluster tree with the property that either f_k depends on x_i, or λ_k has two children clusters that both depend on x_i. As we update the clusters in batch, we can determine the variables for which the above condition is not satisfied until we reach the root cluster. In addition, we also mark the bottommost clusters at which the above condition is not satisfied. Starting from these marked clusters, we search downwards level-by-level until we find the new elimination points. At each step λ_k, we check if there is a variable x_i such that $x_i \not\sim f_j$ and only one child cluster of λ_k depends on x_i. If there is not, we stop the search; if there is, we continue searching towards those clusters. Since each step takes $O(w)$ time, the total time to find all new elimination points is $O(w\ell \log(n/\ell))$. We then update the clusters upwards starting from the new elimination points until the root, which takes $O(d^{3w}\ell \log(n/\ell))$ time.

Combining the arguments above, we have the following theorem.

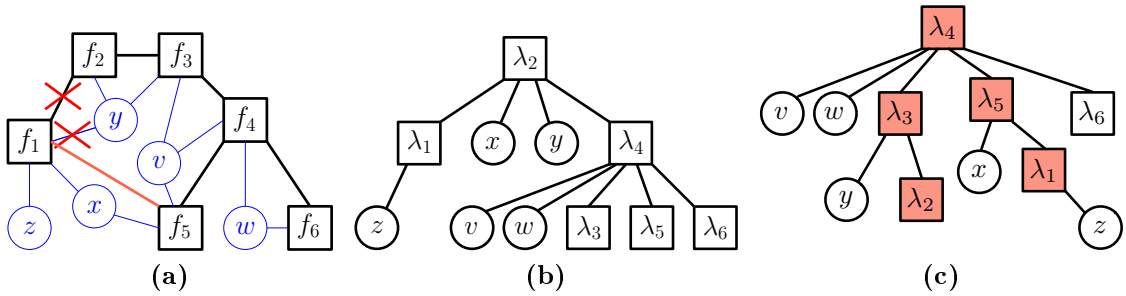


Figure 5.4: Updating the elimination tree. Suppose we modify the input factor graph by removing (y, f_1) from the factor graph and replacing (f_1, f_2) by (f_1, f_5) in the elimination tree as shown in (a). The original cluster tree (b) must be changed reflect these changes. We must revisit the decisions made during the hierarchical clustering for the affected factors (in red).

Theorem 5.1.1 *Let $G = (X, F)$ be a factor graph with n nodes and $\mathcal{H}$ be the cluster tree obtained using an elimination tree T with width w. Suppose that we make ℓ changes to the model, each consisting of either adding or removing an edge or modifying the parameters of some factor, while holding T fixed. Then, we can recompute the cluster tree $\mathcal{H}'$ in $O(d^{3w}\ell \log(n/\ell))$ time.*

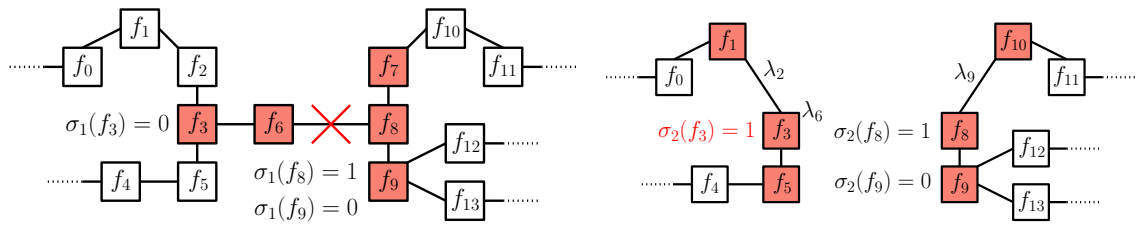
5.2 Structural changes to the elimination tree

Many changes to the graphical model will be accompanied by some change to the desired elimination ordering. For example, changing the arguments of a factor may suggest some more efficient ordering that we may wish to exploit. However, changing the input elimination order also requires modifying the cluster tree constructed from it. Figure 5.4 shows such a scenario, where removing a dependency suggests an improved elimination tree. In this section we prove that it is possible to efficiently reorganize the cluster tree after a change to the elimination tree.

As in the previous section, we wish to recompute only those nodes in the cluster tree whose values have been affected by the update. In particular we construct the new cluster tree by stepping through the creation of the original sequence $T_1, T_2, \ldots$, marking some nodes

as *affected* if we need to revisit the deferred elimination decision we made in constructing the cluster tree, and leaving the rest unchanged. We first describe the algorithm itself, then prove the required properties: that the original clustering remains valid outside the affected set; that after re-clustering the affected set, our clustering remains a valid maximal independent set and is thus consistent with the theorems in Chapter 4; and finally that the total affected set is again only of size $O(\log n)$. Since the elimination tree can be arbitrarily modified by performing edge deletions and insertions successively, for ease of exposition we first focus on how the cluster tree can be efficiently updated when a single edge in the elimination tree is inserted or deleted. For the remainder of the section, we assume that the hierarchical clustering process produced intermediate trees $(T_1, T_2, \ldots, T_k)$ and that (f_i, f_j) is the edge being inserted or deleted.

Observe that, to update any particular round of the hierarchical clustering, for any factor f_k we must be able to efficiently determine whether its associated cluster must be recomputed due to the insertion or deletion of an edge (f_i, f_j). A trivial way to check this would be to compute a new hiearchical clustering $(T'_1, T'_2, \ldots, T'_l)$ using the changed elimination tree. Then, the cluster λ_k that is generated after eliminating f_k depends only on the set of clusters around f_k at the time of the elimination. If $\mathcal{C}_i(f_k)$ and $\mathcal{C}'_i(f_k)$ are the set of clusters around f_k on T_i and T'_i, respectively, then f_k is affected at round i if the sets $\mathcal{C}_i(f_k)$ and $\mathcal{C}'_i(f_k)$ are different. Note that we consider $\mathcal{C}_i(f_k) = \mathcal{C}_i(f_k)$ if $\lambda_j \in \mathcal{C}_i(f_k) \iff \lambda_j \in \mathcal{C}'_i(f_k)$ and the values of λ_j are identical in both sets. Clearly, this approach is not efficient, but motivates us to (incrementally) track whether or not $\mathcal{C}_i(f_k)$ and $\mathcal{C}'_i(f_k)$ are identical in a more efficient manner. To do this, we define the *degree-status* of the neighbors of f_k, and maintain it as we update the cluster tree. Given two hierarchical clusterings $(T_1 = (F_1, E_1), T_2 = (F_2, E_2), \ldots, T_k = (F_k, \emptyset))$ and $(T_1 = (F'_1, E'_1), T_2 = (F'_2, E'_2), \ldots, T_l = (F'_l, \emptyset))$, we define



(a) Round 1 **(b)** Round 2

Figure 5.5: Affected nodes in the clustering. By rule 2 for marking factors as affected, eliminating f_6 in the first round makes $\sigma_2(f_3) = 1$, thereby making f_1 and f_5 affected. In contrast, since $\sigma_2(f_9) = 0$, f_{12} and f_{13} are not marked as affected. By rule 1, eliminating f_7 in the first round makes f_{10} affected.

the degree-status $\sigma_i(f)$ of a factor f at round i as

$$\sigma_i(f) = \begin{cases} 1 & \text{if} \quad \deg_{T_i}(f) \leq 2 \text{ or } \deg_{T_i'}(f) \leq 2 \text{ or } f \notin F_i \cap F_i' \\ 0 & \text{if} \quad \deg_{T_i}(f) \geq 3 \text{ and } \deg_{T_i'}(f) \geq 3 \end{cases}$$

The degree status tells us whether f is a candidate for elimination in either the previous or the new cluster tree.

At a high level, we step through the original clustering, marking factors as affected according to their degree-status. For a factor f_j, if $\sigma_i(f_j) = 1$, then f_j is either eliminated or a candidate for elimination at round i in one or both of the previous and new hierarchical clusterings. Since we must recompute clusters for affected factors, if we mark f_j as affected, then its unaffected neighbors should also be marked as affected in the next round. This approach conservatively tracks how affectedness "spreads" from one round to the next; we may mark factors as affected unnecessarily. However, we will be able to show that any round of the new clustering has a constant number of factors for which we must recompute clusters.

We now describe our algorithm for updating a hierarchical clustering after a change to the elimination tree. We first insert or remove the edge (f_i, f_j) in the original elimination tree and obtain $T_1' = (V_1', E_1')$ where $E_1' = E_1 \cup \{(f_i, f_j)\}$ if the edge is inserted or $E_1' = E_1 \setminus \{(f_i, f_j)\}$ if deleted. For $i = 1, 2, \ldots, l$, the algorithm proceeds by computing the affected set A_i, an

independent set $M_i \subseteq A_i$ of affected factors of degree at most 2 in T'_i, and then eliminating M_i to form T'_{i+1}. We let $A_0 = \{f_i, f_j\}$, $M_0 = \emptyset$ and $T'_0 = T'_1$. For round $i = 1, 2, \ldots l$ we do the following:

- We obtain the new elimination tree $T'_i = (F'_i, E'_i)$ by eliminating the factors in M_{i-1} from T_{i-1} via deferred factor elimination subroutine.
- All affected factors left in T'_i remain affected, namely the set $A_{i-1} \setminus M_{i-1}$. We mark a previously unaffected factor f as affected if

  1. f has an affected neighbor g in T'_{i-1} such that $g \in M_{i-1}$ or
  2. f has an affected neighbor g in T'_i such that $g \in A_{i-1} \setminus M_{i-1}$ with $\sigma_i(g) = 1$.

  Let N_i be the set of factors that are marked in this round according to these two rules, then $A_i = (A_{i-1} \setminus M_{i-1}) \cup N_i$.
- Initialize $M_i = \emptyset$ and greedily add affected factors to M_i starting with the factors that are adjacent to an unaffected factor. Let $f \in A_i$ be an affected factor with an unaffected neighbor $g \in V'_i \setminus A_i$. If g is being eliminated at round i we skip f, otherwise f is included in M_i if $\deg_{T'_i}(f) \leq 2$. We continue traversing the set of affected factors with degree at most two and add as many of them as we can to M_i, subject to the independence condition.

Observe that a factor f in T'_i becomes affected either if an affected neighbor of f is eliminated at round $i-1$ or if f has neighbor that was affected in earlier rounds with degree-status 1 in T'_i. Once a factor becomes affected, it stays affected. For an unaffected factor f at round i, f's neighbors have to be (i) unaffected, (ii) affected with degree-status 0, or (iii) have become affected at round i.

In order to establish that the procedure above correctly updates the hierarchical clustering, we first prove that we are able to correctly identify unaffected factors, and incrementally maintain maximal independent sets.

Lemma 5.2.1 *Given $T = (T_1, T_2, \ldots, T_k)$, let $T' = (T'_1, T'_2, \ldots, T'_l)$ be the updated hierarchical clustering. For any round $i = 1 \ldots l$, let $T'_i = (F'_i, E'_i)$, let $P_i = F'_i \setminus A_i$ be the set of unaffected factors and $R_i = P_i \setminus F'_{i+1}$ be the ones that are eliminated at round i. Then, the following statements hold:*

- *$R_i \cup M_i$ is a maximal independent set among vertices of degree at most two in F'_i.*
- *For any $f \in P_i$, the set of clusters around f and the set of neighbors of f are the same in T_i as in T'_i.*

Proof: For the first claim, we first observe that R_i is an independent since it is contained in M_i. For maximality, assume that $R_i \cup M_i$ is not a maximal independent set among degree ≤ 2 vertices of F'_i. Then there must be a factor f with two neighbors g, h with degrees ≤ 2 and none of which are eliminated at round i. This triplet (f, g, h) can not be entirely in A_i or $F'_i \setminus A_i$, because the sets R_i and M_i are maximal on their domain, namely R_i is a maximal independent set over $F'_i \setminus A_i$ and M_i is a maximal independent set over A_i. On the other hand, the triplet (f, g, h) can not be on the boundary either because the update algorithm eliminates any factor with $\deg_{T'_i} \leq 2$ if it is adjacent to an unaffected factor that is not eliminated at round i. Therefore, $R_i \cup M_i$ is a maximal independent set over degree ≤ 2 vertices of F'_i.

We now prove the first part of the second claim by induction on i. Let $\mathcal{C}_i(f)$ and $\mathcal{C}'_i(f)$ be the set of clusters around f in T_i and T'_i, respectively. The claim is trivially true for $i = 1$ because $\mathcal{C}_i(f) = \mathcal{C}'_i(f) = \emptyset$ for all factors. Assume that $\mathcal{C}_j(f) = \mathcal{C}'_j(f)$ for all unaffected factors at round j where $j = 1, \ldots, i-1$. Since $f \in P_i$ imples that $f \in P_{i-1}$, we have that $\mathcal{C}_{i-1}(f) = \mathcal{C}'_{i-1}(f)$. Since the set of clusters around a factor changes only if any of its neighbors are eliminated, we must prove that if a neighbor of f is eliminated in T_{i-1}, then it must be eliminated in T'_{i-1} and vice versa; additionally we must prove that they also generate the same clusters. Since $f \in P_{i-1}$, the neighbors of f in T'_i can be unaffected, affected with degree-status 0 or newly affected in round i. When an unaffected factor g is

eliminated in T_{i-1}, it is eliminated in T'_i as well, the resulting clusters are identical since $\mathcal{C}_{i-1}(g) = \mathcal{C}'_{i-1}(g)$. So any change to $\mathcal{C}_i(f)$ due to f's unaffected neighbors is replicated in $\mathcal{C}'_i(f)$. On the other hand, by definition we cannot eliminate a factor with degree-status 0, so they do not pose a problem even if they are affected. The last case is a newly affected neighbor g of f in T_{i-1} with $\sigma_{i-1}(g) = 1$. But this case is impossible because, if g is eliminated then we would have marked f as affected in T_i via the first rule, or if g is not eliminated then by the second rule and the fact that $\sigma_i(g) = 1$, we would have marked f as affected in T_i. Therefore $\mathcal{C}_i(f) = \mathcal{C}'_i(f)$ for all unaffected factors. This implies that clusters of unaffected factors are identical and do not have to be recalculated in T'_i.

Let $\mathcal{N}_i(f)$ and $\mathcal{N}'_i(f)$ be the set of neighbors of f in T_i and T'_i, respectively. Proving the second part of the second claim (i.e., $\mathcal{N}_i(f) = \mathcal{N}'_i(f)$) proceeds similarly to that for $\mathcal{C}_i(f) = \mathcal{C}_i(f)$. The only difference is the initial round when $i = 1$. In round 1, the update algorithm marks all the factors that are incident to the added or removed edges as affected, so for all unaffected factors their neighbor set must be identical in T_i and T'_i. ■

Using this lemma, we can now prove the correctness of our method to incrementally update a hierarchical clustering.

Theorem 5.2.2 *Given a valid hierarchical clustering T, let $T' = (T'_1, T'_2, \ldots, T'_l)$ be the updated hierarchical clustering, where $T'_i = (F'_i, E'_i)$. Then, T' is a valid hierarchical clustering, i.e.,*

- *the set $M_i = F'_i \setminus F'_{i+1}$ is a maximal independent set containing vertices of degree at most two, and*
- *T'_{i+1} is obtained from T'_i by applying deferred factor elimination to the factors in M_i.*

Proof: Recall that A_i is the set of affected factors marked and $M'_i \in A_i$ be the independent set chosen by the algorithm. Let $P_i = F'_i \setminus A_i$ be the set of unaffected factors and $R_i = P_i \setminus F'_{i+1}$ be the ones that are eliminated at round i. The fact that M_i is a maximal

independent set follows from Lemma 5.2.1 because $M_i = R_i \cup M'_i$. Since the update algorithm keeps the decisions made for the unaffected factors, the set of eliminated vertices are precisely $M_i = R_i \cup M'_i$ and by Lemma 5.2.1, M_i is a maximal independent set over degree ≤ 2 vertices of T'_i. The update algorithm applies the deferred factor elimination subroutine on the set M'_i, so what remains to be shown is the saved values for R_i are the same as if we eliminate them explicitly. By Lemma 5.2.1, the factors in R_i have the same set of clusters around them in T_i and T'_i, which means that deferred factor elimination procedure will produce the same result in both elimination trees when unaffected factors are eliminated. Therefore, we can reuse the clusters in R_i. ■

Theorem 5.2.2 shows that our update method correctly modifies the cluster tree, and thus marginals can be correctly computed. Note that, by Lemma 4.2.2, we also have that the resulting cluster tree also has logarithmic depth. It remains to show that we can efficiently update the clustering itself. We do this by first establishing a bound on the number of affected nodes in each round.

Lemma 5.2.3 *For $i = 1, 2, \ldots, l$, let A_i be the set of affected nodes computed by our algorithm after inserting or deleting edge (f_k, f_j) in the elimination tree. Then, $|A_i| \leq 12$.*

Proof: First, we observe that the edge (f_k, f_j) defines two connected components, that are either created or merged, in the elimination tree. Since an unaffected node becomes affected only if it is adjacent to an affected factor, the set of affected nodes forms a connected subtree throughout the elimination procedure. For the remained of the proof, we focus on the component associated with f_k, and show that it has at most 6 affected nodes. A similar argument can be applied to the component associated with f_j, thereby proving the lemma.

For round i, let B_i be the set of affected neighbors of f_k with at least one unaffected neighbor and let N_i be the set of newly affected factors. We claim that $|B_i| \leq 2$ and

$|N_i| \leq 2$ at every round i. This can be proven inductively: assume that $|B_i|$ and $|N_i|$ are at most 2 in round $i \geq 0$. Rule 1 for marking a factor affected can make only one newly affected factor at round $i+1$, in which case it is eliminated, and hence $|B_i|$ cannot increase. Rule 2 for marking a factor affected can make two newly affected factors, as shown in the example Figure 5.5. What is left to be shown is that if $|B_i| = 2$, then rule 2 cannot create two newly affected factors and make $|B_i| > 2$. Let $B_i = \{f_a, f_b\}$ and suppose f_a can force two previously unaffected factors affected in the next round. For this to happen, the degree-status of f_a has to be 1 in round $i+1$. However, this cannot because f_a must have at least three neighbors in both T_{i+1} and T'_{i+1}. This is because it has two unaffected neighbors plus an affected neighbor that is eventually connected to another unaffected factor through f_b. Note that Figure 5.5 has $|B_i| = 1$, so we can increase $|B_i|$ by 1.

We have now established the fact that the number of affected nodes can increase at most by 2 in each round, and it remains to be shown that the number of affected nodes is at most 6 in each connected component.

To prove this, we argue that if there are more than 6 affected nodes in the connected component, our algorithm eliminates at least 2 factors. Since affected nodes form a sub-tree that interacts with the rest of the tree on at most 2 factors, what remains to be shown is that in any tree with at least 4 nodes, the size of a maximal independent set over the nodes with degree at most 2 is at least 2. To see this, observe that every tree has two leaves, and if the size of the tree is at least 4, the distance between these two leaves is at least 2 or the tree is star-shaped. In either case, any maximal independent set must include at least 2 nodes, proving the claim. ■

Combining the above arguments, we now conclude that a cluster tree can be efficiently updated if the elimination tree is modified.

Theorem 5.2.4 *Let $G = (X, F)$ be a factor graph with n nodes and $\mathcal{H}$ be the cluster tree*

obtained using an elimination tree T. If we insert or delete a single edge from T, it suffices to re-compute $O(\log n)$ clusters in $\mathcal{H}$ to reflect the changes.

Proof: Since the number of affected factors is constant at each round by Lemma 5.2.3 and the number of rounds is $O(\log n)$ by Lemma 4.2.2, the result follows. ■

We can easily generalize these results to multiple edge insertions and deletions by separately considering each connected component resulting from a modification. As we discussed in Section 5.1, we only need to recalculate $O(\ell \log(n/\ell))$ many clusters where ℓ is the number of modifications to the elimination tree. We can now state the running time efficiency of our update algorithm under multiple changes to the elimination tree.

Theorem 5.2.5 *Let $G = (X, F)$ be a factor graph with n nodes and $\mathcal{H}$ be the cluster tree obtained using an elimination tree T. If we make ℓ edge insertions or deletions in T, we can recompute the new cluster tree in $O(d^{3w}\ell \log(n/\ell))$ time.*

5.3 Experiments

In this section, we evaluate the performance of our approach by comparing the running times for building, querying, and updating the cluster-tree data structure against (from-scratch or complete) inference using the standard sum-product algorithm. For the experiments, we implemented our proposed approach as well as the sum-product algorithms in Python. In our implementation, all algorithms take the elimination tree as input; when it is not possible to compute the optimal elimination tree for a given input, we use a simple greedy method to construct it (the algorithm grows the tree incrementally while minimizing width). We evaluate the practical effectiveness of our proposed approach by considering synthetically generated graphs to compute marginals (Section 5.3.2). These experiments show that adaptive inference can yield significant speedups for reasonably chosen inputs. To further explore

the limits of our approach, we also perform a more detailed analysis in which we compute the speedup achievable by our method for a range tree-width, dimension, and size parameters. This analysis allows us to better interpret how the asymptotic bounds derived in the previous sections fare in practice.

For our experiments, we randomly generate problems consisting of either tree-structured graphs or loopy graphs and measure the running-time for the operations supported by the cluster tree data structure and compare their running-times to that of the sum-product algorithm. Since we perform exact inference, the sum-product algorithm offers an adequate basis for comparison.

5.3.1 Data Generation

For our experiments on synthetically generated data, we randomly generate input instances consisting of either tree-structured graphs or loopy graphs, consisting of n variables, each of which takes on d possible values. For tree-structured graphs, we define how a factor f_i $(1 \leq i < n)$ depends on any particular variable x_j $(1 \leq j < n)$ through the following distribution:

$$\Pr\left\{f_i \text{ depends on } x_j\right\} = \begin{cases} 1 & \text{if} \quad j = i+1 \\ p(1-p)^{i-j} & \text{if} \quad j = 2, \ldots, i \\ 1 - \sum_{s=2}^{i} p(1-p)^{i-s} & \text{if} \quad j = 1 \end{cases}$$

Here, p is a parameter that when set to 1 results in a linear chain. More generally, the parameter p determines how far back a node is connected while growing the random tree. The ith node is expected to connect as far back as the jth node where $j = i - 1/p$, due to the truncated geometric distribution. In our experiments we chose $p = .2$ and $d = 25$ when generating trees.

For loopy graphs, we start with a simple Markov chain, where each factor f_i depends on

variables x_i and x_{i+1}, where $1 \leq i < n$. Then for parameters w and p, we add a cycle to this graph as follows: if i is even and less than $n - 2(w-1)$, with probability p we create a cycle by adding a new factor g_i that depends on x_i and $x_{i+2(w-1)}$. This procedure is guaranteed to produce a random loopy graph whose width along the chain $x_1, \ldots, x_n$ is at most w; to ensure that the induced width is exactly w we then discard any created loopy graph with width strictly less than w. In our experiments, we set $p = (0.2)^{1/(w-1)}$ so that the maximum width is attained by 20% of the nodes in the chain regardless of the width parameter w. We use an elimination tree $T = (F, E)$ that eliminates the variables $x_1, \ldots, x_n$ in order. More specifically, E includes $\{(f_i, f_{i+1}) : i = 1, \ldots, n-1\}$ and any (f_i, g_i) with $2 \leq i \leq n - 2(w-1)$ that is selected by the random procedure above. In our experiments, we varied n between 10 and 50000.

For both tree-structured and loopy factor graphs, we generate the entries of the factors (i.e., the potentials) by sampling a log-normal distribution, i.e., each entry is randomly chosen from e^Z where Z is a Gaussian distribution with zero mean and unit variance.

5.3.2 Measurements

To compare our approach to sum- and max-product algorithms when the underlying models undergo changes, we measure the running-times for build, update, structural update, and query operations. To perform inference with a graphical model that undergoes changes, we start by performing an initial *build* operation that constructs the cluster-tree data structure on the initial model. As the model changes, we reflect these changes to the cluster tree by issuing *update* operations that change the factors, or *structural-update* operations that change the dependencies in the graph (by inserting/deleting edges) accordingly, and retrieve the updated inference results by issuing *query* operations. We are interested in applications where after an initial build, graphical models undergo many small changes over time. Our goal therefore is to reduce the update and query times, at the cost of a slightly slower initial

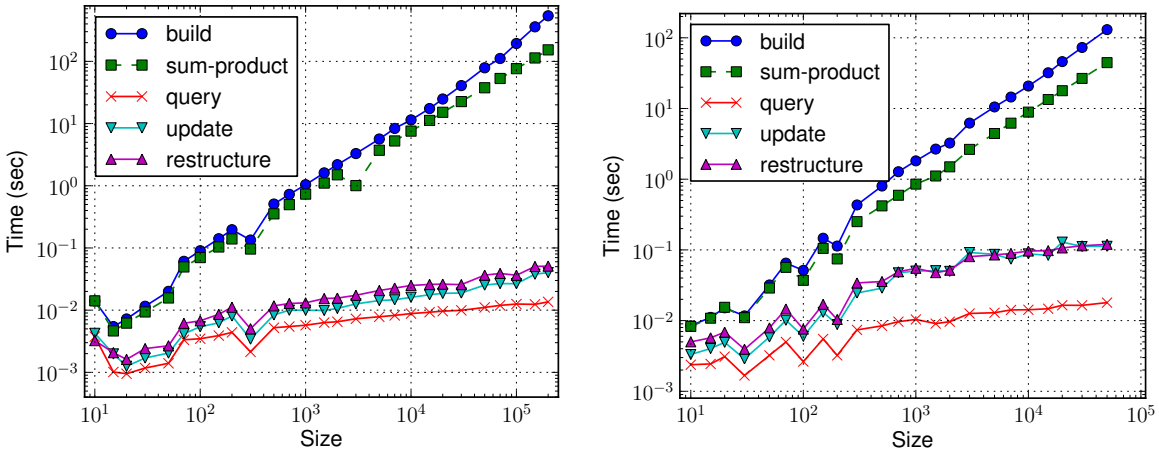
build operation.

Marginal Computations

We consider marginal computation and how we can compute marginals of graphical models that undergo changes using the proposed approach. To this end we measure the running-time for the build, update, structural-update and query operations and compare them to the sum-product algorithm on the junction tree that uses the same elimination tree as our algorithm. We consider graphs with tree-width one (trees) and three and between 10 and $200,000$ nodes. For trees, we set $d = 25$, and for graphs we set $d = 6$.

For the build time, we measure the time to build the cluster tree data structure for graphs generated for various input sizes. The running-time of sum-product is defined as the time to compute messages from leaves to a chosen root node in the factor graph. To compute the average time for a query operation, we take the average time over 100 trials to perform a query for a randomly chosen marginal. To compute the update time, we take the average over 100 trials of the time required to change a modify a randomly chosen factor (to a new factor that is randomly generated). To compute the average time required for a structural updates (i.e, restructure operations), we take the average over 100 trials of the total time required to remove a randomly chosen edge, update the cluster tree, and to add the same edge back to the cluster tree.

Figure 5.6 shows the result of our measurements for tree-structured factor graphs and loopy graphs with tree-width 3. We observe that the running-time for the build operations, which constructs the initial cluster tree, is comparable to the time required to perform sum-product. Since we perform exact inference, sum-product is the best we can expect in general. We observe that all of our query and update operations exhibit running times that are logarithmic in n, and are between one to four orders of magnitude faster than a from-scratch inference with the sum-product algorithm. Update and restructuring operations



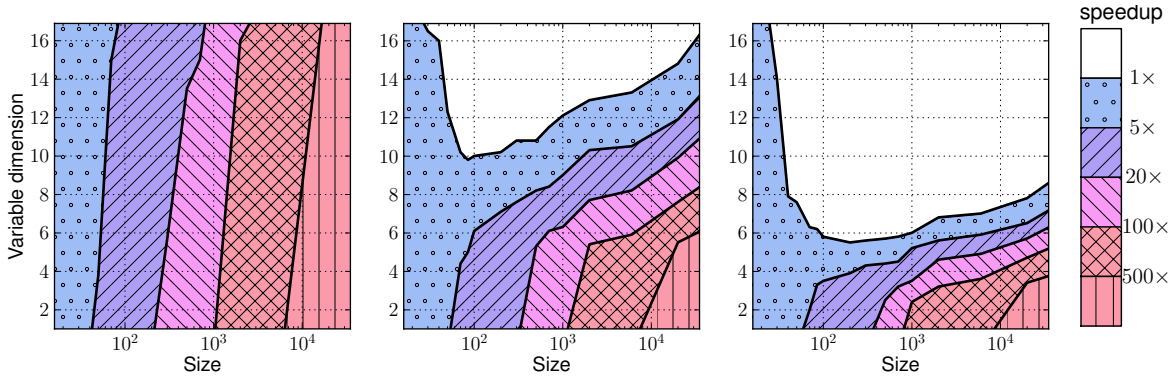
(a) tree-structured factor graphs **(b)** loopy factor graphs with tree-width 3

Figure 5.6: Marginalization queries and model updates. We measure the run times for naive sum-product, building the cluster tree, computing marginal queries, updating factors, and restructuring (adding and deleting edges to the elimination tree) for tree-structured and loopy factor graphs. Building the cluster tree is slightly more expensive than a single execution of sum-product, but subsequent updates and queries are much more efficient than recomputing from scratch. For both tree-structured and loopy graphs, our approach is about three orders of magnitude faster than sum-product.

are costlier than the query operation, as predicted by our complexity bounds on updates ($O(d^{3w} \log n)$, Theorem 5.1.1) and queries ($O(d^{2w} \log n)$, Theorem 4.3.1). The overall trend is logarithmic in n, and even for small graphs (100–1000 nodes) we observe a factor of 10–30 speedup. In the scenario of interest, where we perform an initial build operation followed by a large number of updates and queries, these results suggest that we can achieve significant speedups in practice.

Efficiency trade-offs and the constant factors

Our experiments with the computations of marginals (Section 5.3.2) suggest that our proposed approach can lead to efficiency improvements and significant speedups in practice. In this section, we present a more detailed analysis by considering a broader range of graphs and by presenting a more detailed analysis by considering constant factors and realized



(a) tree width 2 **(b)** tree width 3 **(c)** tree width 4

Figure 5.7: Speedup Analysis. The regions where we obtain speedup, defined as the ratio of running time of our algorithm for a single update and query to the running time of standard sum-product, are shown for loopy graphs with width $2, 3$ and 4.

exponents.

For a graph of n nodes with tree-width w and dimension d, inference of marginals using sum product algorithm requires $O(d^{w+1}n)$ time. With adaptive inference, the preprocessing step takes $O(d^{3w}n)$ time whereas updates and queries after unit changes require $O(d^{3w}\log n)$ and $O(d^{2w}\log n)$ time respectively. These asymptotic bounds imply that using updates and queries, as opposed to performing inference with sum-product, would yield a speedup of $O(\frac{n}{d^{3w}\log n})$, where d is the dimension (domain size) and w and n is the tree-width and the size of the graphical model. In the case that d and w can be bounded by constants, this speedup would result in a near linear efficiency increase as the size of the graphical model increases. At what point and with what inputs exactly the speedups materialize, however, depends on the constant factors hidden by our asymptotic analysis. For example in Figure 5.6, we obtain speedups for nearly all graphs considered.

Speedups for varying input parameters. To assess further the practical effectiveness of adaptive inference, we have measured the performance of our algorithm versus sum-product for graphical models generated at random with varying values of d, w and n. Specifically, for a given d, w, n we generate a random graphical model as previously described and measure

the average time for ten randomly generated updates plus queries, and compare this to the time to perform from-scratch inference using the sum-product algorithm. The resulting speedup is defined as the ratio of the time for the from-scratch inference to the time for the random update plus query.

Figure 5.7 illustrates a visualization of this speedup information. For tree-widths $2, 3, 4$, we show the speedup expected for each pair of values (n, d). Given fixed w, d we expect the speedup to increase as n increases. The empirical evaluation illustrates this trend; for example, at $w = 3$ and $d = 4$, we see a five fold or more speedup starting with input graphs with $n \approx 100$. As the plots illustrate, we observe that when the tree-width is 2 or less, as in Figure 5.7a, adaptive inference is preferable in many cases even for small graphs. With tree-widths 3 and 4, we obtain speedups for dimensions below 10 and 6 respectively. We further observe that for a given width w, we obtain higher speedups as we reduce the dimensionality d and as we increase n, except for small values of n. Disregarding such small graphs, this is consistent with our theoretical bounds. In small graphs ($n < 100$) we see higher speedups than predicted because our method's worst-case exponential dependence is often not achieved, a phenomenon we examine in more detail shortly.

Constant Factors. The experiments shown in Figures 5.7 and 5.6 show that adaptive inference can deliver speedups even for modest input sizes. To understand these result better, it helps to consider the constant factors hidden in our asymptotic bounds. Taking into account the constant factors, we can write the dynamic update times with adaptive inference as $\alpha_a d^{3w} \log n + \beta_a \log n$, where α_a, β_a are constants dependent on the cost of operations involved. The first term $\alpha_a d^{2w} \log n$ accounts for the cost of matrix computations (when computing the cluster functions) at each node and the term $\beta_a \log n$ accounts for the time to locate and visit the $\log n$ nodes to be updated in the cluster-tree data structure. In comparison, sum-product algorithm requires $\alpha_s d^{w+1} n + \beta_s n$ time for some constants α_s, β_s which again represent matrix computation at each node and the finding and visiting of the

nodes. Thus the speedup would be $\frac{\alpha_s d^{w+1} n + \beta_s n}{\alpha_a d^{3w} \log n + \beta_a \log n}$.

These bounds suggest that for fixed d, w, there will be some n_0 beyond which speedups will be possible. The value of n_0 depends on the relationships between the constants. First, constants α_a and α_s are similar because they both involve similar matrix operations. Also, the constants β_a and β_s are similar because they both involve traversing a tree in memory by following pointers. Given this relationship between the constants, if the non-exponential terms dominate, i.e., $\beta \gg \alpha$, then we can obtain speedups even for small n.

Our experiments showing that speedups are realized at relatively modest input sizes suggest that the βs dominate the αs. To test this hypothesis, we measured separately the time required for the matrix operations. For an example model with $n = 10000, w = 3, d = 6$, the matrix operations (the first term in the formulas) consumed roughly half the total time: 8.3 seconds, compared to 7.4 seconds for the rest of the algorithm. This suggests that βs are indeed larger than the αs. This should be expected: the constant factor for matrix computation, performed locally and in machine registers, should be far smaller than the parts of the code that include more random memory accesses (e.g., for finding nodes) and likely incur cache misses as well, which on modern machines can be hundreds of times slower than register computations.

While this analysis compares the dynamic update times of adaptive inference, comparing the pre-processing (build) time of our cluster tree data structures (Figure 5.6) suggests that a similar case holds. Specifically, in Theorem 4.2.1 we showed that the building the cluster tree takes in the worst case $\Theta(d^{3w} \cdot n)$ whereas the standard sum-product takes $\Theta(d^{w+1} \cdot n)$. Thus the worst-case build time could be $d^{2w} = 6^{2 \cdot 3} = 46656$ times slower than standard sum-product. In our experiments, this ratio is significantly lower. For a graph of size 50,000, for example, it is only 3.05. Figure 5.6(b) also shows a modest increase in build time as the input size grows. For example at $n = 100$, our build time is about 1.20 slower than performing sum-product. Another 100-fold increase in the size makes our build time about

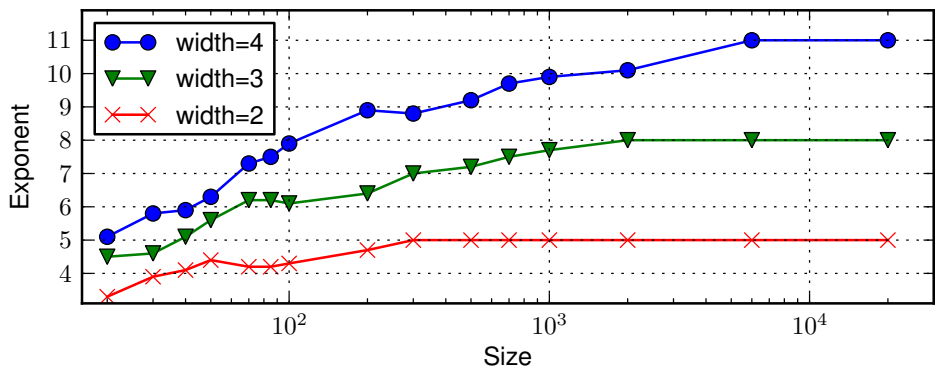


Figure 5.8: Cost of cluster computation. The maximum exponent e during the computation of clusters, which takes $O(d^e)$ time, is plotted as a function of the input size. As can be seen, the exponent starts relatively small and increases to reach the theoretical maximum of three times the tree-width as the graph size increases. Since the cost of computing clusters in our algorithm is $O(d^e)$, our approach can yield speedup even for small and medium-sized models. This shows that our worst-case bound of $O(d^{3w})$ for computing clusters can be pessimistic, i.e., it is not tight except in larger graphs.

2.05 slower. As we illustrate in next this section, this is due to our bounds not being tight in small graphs.

It is also worth noting that the differences between the running times of query and update operations are also low in practice, in contrast to the results of Theorems 5.2.5 and 4.3.1. According to Theorems 5.2.5 and 4.3.1, the query operation could, in the worst-case, be $d^w = 6^3 = 216$ times faster than an update operation. However, in practice we see that, for example at $n = 100$, the queries are about 2.5 times faster than updates. This gap does increase as n increases, e.g., at $n = 50000$, queries are about 6.7 times faster than updates; this is again due to our bounds not being tight in small graphs (described in detail next).

Tightness of our bounds in small graphs. Our experiments with varying sizes of graphs show some unexpected behavior. For example, contrary to our bound that predicts speedup to increase as the input size increases, we see in Figure 6.4 that speedups occur for very small graphs (less than 100 nodes) then disappear as the graph size increases. To understand the reasons for this we calculated the actual exponential factor in our bounds occurring in our

randomly generated graphs, by building each cluster-tree and calculating the maximum exponent encountered during the computation. Figure 5.8 shows the measurements, which demonstrate that for small graphs the worst case asymptotic bound is not realized because the exponent remains small. In other words, we perform far fewer computations than would be predicted by our worst-case bound. As the graph size grows, the worst case configurations become increasingly likely to occur, and the exponent eventually reaches the bound predicted by our analysis. This suggests that our bounds may be loose for small graphs, but more accurate for larger graphs, and explains why speedups are possible even for small graphs.

CHAPTER 6

MAINTAINING MAP CONFIGURATIONS

The previous chapters provide for efficient marginal queries to user-specified variables and can be extended to compute max-marginals when each sum is replaced with max in the formulas. While we can query each max-marginal, since we do not know *a priori* which entries of the MAP configuration have changed, in the worst case it may take linear time to update the entire MAP configuration. In this chapter, we show how to use the cluster tree data structure along with a tree traversal approach to efficiently update the entries of the MAP configuration. More precisely, for a change to the model that induces m changes to a MAP configuration, our algorithm computes the new MAP configuration in time proportional to $m \log(n/m)$, without requiring *a priori* knowledge of m or which entries in a MAP configuration will change.

6.1 Computing MAP configurations using a cluster tree

In Chapter 4, we described how to compute a cluster tree for computing marginals for any given variable. In this section, we show how this cluster tree can be modified to compute a MAP configuration. First, we modify Equation (4.1) for computing a cluster λ_j to be

$$\lambda_j = \max_{\mathcal{V}_j} f_j \prod_{\lambda_k \in \mathcal{C}_T(f_j)} \lambda_k \tag{6.1}$$

where $\mathcal{V}_j \subseteq X$ is the set of children variables of λ_j and $\mathcal{C}_T(f_j)$ is the set of children clusters of λ_j in the cluster-tree. For MAP computations, rather than using boundaries we make use of the argument set of clusters. The argument set X_{λ_j} of a cluster λ_j is the set of variables λ_j depends on at time it was created and it is implicitly computed as we perform hierarchical clustering.

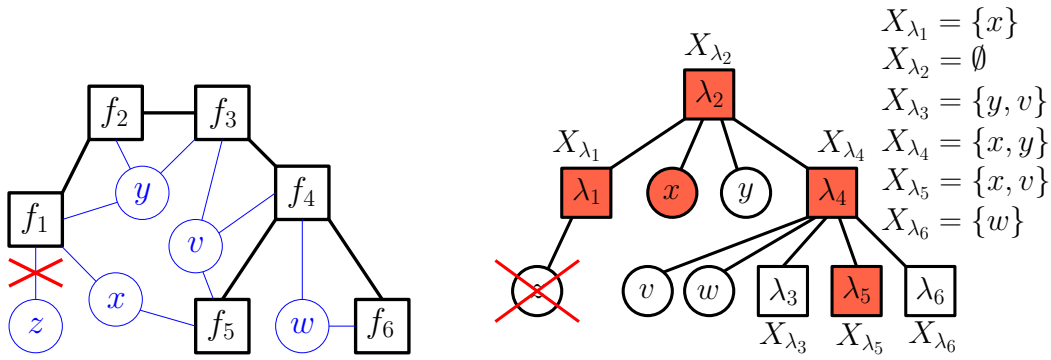


Figure 6.1: Updating a MAP configuration. Factor f_1 is modified and no longer depends on z on the factor graph (left). We first update the clusters on the path from modified clusters to the root, namely, λ_1 and λ_2. Then, we check for changes to the MAP configuration using a top-down traversal in the cluster-tree (right). Here x is assumed to have a different MAP configuration than before, which requires us to check for changes to the MAP configuration in clusters with x in their argument sets, namely λ_4, λ_5. The argument set for each cluster is annotated in the cluster tree.

We now perform a downward pass, in which we select an optimal configuration for the variables associated with the root of the cluster tree, then at its children, and so on. During this downward pass, as we reach each cluster λ_j, we choose the optimal configurations for its children variables $\mathcal{V}_j$ using

$$\mathcal{V}_j^* = \arg\max_{X} \; \delta(X_{\lambda_j} = X_{\lambda_j}^*) \; f_j \prod_{\lambda_k \in \mathcal{C}_T(f_j)} \lambda_k \tag{6.2}$$

where $\delta(\cdot)$ is the Kronecker delta, ensuring that λ_j's argument set X_{λ_j} takes on value $X_{\lambda_j}^*$. By the recursive nature of the computation, we are guaranteed that the optimal configuration $X_{\lambda_j}^*$ is selected before reaching the cluster λ_j. This can be proven inductively: assume that X_{λ_j} has an optimal assignment when the recursion reaches the cluster λ_j. We are conditioning on X_{λ_j}, which is the Markov blanket for λ_j, and can therefore optimize the subtree of λ_j independently. The value in Equation (6.2) is thus the optimal configuration for $\mathcal{V}_j$ (which by definition includes Markov blanket) for each child cluster λ_k; see Figure 6.1 for an example.

Theorem 6.1.1 *Let G be a factor graph with n nodes and T be an elimination tree on G with tree-width w. The MAP configuration can be computed in $O(nd^{3w})$ time.*

Proof: Computation of the formulas in Equations (6.1) and (6.2) takes $O(d^{3w})$ by Lemma 4.2.2. Since the algorithm visits each node twice, once bottom-up using Equation (6.1) and once top-down using Equation (6.2) the total cost is $O(nd^{3w})$. ■

6.2 Updating MAP configurations under changes

In this section we show, somewhat surprisingly, that the time required to update a MAP configuration after a change to the model is proportional to the number of changed entries in the MAP configuration, rather than the size of the model. Furthermore, the cost of updating the MAP configuration is in the worst case linear in the number of nodes in the factor graph, ensuring that changes to model result in no worse cost than computing the MAP from scratch. This means that, although the extent of any changed configurations is not known *a priori*, it is identified automatically during the update process. For the sake of simplicity, we present the case where we modify a single factor. However, with little alteration the algorithm also applies to an arbitrary number of modifications both to the factors and to the structure of the model.

Let $G = (X, F)$ be a factor graph and $\mathcal{H}$ be its cluster tree. Suppose that we modify a factor $f_1 \in F$ and let λ_1 be the cluster formed after eliminating f_1. Let $\lambda_1, \lambda_2, \ldots, \lambda_k$ be the path from λ_1 to the root λ_k in $\mathcal{H}$. As in Chapter 5, we recompute each cluster along the path using Equation (6.1). We additionally mark these clusters *dirty* to indicate that they have been modified. In the top-down phase we search for changes to and update the optimal configuration for the children variables of each cluster. Beginning at the root, we move downward along the path, checking for a MAP change. At each node, we recompute the optimal MAP configuration for the children variables and recurse on any children cluster who

is marked as dirty or whose argument set has a variable with a changed MAP configuration.

Figure 6.1 shows an example of how a MAP configuration changes after a factor (f_1) is changed in the factor graph. The bottom-up phase marks λ_1 and λ_2 dirty and updates them. The top-down phase starts at the root and re-computes the optimal configuration for x and y. Assuming that the configuration for x is changed, the recursion proceeds on λ_1 due to the dirty cluster and λ_4 due to the modified argument set. At λ_4 we re-compute the optimal MAP configurations for v and w and assuming nothing has changed, we proceed to λ_5 and terminate.

We now prove the correctness and overall running time of this procedure.

Theorem 6.2.1 *Suppose that we make a single change to a factor in the input factor graph G, and that a MAP configuration of the new model differs from our previous result on at most m variables. Let $\gamma = min(1 + rm, n)$, where r is the maximum degree of any node in G. After updating the cluster tree, the MAP update algorithm can find m variables and their new MAP configurations in $O(\gamma(1 + \log(\frac{n}{\gamma}))d^w)$ time.*

Proof: Suppose that after the modified factor is changed, we update the cluster tree as described in Chapter 5. To find the new MAP, we revisit our decision for the configuration of any variables along this path.

Consider how we can rule out any changes in the MAP configuration of a subtree rooted at λ_j in the cluster tree. First, suppose that we have found all changed configurations above λ_j. The decision at λ_j is based on its children clusters and the configuration of its argument set which is the set of variables on the boundary of the cluster: if none of these variables have changed, and no clusters used in calculating λ_j have changed, then the configuration for all nodes in the subtree rooted λ_j remains valid. Thus, our dynamic MAP update procedure correctly finds all the changed m variables and their new MAP configurations.

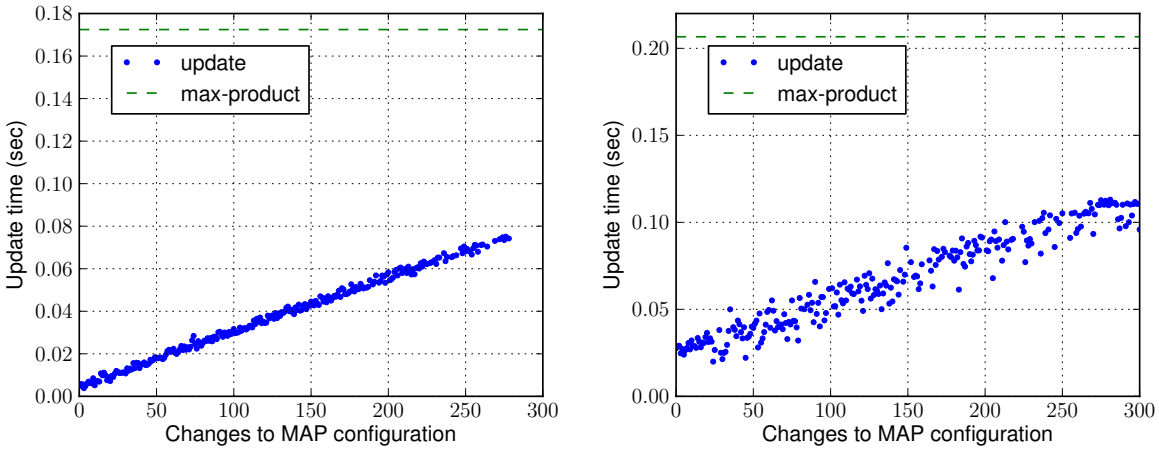
Now suppose that m variables have changed the value they take on in the new MAP configuration. The total number of paths with changed argument set is then at most rm.

These paths are of height $O(\log n)$, and every node is checked at most once, ensuring that the total number nodes visited is at most $O(\gamma \log(\frac{n}{\gamma}))$ where $\gamma = \min(1 + rm, n)$. Each visit to a cluster λ_j decodes the optimal configuration for its children variables $\mathcal{V}_j$ using Equation (6.2). Since we are conditioning on the argument set, this computation takes $O(d^{|\mathcal{V}_j|})$ time. Using arguments as in the proof of Lemma 4.1.1, we can show that $|\mathcal{V}_j| \leq w$. Therefore the top-down phase takes $O(\gamma(1 + \frac{n}{\gamma})d^w)$ time. ■

It is also possible, using essentially the same procedure, to process batch updates to the input model. Suppose we modify G or its elimination tree T by inserting and deleting a total of ℓ edges and nodes. First, we use the method described in Chapter 5 to update the clusters. Then, the total number of nodes recomputed (and hence marked dirty) is guaranteed to be $O(\ell \log(n/\ell))$. Note that we also require $O(\ell \log(n/\ell))$ time to identify new points of elimination for at most ℓ variables. Therefore, the bottom-up phase will take $O(d^{3w} \ell \log(n/\ell))$ time. The top-down phase works exactly as before and can check an additional $O(rm)$ paths for MAP changes where m is the number of variables with changed MAP value and r is the maximum degree in G. Therefore the top-down phase takes $O(\gamma \log(\frac{n}{\gamma})d^w)$ time where $\gamma = \min(\ell + rm, n)$.

6.3 Experimental Analysis

We evaluate the effectiveness of our approach for synthetically generated graphs as well as two applications in computational biology. The synthetic examples aim at demonstrating linear dependence of adaptively computing MAP computations on the number of entries changed in the MAP configuration. The first computational biology application studies adaptivity in the context of using an HMM for the standard task of protein secondary structure prediction. For this task, we show how a MAP configuration that corresponds to the maximum likelihood secondary structure can be maintained as mutations are applied to the primary



(a) tree-structured **(b)** loopy

Figure 6.2: Updates to MAP configurations. We report the time required to update a MAP configuration after a single change is made to the input model, in both tree-structured and loopy factor graphs, with 300 variables. Our algorithm takes time that is roughly linear in the number of changed entries, unlike the standard max-product algorithm, which takes time that is linear in the size of the model.

sequence. The second application applies our approach to higher-order graphical models that are derived from three-dimensional protein structure. We show our algorithm can efficiently maintain the minimum-energy conformation of a protein as its structure undergoes changes to local sidechain conformations.

6.3.1 Experiments with Synthetic Data

As described in Section 5.3.1 in detail, we generate two sets of synthetic data: tree-structured and loopy factor graphs. A random tree-structured graph with n variables and domain size d is generated with a "chainness" parameter p. When p is set to one, the random graph becomes a chain, $p = 0$ creates a random graph where every edge is equally likely. The second set of generated graphs are loopy graphs where we start with a simple Markov chain. Then for parameter w, we generate random links between nodes so that the width of the factor graph along the Markov chain is w.

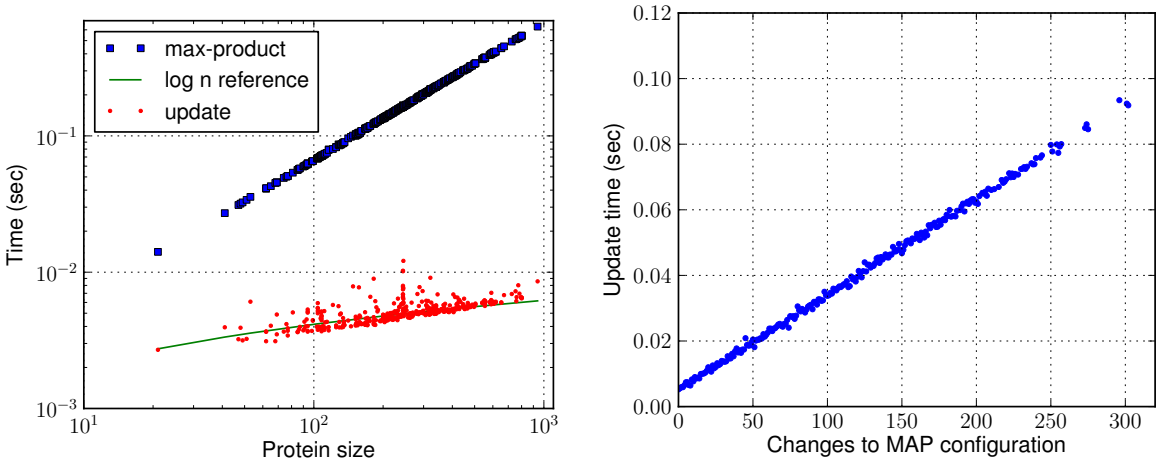
For these experiments we generated factor graphs with tree-width one (trees) and three

comprised of $n = 300$ variables. For trees, we choose $d = 25$ and for graphs we choose $d = 6$. We compute the update time by uniformly randomly selecting a factor and replacing with another factor, averaging over 100 updates. We compare the update time to the running-time of the max-product algorithm, which computes messages from leaves to a chosen root node in the factor graph and then performs maximization back to the leaves.

Figure 6.2 show the results of our experiments. For both tree-structured and loopy factor graphs, we observed strong linear dependence between the time required to update the number of changed entries in the MAP configuration. We note that while there is an additional logarithmic factor in the running time, it is likely negligible since n was set to be small enough to observe changes to the entire MAP configuration. Overall, our method of updating MAP configurations was substantially faster than computing a MAP configuration from scratch in all cases, for both tree-structured and loopy graphs.

6.3.2 Sequence Analysis with Hidden Markov Models

HMMs are a widely-used tool to analyze DNA and amino acid sequences; typically an HMM is trained using a sequence with known function or annotations, and new sequences are analyzed by inferring hidden states in the resulting HMM. In this context, our algorithm for updating MAP configurations can be used to study the effect of changes to the model and observations on hidden states of the HMM. We consider the application of secondary structure prediction from the primary amino acid sequence of a given protein. This problem has been studied extensively [22], and is an ideal setting to demonstrate the benefits of our adaptive inference algorithm. An HMM for protein secondary structure prediction is constructed by taking the observed variables to be the primary sequence and setting the hidden variables (i.e., one hidden state per amino acid) to be the type of secondary structure element (α-helix, β-strand, or random coil) of the corresponding amino acid. Then, a MAP configuration of the hidden states in this model identifies the regions with α helix and β



(a) Update runtimes for 400 proteins

(b) Update times for protein 1QOY

Figure 6.3: Secondary structure prediction using HMMs. We applied our algorithm to perform udpates in HMMs for secondary structure prediction. For our data set, we can perform MAP updates about 10-100 faster than max-product, and we see a roughly logarithmic trend as the size of the protein increases. For a single protein, *E. coli hemolysin*, we see that the time required to upate the MAP configuration is linear in the number of changes to the MAP configuration, rather than in the size of the HMM.

strands in the given sequence. This general approach has been studied and refined [13, 38], and is capable of accurately predicting secondary structure. In the context of secondary strucure prediction, our algorithm to adaptively update the model could be used in protein design applications, where we make "mutations" to a starting sequence so that the resulting secondary structure elements match a desired topology. Or, more conventionally, our algorithm could be applied to determine which residues in the primary sequence of a given protein are critical to preserving the native pattern of secondary structure elements. It is also worth pointing out that our approach is fully general and can be used in any application where biological sequences are represented by HMMs (e.g., DNA, RNA, exon-intron chains and CpG islands) and we want to study the effects of changes to these sequences.

For our experiments, we constructed an HMM for secondary structure prediction by constructing an observed state for each amino acid in the primary sequence, and a corresponding hidden state indicating its secondary structure type. We estimated the model

parameters using 400 protein sequences labeled by the DSSP algorithm [28], which annotates a three-dimensional protein structure with secondary structure types using standard geometric criteria. Since repeated modification to a protein sequence typically causes small updates to the regions with α helices and β strands, we expect to gain significant speedup by using our algorithm. To test this hypothesis, we compared the time to update MAP configuration in our algorithm against the standard max-product algorithm. The results of this experiment are given in Figure 6.3(a). We observed that overall the time to update secondary structure predictions were 10-100 times faster than max-product. The overall trend of running times, when sorted by protein size, is roughly logarithmic. In some cases, smaller proteins required longer update times; in these cases it is likely that due to the native secondary structure topology, a single mutation induced a large number of changes in the MAP configuration. We also studied the update times for a single protein, *E. coli hemolysin* (PDB id: 1QOY), with 302 amino acids, as we apply random mutations (see Figure 6.3(b)). As in Section 6.3.1 above, we see that the update time scales linearly with the number of changes to a MAP configuration, rather than depending on the size of the primary sequence.

6.3.3 Protein Sidechain Packing with Factor Graphs

In the previous section, we considered an application where the input model was a chain-structured representation of the protein primary sequence. In this section, we consider a higher-order representation that defines a factor graph to model the three-dimensional structure of protein, which essentially defines its biochemical function. Graphical models constructed from protein structures have been used to successfully predict structural properties [56] as well as free energy [29]. These models are typically constructed by taking each node as an amino acid whose states represent a discrete set of local conformations called *rotamers* [20], and basing conditional probabilities on a physical energy function (e.g., [54, 12]).

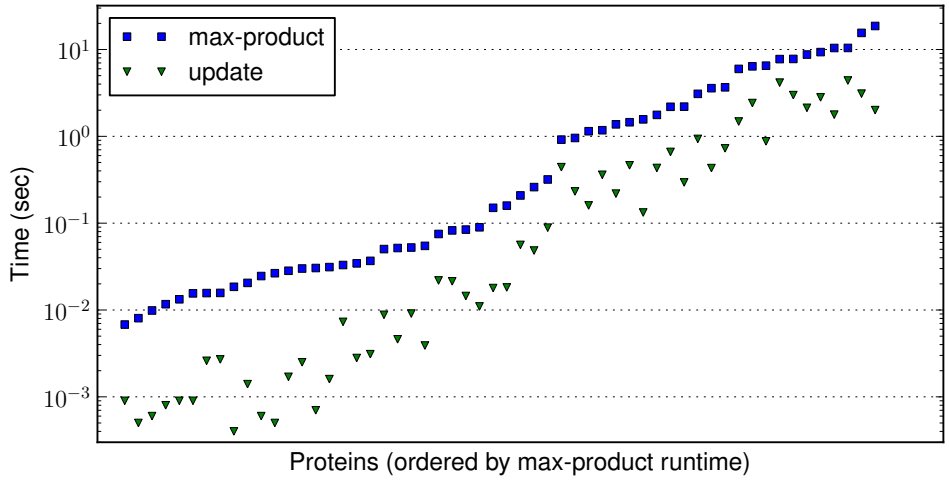


Figure 6.4: Adaptive sidechain packing for protein structures. For 60 proteins from the SCWRL benchmark, we compared the time to adaptively update a MAP configuration against max-product. Since this set of proteins has a diverse set of folds (and thus graph structures), we order the inputs by the time taken by max-product. The speedup achieved by our algorithm varies due to the diversity of protein folds, but on average our approach is 6.88 times faster than computation from scratch.

The typical goal of using these models is to efficiently compute a maximum–likelihood (i.e. minimum–energy) conformation of the protein in its native environment. Our algorithmic framework for updating MAP configurations allows us to study, for example, the effects of amino acid mutations, and the addition and removal of edges corresponds directly to allowing backbone motion in the protein. Applications that make use of these kinds of perturbations include protein design and ligand-binding analysis. The common theme of these applications is that, given an input protein structure with a known backbone, we wish to characterize the effects of changes to the underlying model (e.g., by modifying amino acid types or their local conformations), in terms of their effect on a MAP configurations (i.e. the minimum energy conformation of the protein).

For our experiments, we studied the efficiency of adaptively updating the optimal sidechain conformation after a perturbation to the model in which a random group of sidechains are fixed to new local conformations. This experiment is meant to mimic a ligand-binding study, in which we would like to test how introducing ligands to parts of the protein structure af-

fect the overall minimum-energy conformation. For our dataset, we took about 60 proteins from the SCWRL benchmark or varying sizes (between 26 and 244 amino acids) and overall topology.

For each protein, we chose applied updates to a random group within a selected set amino acids (e.g., to represent an active site) by choosing a random rotameric state for each. With appropriate preprocessing (using Goldstein dead-end elimination), we were able to obtain accurate models with an induced width of about 5 on average. For the cluster tree corresponding to each protein we selected a set of 10 randomly chosen amino acids for modification, and recorded the average time, over 100 such trials, to update a MAP configuration and compared it against computing the latter from scratch. The results of our experiment are given in Figure 6.4. Due to the diversity of protein folds, and thus the resulting factor graphs, we sort the results according to the time required for max-product. We find that our approach consistently outperforms max-product, and was on average 6.88 times faster than computation from scratch.

We note that the overall trend for our algorithm versus max-product is somewhat different than the results in Sections 6.3.1 and 6.3.2. In those experiments we observed a clear logarithmic trend in running times for our algorithm versus max-product, since the constant-factor overheads (e.g., for computing cluster functions) grew as a function of a model size. For adaptive sidechain packing, it is difficult to make general statements about the complexity of a particular input model with respect to its size: a small protein may be very tightly packed and induce a very dense input model, while a larger protein may be more loosely structured and induce a less dense model.

CHAPTER 7

PARALLELISM FOR APPROXIMATE INFERENCE

In this chapter, we present an approach to parallelizing dual-decomposition solvers using the cluster tree data structure that we developed in Chapter 4. The cluster tree can be used in a way that combines the per-iteration advantages of large subproblems while also enabling a high degree of parallelism. In addition to being highly parallelizable, the adaptive properties of the cluster tree allow minor changes to a model to be incorporated and re-solved in far less time than required to solve from scratch. We show that in dual-decomposition solvers this adaptivity can be a major benefit, since often only small portions of the subproblems' parameters are modified at each iteration.

7.1 Parallelism without sacrifice

Parallel calculation of tree-structured formulas has been studied in the algorithms community using a technique called *tree contraction* [44]. Algorithms based on this idea have been applied to speed up exact inference tasks in a variety of settings [42, 55]. The cluster tree is one such data structure that is based on tree contraction and can be applied effectively to improve dual-decomposition solvers.

As a motivating example, consider a subproblem that consists of a single Markov chain (Figure 7.1a). A standard solver for this model works by dynamic programming, sequentially eliminating each f_i from leaves to root and computing a message λ_i, interpreted as a "cost to go" function, then back-solving from root to leaves for an optimal configuration of variables. This process is hard to parallelize, since any exact solver must propagate information from one end of the chain to the other end and thus requires a sequence of $\Omega(n)$ operations in this framework. Instead, if we use the cluster tree Figure 7.1b, its balanced shape will make it possible to compute many branches in parallel. Using similar ideas, [42] showed that, with

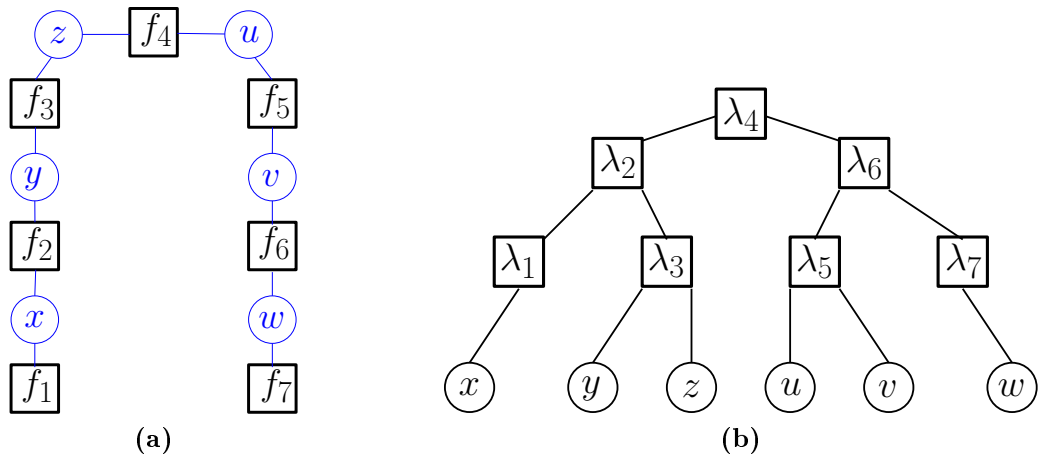


Figure 7.1: Markov chain parallelizability. An example cover tree in the form of a Markov chain **(a)** subproblem. It is hard to parallelize due to the information that must be traveled from one end f_1 of the chain to the other end f_7. When cluster tree is used **(b)**, the branches can potentially be computed in parallel.

a sufficient number of processors, parallel exact inference requires $O(\log n)$ time.

As previously discussed, the benefit of parallelizing dual-decomposition inference methods can be lost if the subproblems are not chosen appropriately. For a concrete example, consider again the Markov chain of Figure 7.1a. A decomposition into edges (Figure 7.2a) achieves high parallel efficiency: each edge is optimized independently in parallel and disagreements are successively resolved by parallel independent projected subgradient updates. Given enough processors, each iteration in this setup requires constant time, but overall inference still requires $\Omega(n)$ iterations for this procedure to converge. Thus there is no substantial gain from parallelizing inference on even this simple model, since in the sequential case we reach the solution in $O(n)$ time. Thus, we must balance the degree of parallelism and the convergence rate to achieve the best possible speedup. In the remainder of this section, we discuss how the cluster tree data structure permits us to achieve this balance, and how its adaptive properties provide an additional speedup near convergence.

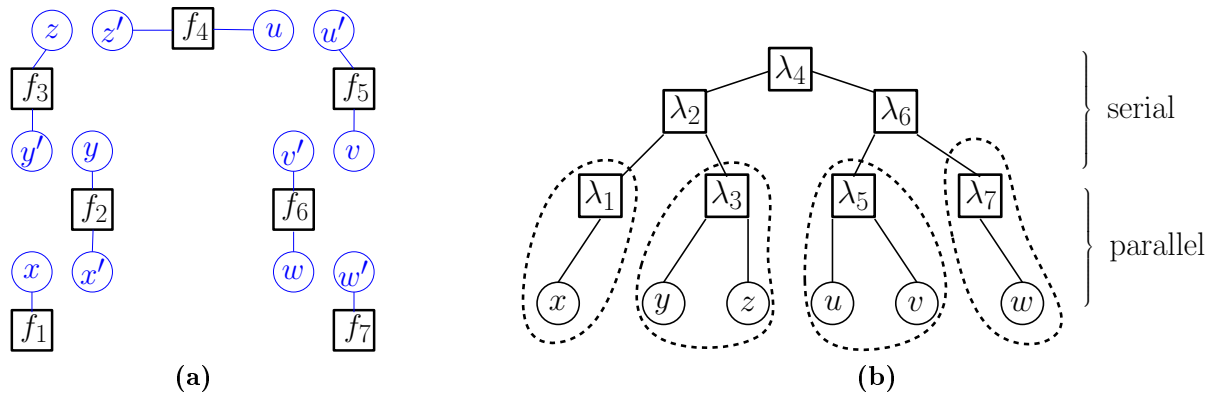


Figure 7.2: Parallelism using independent factors vs. cluster tree. **(a)** Dual-decomposition using independent factors is highly parallelizable but slow to converge. **(b)** The cluster tree is also easily parallelizable, but since the underlying model is a covering tree, convergence is not compromised.

7.2 Parallelization with a Cluster Tree

A cluster tree structure can manage both long-range influence in each iteration (improving convergence rate) as well as parallelization obtained by exploiting its balanced nature. To perform parallel computations in a cluster tree with n nodes, we split the tree into a set of *parallel trees* and a *serial tree*, as shown in Figure 7.2. For bottom-up computations, we can assign a processor to each parallel tree. When these computations are complete, the remainder of the bottom-up computation is performed on a single processor. Top-down computations are performed analogously, except that they start with the serial tree and then continue in parallel on the parallel trees. The number of parallel trees defines both the level of parallelism as well as the depth of the serial tree, and thus these two choices must be balanced. In practice we choose the depth of the serial tree to be half of the depth of the cluster tree, to keep synchronization costs low. Then, for a cluster tree with n nodes we achieve parallelism of $\sqrt{n}$, while preserving a high convergence rate.

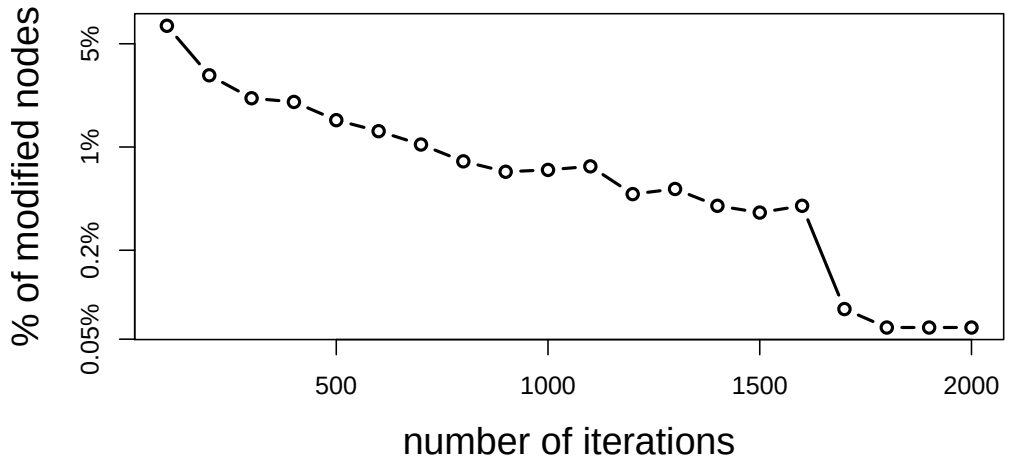


Figure 7.3: Importance of adaptivity in dual-decomposition methods. The number of changes made to the parameters by the projected subgradient updates, measured on a stereo matching example (Venus) from the experimental section.

7.3 Adaptivity for subgradient updates

Another major source of improvement in speed is obtained by judiciously reusing solutions at the previous iteration. In particular, the subgradient update Equation (3.4) depends only on the optimal configuration a^t of each subproblem t, and modifies only those subproblems that do not share the optimal value of some x_i. Many ψ_i^t, then, are not modified from iteration to iteration; the updates become sparse. This behavior is common in many real-world models: Figure 7.3 illustrates for a stereo matching problem, showing that in practice the number of updates diminishes significantly near convergence. The standard approach recomputes the optimal values for every variable in an updated subproblem ψ^t.

This observation motivates an *adaptive* subgradient calculation, in which we leverage the previous iteration's solution to speed up the next. In collections of very small subproblems, this is easily accomplished: we can simply preserve the solution of any subproblem whose potentials ψ^t were not modified at all. While this may appear hard to use in a cover tree

representation (in which all variables are present), in this section we show how to use the cluster tree data structure to leverage update sparsity and give further speedup even if the underlying subproblem is a cover tree.

Let G be the tree corresponding to the subproblem ψ^t and T be the associated cluster tree constructed as described in Chapters 4 and 6. Suppose that the projected subgradient updates modify singleton potentials $\psi_1, \psi_2, \ldots, \psi_\ell$. For the next iteration, we must: (i) update the cluster tree given the new potential values, and (ii) compute any changed entries in the optimal configuration.

We use the MAP maintenance method developed in Chapter 6 as a sub-routine to repeatedly perform these steps. The subroutine performs step (i) by updating the cluster tree messages in a bottom-up fashion, starting at nodes that correspond to the changed potentials and updating their ancestors. For each re-computation, it flags the messages as having been modified. We showed in Chapter 6 that this step recomputes only $O(\ell \log n/\ell)$ many messages. For part (ii), the subroutine computes a new optimal configuration by traversing top-down. It begins by selecting an optimal configuration for the root node x_r of the cluster tree. It then recursively proceeds to any children x_i for which either any λ_j was modified for which $x_j \in C_i$, or the optimal configuration of the variables in E_i were modified. If the projected subgradient update Equation (3.4) modifies k configurations, we can solve the subproblem in $O(k \log n/k)$ time: potentially much faster than doing the same updates in the original tree. For illustration, consider Figure 7.3; between iterations 1700 and 2100, the re-computation required is about one 1000th of the 1st iteration. This property alone can provide significant speed-up near convergence.

7.4 Experiments

Our cluster tree representation combines the benefits of large subproblems (faster convergence rates) with efficient parallelization and effective solution re-use. To assess its effec-

tiveness, we compare our algorithm against several alternative subproblem representations on both synthetic and real-world models. We focus on the general case of irregularly connected graphs for our experiments. Note that on very regular graphs, it is often possible to hand-design updates that are competitive with our approach; for example on grid-structured graphs, the natural decomposition into rows and columns results in $O(\sqrt{n})$ parallel subproblems of length $O(\sqrt{n})$ and will often attain a similar balance of adaptive, parallel updates and good convergence rate.

We compared our framework for performing dual-decomposition against three different algorithms: CoverTree, Edges and Edges-ADP. CoverTree decomposes the graph using a cover tree and updates the model without any parallelism or adaptivity. On the other end of the dual-decomposition spectrum, the Edges and Edges-ADP algorithms decompose the graph into independent edges and update them in parallel. Edges-ADP is the trivially adaptive version of the Edges algorithm, in which only the edges adjacent to an modified node are re-solved. We refer to our algorithm as ClusterTree; it uses the same cover tree graph as CoverTree in all of our experiments. All algorithms were implemented in Cilk++ [36] without locks. For synchronization we used Cilk++ reducer objects (variables that can be safely used by multiple strands running in parallel). All experiments in this section were performed on a 1.87Ghz 32-core Intel Xeon processor.

7.4.1 Synthetic Examples

We first test our approach on random, irregular graphical models. We generated graphs over n variables $x_1, \ldots, x_n$, each with domain size $d = 8$. The graph edges were generated at random by iterating over variables $x_3, \ldots, x_n$. For node x_i, we choose 2 random neighbors without replacement from the previous nodes $\{x_{i-1}, x_{i-2}, \ldots, x_1\}$ using a geometric distribution with probabilities $p = 1/2$ and $q = 1/\sqrt{n}$, respectively. With these settings, every node x_i is expected to have two neighbors whose indices are close to i and two neighbors

whose indices are roughly $i-\sqrt{n}$ and $i+\sqrt{n}$. Although highly irregular, the generated graph has characteristics similar to a grid with raster ordering, where each node x_i inside the grid has four neighbors indexed $x_{i-1}, x_{i+1}, x_{i-\sqrt{n}}$ and $x_{i+\sqrt{n}}$. Node potentials $\theta_i(x_i)$ are drawn from a Gaussian distribution with mean 0 and standard deviation $\sigma = 4$. Edge potentials $\theta_{ij}(x_i, x_j)$ follow the Potts model, so that $\theta_{ij}(x_i, x_j) = \delta(x_i \neq x_j)$. We then generate the factors: $f_i(x_i) = e^{-\theta_i(x_i)}$ for each $i \in \{1, \ldots, n\}$ and $f_{ij}(x_i, x_j) = e^{-\theta_{ij}(x_i, x_j)}$ for every edge (i, j).

We ran our algorithms for graph sizes ranging from 500 to $100,000$. To control for step size effects, γ was optimized for each algorithm and each problem instance. All algorithms were run until agreement between variable copies (indicating an exact solution). For CoverTree and ClusterTree, we used a breadth-first search tree as the cover tree. Figure 7.4a–c gives a comparison of convergence results for a representative model with $n = 20000$.

As expected, the cover tree has a better convergence rate than using independent edges as subproblems (see Figure 7.4a). When the algorithms were executed serially (see Figure 7.4b), although initally slower ClusterTree catches up to finish faster than CoverTree (due to adaptivity), and remains faster than Edges-ADP (due to a better convergence rate). With parallel execution, we observe a speedup of roughly 20× for ClusterTree, Edges-ADP and Edges; see Figure 7.4c. We can see that with parallelism, although Edges-ADP is preferable to CoverTree, ClusterTree finishes roughly two orders of magnitude faster than the other algorithms.

We also consider the convergence time of each algorithm as the graph size increases (see Figure 7.6). For relatively small graphs (e.g. $n = 500$) the difference is negligible; however, as we increase the number of nodes, the ClusterTree converges significantly more quickly than the other algorithms.

7.4.2 Stereo matching with super-pixels

The stereo matching problem estimates the depth of objects in a scene given two images, as if seen from a left and right eye. This is done by estimating the *disparity*, or horizontal shift in each object's location between the two images.

It is common to assume that the disparity boundaries coincide with color or image boundaries. Thus, one approach estimating stereo depth is to first segment the image into super-pixels, and then optimize a graphical model representing the super-pixels; see [25, 50]. This approach allows stereo matching to be performed on much larger images. We studied the performance of our algorithm for the task of stereo matching using a model constructed from a segmented image in this manner.

To define a graphical model G given super-pixels $\{s_i, \ldots, s_n\}$, we define a node for each super-pixel and add an edge (s_i, s_j) if they contain adjacent pixels in the reference image. The node potentials are defined as the cumulative truncated absolute color differences between corresponding pixels for each disparity:

$$\theta_i(d) = \sum_{(x,y) \in s_i} \min \{|\mathcal{I}_L(x,y) - \mathcal{I}_R(x-d,y)|, 20\}$$

where $\mathcal{I}_L$ and $\mathcal{I}_R$ are the intensities of the left and right image, respectively. The edge potentials are defined as

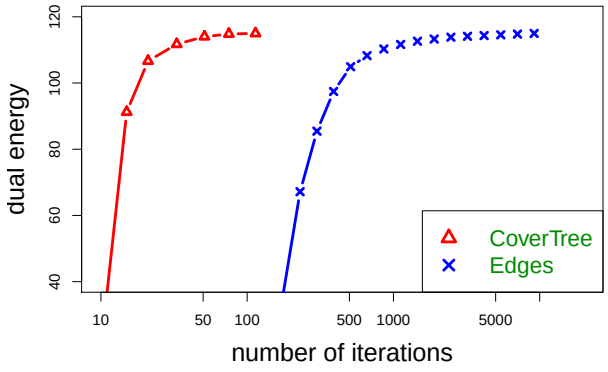
$$\theta_{ij}(d_1, d_2) = 5 \cdot E(s_i, s_j) \cdot \min \left\{ |d_1 - d_2|^{1.5}, 5 \right\}$$

where $E(s_i, s_j)$ is the number of adjacent pixel pairs (p, q) where $p \in s_i$ and $q \in s_j$. This is a common energy function for grids, [49], applied to super-pixels by [25]. We then define factors for each vertex s_i and edge (s_i, s_j) as follows

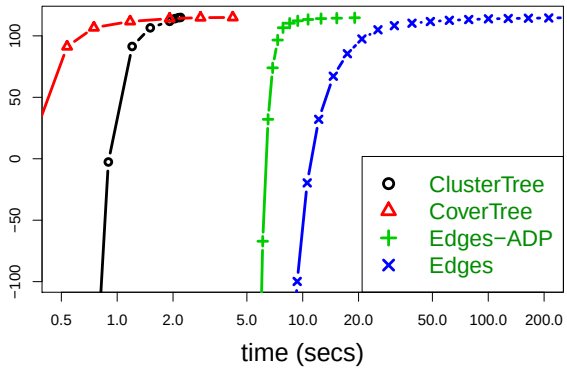
$$f_i = e^{-\theta_i(d)} \qquad f_{(i,j)} = e^{-\theta_{ij}(d_1,d_2)}$$

We tested our algorithm by constructing the above model for four images (Tsukuba, Venus, Teddy and Cones) from the Middlebury stereo data set [46, 45], using the SLIC program [10] to segment each input image into 5000 super-pixels. For CoverTree and ClusterTree, we used the maximum-weight spanning tree of G (with weights $E(s_i, s_j)$) as part of the cover tree; this is a common choice for stereo algorithms that use dynamic programming [51]. Since the model's gap between distinct energies is at least 1, the algorithms are considered converged when their lower bound is within 1 of the optimal energy.

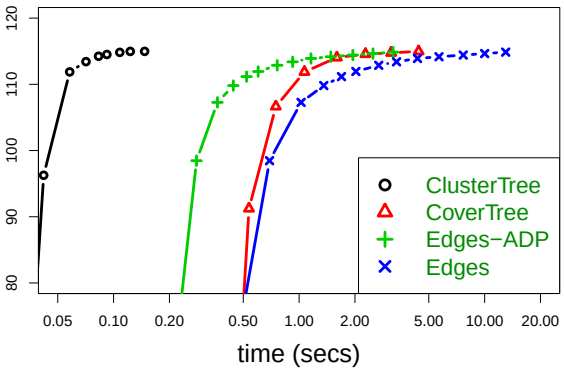
As with synthetic graphs, for the image datasets we observed that ClusterTree inherits the improved convergence rate of CoverTree but parallelizes well and thus gives much better overall performance than Edges or Edges-ADP. Representative serial and parallel executions of the algorithms are shown for the Venus dataset in Figure 7.5a–c, while convergence times are shown for all datasets in Figure 7.7. While we still observe that CoverTree has a better convergence rate than Edges, it is less dramatic than in the synthetic models (Figure 7.4a vs. Figure 7.5a); this is likely due to the presence of strong local information in the model parameters ψ_i. This fact, along with most modifications also being local, means that Edges-ADP manages to outperform CoverTree in the serial case (Figure 7.5b). In the parallel case, ClusterTree remains ahead, with a speedup of about 2× over Edges-ADP.



(a) Convergence rate

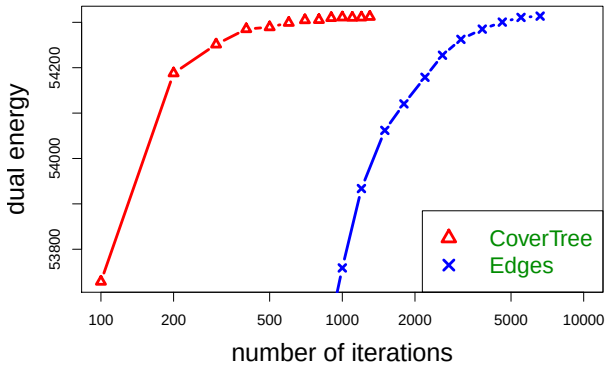


(b) Single-core convergence time

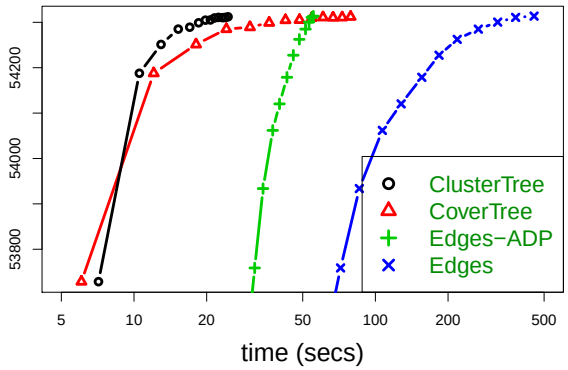


(c) 32-core convergence time

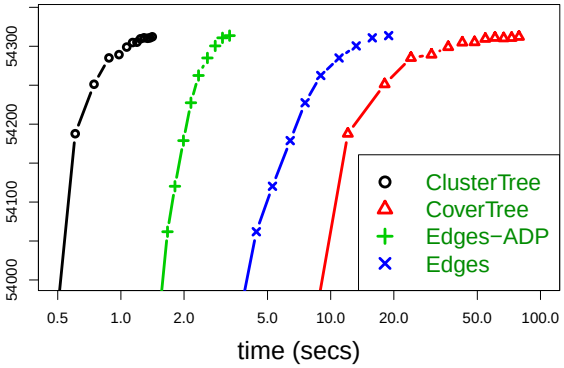
Figure 7.4: Synthetic convergence results. Representative convergence results on a random graph problem with 20000 nodes **(a-c)**. As expected, the CoverTree requires less iteration than the Edges algorithm **(a)** to converge. Without parallelism, both CoverTree and ClusterTree outperforms Edges and Edges-ADP **(b)**. With parallelism our algorithm outperforms the others **(c)**.



(a) Convergence rate for Venus



(b) Single-core convergence time for Venus



(c) 32-core convergence time for Venus

Figure 7.5: Stereo matching convergence results. Representative convergence results on the "Venus" dataset **(a-c)**. The per-iteration advantage of CoverTree over Edges **(a)** is lost when the adaptivity is taken into account **(b)**. However in terms of total running time, our algorithm ClusterTree outperforms both CoverTree and Edges-ADP with our without parallelism enabled **(b-c)**.

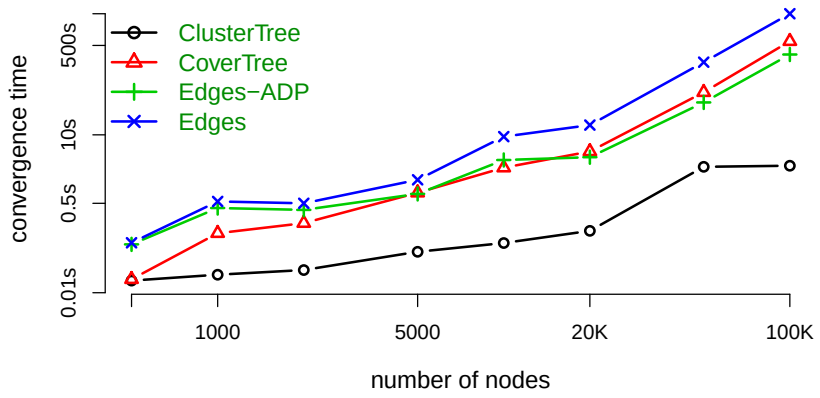


Figure 7.6: Dual-decomposition on synthetic data. For randomly generated graphs, our algorithm ClusterTree achieves a significant speedup as the model size increases when a 32-core machine is used.

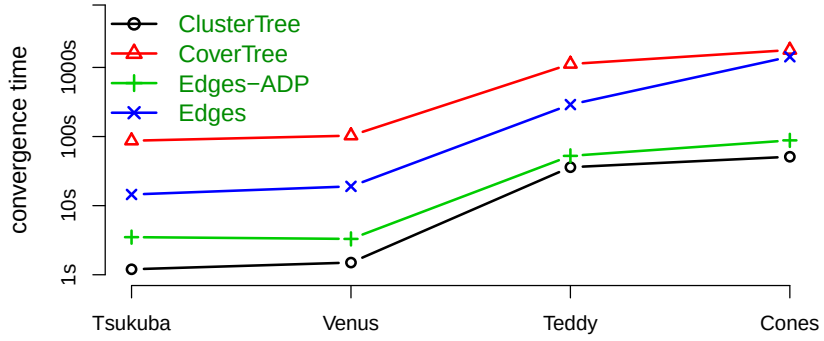


Figure 7.7: Dual-decomposition on stereo matching. For images from the Middlebury dataset, we observe the same basic trend as in synthetic data. On average our algorithm is about twice as fast as Edges-ADP.

CHAPTER 8

IMPLEMENTATION

In this chapter, we provide the python implementation for the algorithms presented in Chapters 4, 5 and 6. The code below skips some general purpose libraries such as graph.py. The reader should assume that the graph class is efficiently implemented and supplies Depth-First Search (DFS) functionality. We present our code in four sections.

Cluster-Tree Class. In our implementation we separated the tree-contraction algorithm from the inference as much as possible. The interface between these two components are provided by this class (called RCForest in the code). The cluster-tree implements an adaptive rooted tree. The tree-contraction part of the code builds and maintains the cluster-tree, then the inference computation part of the code reads and traverses this cluster-tree data structure.

Tree-Contraction Algorithm. In our context, the tree-contraction is responsible for eliminating the factors from the elimination tree. The implementation here is a general tree-contraction implementation and abstracted out from the application specific computations. The elimination tree is built and modified using `link()` (to add an edge) and `cut()` (to remove an edge) functions. The user of this library then calls `commit()` function to perform the elimination algorithm. The subsequent calls of `link()` and `cut()` and finally `commit()` is handled adaptively as described in Chapter 5. The code closely follows the exposition in Chapter 5. In particular the concept of degree status and affectedness are clearly defined in functions `__determine_degree_status()` and `__mark_affected()`.

Base Class for Exact Inference. This code provides the basic interface of the library to the user. Using this class, one can construct a factor graph with and associated elimination

tree, can modify the structure of both the factor graph and the elimination tree. This module provides the framework of bottom-up updates and marginal computations.

Inference, marginalization and computing MAP configurations. This class provides the basic operations on factors such as multiplication and maximization. The library works for any semi-ring that supplies join and project operations. In the case of marginalization we use multiplication of factors as the join and summation of factors as the project operation. The main class that inherits the base inference class is called `ApproximateInference` for a reason. The module in fact approximates the factor multiplications whenever the operation is beyond the computational limits of the program. This feature is not used in the experiments presented in this thesis. We always solve the exact inference by setting the computational limit to infinity.

8.1 Cluster-Tree Class

```
class RCForest:
  '''
  There are both python and C implementation:
  Graph-Theoretical Fields
    V = ['a', 'b', 'c']
    __Adj = { 'a':['b','c'] , 'b':['a'], 'c':['a'] }
    __parent = {'a':None, 'b':'a', 'c':'a'}
    __children = {'a':set(['b','c']), 'b':set(), 'c':set()}
    __roots = set(['a'])

  Tree-Contraction Fields, these values are just pointers
    __height = {'a':2, 'b':1, 'c':1}
```

```
    __cluster_type = {'a':0, 'b':1, 'c':1}
    __updates = ['b','c','a']

  Operations used by the inference library
    RCT.parent(v)       returns the parent vertex
    RCT.children(v)     returns the set of children of v
    RCT.roots()         returns the set of roots
    RCT.height(v)       returns the non-negative integer height of v,
                                -1 if unavailable
    RCT.cluster_type(v) returns the non-negative integer cluster type of v,
                                -1 if unavailable
    RCT.updates()       returns the list of updates

  Operations used by the tree-contraction backend
    unused (C lib doesn't implement): add_vertex, del_vertex
    make_root, make_child, push_updates
    push_updates(v,h,ct) where v = vertex, h = height, ct = cluster_type
  '''
  def __init__(self, V,  name = 'RCForest'):
    self.name = name
    if type(V) == type(1):
      self.V = range(V)
    else:
      self.V = V
    self.__roots = set(self.V)
    self.__Adj = {}
```

```
    self.__parent = {}
    self.__children = {}
    for v in self.V:
      self.__Adj[v] = set()
      self.__parent[v] = None
      self.__children[v] = set()

    self.__updates = []
    self.__height = {}
    self.__cluster_type = {}

  def parent(self, v):
    return self.__parent.get(v, None)

  def children(self, v):
    return self.__children.get(v, [])

  def children_set(self): # this is used for debugging
    return self.__children

  def roots(self):
    return self.__roots

  def height(self, v):
    return self.__height.get(v, -1)

```

```
  def cluster_type(self, v):
    return self.__cluster_type.get(v, -1)

  def updates(self):
    return self.__updates

  def add_vertex(self,v):
    if type(self.V) is type([]):
      self.V.append(v)
    elif type(self.V) is type(set()):
      self.V.add(v)
    else :
      return NotImplemented
    self.__Adj[v] = set()
    self.__parent[v] = None
    self.__children[v] = set()
    self.__roots.add(v)

  def del_vertex(self, v):
    children_to_loose = list(self.__children[v])
    self.make_root(v)
    for u in children_to_loose:
      self.make_root(u)

    self.__roots.remove(v)
    self.__Adj.pop(v)
```

```
    self.__parent.pop(v)
    self.__children.pop(v)
    self.V.remove(v)

  def make_root(self, v):
    self.__roots.add(v)
    u = self.__parent[v]
    if u is not None:
      self.__Adj[u].remove(v)
      self.__children[u].remove(v)

      self.__Adj[v].remove(u)
      self.__parent[v] = None

  def make_child(self, u, v):
    '''make_child(u,v): u becomes v's parent'''
    pv = self.__parent[v]
    if pv == u:
      return
    pu = self.__parent[u]
    if pu == v:
      raise Exception('RCForest.make_child unallowed call')
    self.make_root(v)
    self.__roots.remove(v)
    self.__Adj[u].add(v)
    self.__children[u].add(v)
```

```
    self.__Adj[v].add(u)
    self.__parent[v] = u

  def clear_updates(self):
    self.__updates = []

  def push_updates(self, v, h, ct):
    self.__updates.append(v)
    self.__height[v] = h
    self.__cluster_type[v] = ct

  def __eq__(self, other):
    if isinstance(other, RCForest):
      return self.children_set == other.children_set
    return NotImplemented

  def __ne__(self, other):
    result = self.__eq__(other)
    if result is NotImplemented:
      return result
    return not result
```

8.2 Tree-Contraction Algorithm

```
from copy import deepcopy, copy
from graph import Graph, displayDFS
from rcforest import RCForest
```

```
4
5  class Contraction:
6    def __init__(self, V, name='H', maximality = False):
7      '''
8      Initialized with the set of vertices V
9      key fields:
10     levels = [T_0, T_1, T_2, ...., T_{k+1}] where T_1 is the original tree
11       and T_k+1 = Graph([]). T_0 is not used usually same as T_1
12     affected_sets = [A_0, A_1, A_2, ...., A_k] where A_i is a subset of
13       vertices of T_i.
14     maximal_independents = [M_0, M_1, M_2, ...., M_k] where M_i \subseteq A_i
15     height[v] = h : height of each vertex v is in T_h but not in T_{h+1}
16     depth = depth of the rc-tree (depth = k)
17     '''
18     self.name = name
19     self.maximality = maximality
20     self.V = set(V)
21     self.N = len(V)
22     T_1 = Graph(self.V, name='T_1')
23     T_2 = Graph(set(), name='T_2')
24     self.levels = [T_1, T_1, T_2]
25     self.affected_sets = [set(),set(), set()]
26     self.frontiers = [set(), set(), set()]
27     self.maximal_independents = [set(), set(), set()]
28     self.height = {}
29     self.cluster_type = {} #0,1,2 for final,rake,compress
```

```
30      if self.maximality:
31        self.degree_threshold = {} # smallest positive round that degree becomes <= 2
32      for v in self.V:
33        self.height[v] = 1
34        self.cluster_type[v] = 0
35        if self.maximality:
36          self.degree_threshold[v] = 1 # this is assigned at the time of switch
37      self.depth = 1
38      self.outstanding_changes = False
39      self.update_depth = 1
40      self.RCT = RCForest(self.V, name='RCF')
41      self.disable_compression = False
42
43    def link(self, edges, v = None):
44      if v is not None:
45        edges = [(edges, v)]
46      T_1 = self.levels[1]
47      A_0 = self.affected_sets[0]
48      for (u,v) in edges:
49        T_1.add_edge(u,v)
50        self.outstanding_changes = True
51        A_0.add(u)
52        A_0.add(v)
53
54    def cut(self, edges, v = None):
55      if v is not None:
```

```
56        edges = [(edges, v)]
57      T_1 = self.levels[1]
58      A_0 = self.affected_sets[0]
59      for (u,v) in edges:
60        T_1.del_edge(u,v)
61        self.outstanding_changes = True
62        A_0.add(u)
63        A_0.add(v)
64
65    def __determine_frontiers(self, cur_lev):
66      T_prev = self.levels[cur_lev-1]
67      A_prev = self.affected_sets[cur_lev-1]
68      F_prev = set()
69      for v in A_prev:
70        for u in T_prev.Adj[v]:
71          if u not in A_prev:
72            F_prev.add(v)
73            break
74      self.frontiers[cur_lev-1] = F_prev
75
76    def __carry_affected(self, cur_lev):
77      if cur_lev == 1:
78        return
79      if len(self.levels) == cur_lev:
80        self.levels.append(Graph(set(), name='T_%d'%(cur_lev+1)))
81        self.frontiers.append(set())
```

```
82        self.affected_sets.append(set())
83        self.maximal_independents.append(set())
84
85      T_prev = self.levels[cur_lev-1]
86      T_next = self.levels[cur_lev]
87      A_prev = self.affected_sets[cur_lev-1]
88      F_prev = self.frontiers[cur_lev-1]
89      M_prev = self.maximal_independents[cur_lev-1]
90
91      for v in A_prev:
92        if v in T_next.V:
93          T_next.clear_undirected_edges(v)
94        else:
95          T_next.add_vertex(v)
96
97      for v in A_prev:
98        for u in T_prev.Adj[v]:
99          if u in T_next.V:
100           T_next.add_edge(u,v)
101         else:
102           padj = T_prev.Adj[u] - set([v])
103           if len(padj) == 1:
104             w = padj.pop()
105             T_next.add_edge(w,v)
106
107   def __contract_vertex(self, cur_lev, v):
```

```
    '''__contract_vertex(k,v) contracts the vertex at T_k
    as a side effect it removes the vertex v from
    T_{k+1}, ..., T_{r+1} as well where r = height[v]
    assigns height[v] = cur_lev-1 and cluster_type[v]=0,1,2
    '''
    T_cur = self.levels[cur_lev]
    neig = T_cur.Adj[v]
    self.cluster_type[v] = len(neig)
    T_cur.del_vertex(v)
    if self.height[v] > cur_lev:
      for i in range(cur_lev+1, self.height[v]+1):
        self.levels[i].del_vertex(v)

    self.height[v] = cur_lev-1
    if len(neig) == 2: #shortcut (compressing)
      T_cur.add_edge(*list(neig))

  def __eliminate(self, cur_lev):
    for v in self.maximal_independents[cur_lev-1]:
      self.__contract_vertex(cur_lev, v)

  def __determine_degree_status(self, cur_lev):
    T_prev = self.levels[cur_lev-1]
    T_next = self.levels[cur_lev]
    A_prev = self.affected_sets[cur_lev-1]
    for v in A_prev:
```

```
134        deg_prev = len(T_prev.Adj[v])
135        deg_next = len(T_next.Adj.get(v,[])) # [] is for deleted vertices
136        if (deg_prev > 2 or cur_lev == 1) and deg_next <= 2:
137          self.degree_threshold[v] = cur_lev
138
139    def __mark_affected(self, cur_lev):
140      T_prev = self.levels[cur_lev-1]
141      T_next = self.levels[cur_lev]
142      A_prev = self.affected_sets[cur_lev-1]
143      F_prev = self.frontiers[cur_lev-1]
144      M_prev = self.maximal_independents[cur_lev-1]
145
146      A_next = A_prev - M_prev # uneliminated affected factors from previous round
147
148      if self.maximality:
149        # RULE 2: u becomes affected if (u,v) \in T_next and v \in A_prev-M_prev
150        #             with degree_status(v) = 1, we compute degree_status(v) as
151        #             degree_threshold[v] <= cur_lev
152        for v in A_next & F_prev:
153          if self.degree_threshold[v] <= cur_lev:
154            for u in T_next.Adj[v]:
155              A_next.add(u)
156              #print 'rule 2:', u
157      else:
158        # RULE 2': u becomes affected if (u,v) \in T_next and v \in A_prev-M_prev
159        #             with height[v] == cur_lev ( v was eliminated at this round)
```

```
      for v in A_next & F_prev:
        if self.height[v] <= cur_lev:
          for u in T_next.Adj[v]:
            A_next.add(u)


    # RULE 1: u becomes affected if (u,v) \in T_prev and v \in M_prev
    #         This means v is being eliminated now
    for v in M_prev & F_prev:
      for u in T_prev.Adj[v]:
        A_next.add(u)

    self.affected_sets[cur_lev] = A_next

  def __choose_maximal_independent(self, cur_lev, elimination_set = None):
    if elimination_set is not None:
      self.maximal_independents[cur_lev] = set(elimination_set)
      return
    T_cur = self.levels[cur_lev]
    A_cur = self.affected_sets[cur_lev]
    M_cur = set()
    rakable = []
    compressible = []
    for v in A_cur:
      degree = len(T_cur.Adj[v])
      if degree == 0:
```

```
186         M_cur.add(v)
187       elif degree == 1:
188         rakable.append(v)
189       elif degree == 2:
190         compressible.append(v)
191
192     traverse = rakable + compressible
193     if self.disable_compression:
194       traverse = rakable
195
196     for v in traverse :
197       include_v = True
198       for u in T_cur.Adj[v]:
199         if u in M_cur or (u not in A_cur and self.height[u]==cur_lev):
200           include_v = False
201           break
202       if include_v:
203         M_cur.add(v)
204
205     self.maximal_independents[cur_lev] = M_cur
206
207   def contract_tree(self, cur_lev, elimination_set = None):
208     self.__determine_frontiers(cur_lev)
209     self.__carry_affected(cur_lev)
210     self.__eliminate(cur_lev)
211     if self.maximality:
```

```
212         self.__determine_degree_status(cur_lev)
213       self.__mark_affected(cur_lev)
214       self.__choose_maximal_independent(cur_lev, elimination_set)
215
216     def __prepare_output(self):
217       self.RCT.clear_updates()
218       for cur_lev in range(1,self.update_depth):
219         for v in self.maximal_independents[cur_lev]:
220           self.RCT.push_updates(v, self.height[v], self.cluster_type[v])
221
222       for cur_lev in range(self.update_depth,0,-1):
223         T_cur = self.levels[cur_lev]
224         for v in self.maximal_independents[cur_lev]:
225           padj = list(T_cur.Adj[v])
226           if self.cluster_type[v] == 0:
227             self.RCT.make_root(v)
228           elif self.cluster_type[v] == 1:
229             self.RCT.make_child(padj[0], v)
230           else: # has to be cluster_type = 2
231             if self.height[padj[0]] < self.height[padj[1]]:
232               self.RCT.make_child(padj[0], v)
233             else:
234               self.RCT.make_child(padj[1], v)
235
236     def commit(self, enforce_eliminations = None):
237       elimination_set = None
```

```
    cur_lev = 1
    while len(self.affected_sets[cur_lev - 1]) > 0 :
      if enforce_eliminations is not None:
        elimination_set = enforce_eliminations[cur_lev]
      self.contract_tree(cur_lev, elimination_set)
      cur_lev += 1
    self.update_depth = cur_lev - 1
    if self.levels[cur_lev-1].V == set(): # otherwise we finished early
      self.depth = cur_lev - 2
    self.outstanding_changes = False
    self.affected_sets[0] = set()
    self.frontiers[0] = set()
    self.__prepare_output()

  def touch(self, vertices):
    updates = []
    rup = {}
    for rank in range(self.depth):
      rup[rank] = set()
    for v in vertices:
      rup[self.height[v]-1].add(v)
    for rank in range(self.depth):
      for v in rup[rank]:
        updates.append(v)
        pv = self.RCT.parent(v)
        if pv is not None:
```

```
264            rup[self.height[pv]-1].add(pv)
265
266      self.RCT.clear_updates()
267      for v in updates:
268        self.RCT.push_updates(v, self.height[v], self.cluster_type[v])
269
270    def copy_RCT(self):
271      T = deepcopy(self.levels[0])
272      T.BFS()
273      T.bfs_order.reverse()
274
275      self.RCT.clear_updates()
276      for v in T.bfs_order[:-1]:
277        u = T.parent[v]
278        self.RCT.make_child(u, v)
279        self.height[u] = max(self.height[u], self.height[v]+1)
280        self.cluster_type[v] = 1
281        self.RCT.push_updates(v, self.height[v], self.cluster_type[v])
282
283      root = T.bfs_order[-1]
284      self.RCT.make_root(root)
285      self.cluster_type[root] = 0
286      self.RCT.push_updates(root, self.height[root], self.cluster_type[root])
```

8.3 Base Class for Exact Inference

```python
from rctree.contraction import Contraction
from graph import Graph
import numpy as np

class ExactInference:
  '''Provides methods to compute the exact Inference on graphical models.

  let G = (X+F, E) is the factor graph we are working with.
  Here is how this class is used

  Initial Run

  * initialize the class
  * initialize the factors
  * construct the graph by adding edges via add_nontree_edge(u,v)
  * give a computation tree (spanning tree) via add_tree_edge (u,v)
      Note that add_spanning_tree_edge(u,v) is equivalent to
      add_nontree_edge(u,v) and add_tree_edge(u,v)
  * run update_boundaries()
  * run update_cluster_functions()
  * run MAP_update()

  Dynamic changes and maintainence
  * make changes on the graph by changing factors or edges
  * run MAP_update()
```

```
26    * look MAP_updates variable to see what variables are changed in this round
27    '''
28    def __init__(self, N, num_of_variables):
29      ''' initializes the ExactInference class
30
31      N = the number of variables + factors
32      num_of_variables = number of variables
33
34
35      the module initializes the following variables
36      RCT = the RC-Tree output of the c++ implementation
37      num_of_variables = number of variables, |X|
38      Variables = the dimensions of the variables
39      Factors = List of factors, where each factor is a function
40              (numpy array with the right dimensions)
41      MAP = assignment to variables (MAP configuration)
42      MAP_updates = the set of changes in the MAP configuration
43                    represented as a list of pairs (location, assignment).
44      RCTree_roots = the root of the RC-Trees
45
46      current_time = the counter for update_cluster_functions() call,
47                     each time it's called this counter is increased
48      update_time = the list of times where the cluster_function is updated
49                    initially all the cluster functions are updated at time 1
50                    as we change the data/structure some of the cluster functions
51                    are updated and their update time are going to be modified
```

```
52      lastMAP_time = the last time we called MAP_update(),
53                     this together with update_time list is used to figure
54                     out if a cluster needs to be recomputed.
55      '''
56      self.R = Contraction(range(N))
57      self.RCT = self.R.RCT;
58      self.data_dir = './';
59      self.num_of_variables = num_of_variables
60      self.Variables = np.ones(num_of_variables)  # user modifies later
61      self.Factors = [0] * N                      # user modifies later
62      self.Labels = ['F_%d'%el for el in range(N)]# user modifies later
63      self.MAP = np.zeros(num_of_variables, 'int', 'C') - 1
64      self.MAP_updates = []
65      self.RCTree_roots = self.RCT.roots
66      self.current_time = 0
67      self.cf_update_time = [0] * N
68      self.lastMAP_time = -1
69      self.MAP_num_visits = 0
70      self.MAP_updated_variables = []
71      self.MAP_update_time = np.zeros(num_of_variables, 'int', 'C')
72      self.boundary_edges = []
73      self.boundary_tree_edges = []
74      self.boundary_variables = []
75      self.nonboundary_variables = []
76      self.cluster_function = []
77      self.maximizers = []
```

```
    self.neighbors = []
    self.tree_neighbors = []
    self.cf_var = []
    self.allvar = []
    self.message_downward = {}
    self.subtree_size = np.zeros(N, 'int', 'C')
    self.subtree_cost = np.zeros(N, 'int', 'C')
    self.node_cost = np.zeros(N, 'int', 'C')
    self.FG = Graph(range(N))
    self.ET = Graph(range(N))

    for i in range(N) :
      self.neighbors.append(set([]))
      self.tree_neighbors.append(set([]))
      self.boundary_edges.append(set([]))
      self.boundary_tree_edges.append(set([]))
      self.boundary_variables.append(set([]))
      self.nonboundary_variables.append(set([]))
      self.cluster_function.append(1)
      self.maximizers.append({})
      self.cf_var.append([])
      self.allvar.append([])

  def isVariable(self, i):
    return i < self.num_of_variables

```

```
  def isFactor(self, i):
    return i >= self.num_of_variables

  def add_spanning_tree_edge(self, i, j):
    self.add_tree_edge(i,j)
    self.add_nontree_edge(i,j)

  def remove_spanning_tree_edge(self, i, j):
    self.remove_tree_edge(i,j)
    self.remove_nontree_edge(i,j)

  def add_tree_edge(self, i, j):
    ''' adds (i,j) to the computation tree
    '''
    self.R.link(i,j)
    self.tree_neighbors[i].add(j)
    self.tree_neighbors[j].add(i)
    self.ET.add_edge(i,j)

  def remove_tree_edge(self, i, j):
    ''' removes (i,j) from the computation tree
    '''
    self.R.cut(i,j)
    self.tree_neighbors[i].remove(j)
    self.tree_neighbors[j].remove(i)
    self.ET.del_edge(i,j)
```

```
130
131   def add_nontree_edge(self, i, j):
132     ''' adds (i,j) to the graph
133     '''
134     self.neighbors[i].add(j)
135     self.neighbors[j].add(i)
136     self.FG.add_edge(i,j)
137
138   def remove_nontree_edge(self, i, j):
139     ''' removes (i,j) from the graph
140     '''
141     self.neighbors[i].remove(j)
142     self.neighbors[j].remove(i)
143     self.FG.del_edge(i,j)
144
145   def update_boundaries(self):
146     ''' computes the boundaries
147     '''
148     for rcn in self.upnodes():
149       id = rcn
150       be = set([])
151       for ne in self.neighbors[id]:
152         if ne < id: be.add((ne,id))
153         else : be.add((id,ne))
154
155       allvar_local = set([])
```

```
156        if self.isVariable(id): allvar_local.add(id)
157        else : allvar_local |= self.neighbors[id] #union update
158
159        for ch in self.children(rcn):
160          be.symmetric_difference_update(self.boundary_edges[ch])
161          allvar_local |= self.boundary_variables[ch]
162
163        self.boundary_edges[id] = be
164
165        bv = set([])
166        bte = set([])
167        for edge in be:
168          bv.add(edge[0])
169          if edge[1] in self.tree_neighbors[edge[0]]:
170            bte.add(edge)
171
172        self.boundary_tree_edges[id] = bte
173        self.boundary_variables[id] = bv
174        self.nonboundary_variables[id] = allvar_local - bv #set diff
175        self.cf_var[id] = list(bv)
176        self.cf_var[id].sort()
177        self.allvar[id] = list(allvar_local)
178        self.allvar[id].sort()
179
180
181    def single_message_update(self, msgfrom, towards, curmsg, dependent_msgs):
```

```
    self.message_computation_counter = \
        getattr(self, 'message_computation_counter', 0) + 1
    return None

  def query(self, varid):
    rcnvar = varid

    def orient_downward(edge):
      (u,v) = edge
      uh = self.RCT.height(u)
      vh = self.RCT.height(v)
      if uh > vh :
        return (u, v)
      else:
        return (v, u)

    to_be_called = []
    msg_direction = {}
    rcn = rcnvar
    towards = None
    while rcn is not None:
      rid = rcn
      curmsg = msg_direction.get(rid, (rid,None))
      if curmsg in self.message_downward:
        break
      bted = self.boundary_tree_edges[rid]
```

```
208         rm = []
209         for edge in bted:
210           (mfrom,mto) = orient_downward(edge)
211           if mfrom not in msg_direction:
212             msg_direction[mfrom] = edge
213             rm.append(edge)
214         to_be_called.append([rid, towards, curmsg, rm])
215
216         towards = rid
217         rcn = self.RCT.parent(rcn)
218
219       to_be_called.reverse()
220       #print to_be_called
221       for rid,towards,curmsg,rm in to_be_called[:-1]:
222         self.message_downward[curmsg] = \
223             self.single_message_update(rid, towards, curmsg, rm)
224
225       return self.single_message_update(*to_be_called[-1])
226
227     def update_cluster_functions(self):
228       '''computes all the cluster functions
229       '''
230       self.current_time += 1
231       self.message_downward = {} # query cache is invalid after an update
232       for rcn in self.upnodes():
233         self.single_cf_update(rcn)
```

```
  def uplist(self):
    return self.RCT.updates()

  def upnodes(self):
    return self.RCT.updates()

  def rootnodes(self):
    return self.RCT.roots()

  def children(self, rcnode_ptr):
    return self.RCT.children(rcnode_ptr)
```

8.4 Inference, marginalization and computing MAP configurations

```
from inference import ExactInference, log
from copy import copy
import numpy as np

class SemiRing:
  def __init__(self, join = np.add, proj = np.max):
    self.join = join
    self.proj = proj
    if self.join == np.multiply :
      self.unit = 1.0
      self.invj = np.divide
    elif self.join == np.add :
```

```
13        self.unit = 0.0
14        self.invj = np.subtract
15      else:
16        raise Exception('SemiRing: unhandeled join operation\n')
17
18    def joinCF(self, cf_out, allvar, dims, cf_in, cfVi):
19      '''computes cf_out = cf_out * cf_in where
20
21      allvar = the list of variables involved in both cf_out and cf_in functions
22      dims = the list of dimensions of allvar
23      cfVi = the list of variables that cf_in depends on
24      allvar = the list of variables that cf_out depends on
25
26      to utilize this function allvar has to be computed in advance and cf_out
27      has to be initialized with allvar dimension.
28
29      The cost of the computation is prod(dims)
30      '''
31      if len(allvar) == 0:
32        return
33      reshape_index = []
34      for i,var in zip(range(len(allvar)), allvar):
35        if var in cfVi:
36          reshape_index.append(dims[i])
37        else :
38          reshape_index.append(1)
```

```
39
40      import lib.stat as stat
41      stat.timerstart()
42      cf_in = cf_in.reshape(reshape_index)
43      self.join(cf_out, cf_in, cf_out)
44      stat.timer('cf')
45
46    def projCF(cf, allvar, cfvar):
47      '''computes the output=max_{allvar - cfvar} cf(allvar)
48
49      allvar = the list of variables that cf depends on
50      cfvar = the list of variables that the output depends on
51      cf = the cluster function with allvar variables
52      '''
53      rv = cf
54      lav = len(allvar)
55      maxix = 0
56      for i in range(lav):
57        var = allvar[i]
58        if var in cfvar:
59          maxix += 1
60        else:
61          rv = self.proj(rv, maxix)
62      return rv
63
64  class Factor:
```

```
  def __init__(self, args, func):
    self.varset = set(args) #set
    self.args = args #ordered list
    self.func = func #numpy table

    assert( type(args) is type([]) )

  def isScalar(self):
    return len(self.varset) == 0

  def project(self, AI, known_argument_set):
    args = list(self.varset - known_argument_set)
    args.sort()
    func = self.func
    for i in range(len(self.args)):
      var = self.args[i]
      if var in known_argument_set:
        value = AI.MAP[var]
        func = func.take([value],i)
    return Factor(args, func.squeeze())

  def __repr__(self):
    return "Factor(%s, shape=%s)" % (str(self.args), str(self.func.shape))

class FCollection():
  def __init__(self, AI):
```

```
    self.AI = AI
    self.join_op = AI.semi_ring.join
    self.joinCF = AI.semi_ring.joinCF
    self.proj_op = AI.semi_ring.proj
    self.unit = AI.semi_ring.unit
    self.invj_op = AI.semi_ring.invj

    self.scalar = Factor([], np.zeros(1) + self.unit)
    self.flist = [self.scalar]
    self.allvarset = set([])
  def allvarlist(self):
    avl = list(self.allvarset)
    avl.sort()
    return avl

  def add_factor(self, f):
    self.allvarset |= f.varset
    if f.isScalar():
      self.join_op(self.scalar.func, f.func, self.scalar.func)
    else:
      self.flist.append(f)

  def violating_factor(self, f):
    '''test if f can be added to the collection without violating the dlimit constraint
    '''
    dims = self.AI.Variables.take(list(self.allvarset | f.varset))
```

```
117     cost = dims.prod()
118     return cost > self.AI.dlimit
119
120   def project(self, var):
121     ''' Returns a new collection of factors that contains var
122     '''
123     rv = FCollection(self.AI)
124     excluded = FCollection(self.AI)
125     for f in self.flist :
126       if var in f.varset:
127         rv.add_factor(f)
128       else :
129         excluded.add_factor(f)
130     return (rv, excluded)
131
132   def sort_decreasing(self):
133     a = [(self.AI.Variables.take(f.args).prod(), f.args, f) for f in self.flist]
134     a.sort()
135     a.reverse()
136     self.flist = [el[2] for el in a]
137
138   def partitioning(self):
139     ''' Returns the vlim-partitioning of the fcollection
140     output format is a list of FCollections
141
142     Use First Fit Decreasing Algorithm
```

```
143       '''
144       self.sort_decreasing()
145       rv = []
146       newfl = self.flist
147       while len(newfl) > 0 :
148         fc = FCollection(self.AI)
149         rv.append(fc)
150         oldfl = newfl
151         newfl = []
152         for f in oldfl:
153           if not fc.violating_factor(f):
154             fc.add_factor(f)
155           else:
156             newfl.append(f)
157       return rv
158
159     def join(self):
160       import lib.stat as stat
161       stat.timerstart()
162
163       allv = self.allvarlist()
164       dims = self.AI.Variables.take(allv)
165       rv = np.zeros(dims) + self.unit
166       for f in self.flist:
167         self.joinCF(rv, allv, dims, f.func, f.args)
168
```

```
    stat.timer('join')
    return Factor(allv, rv)

  def proj(self, var):
    ''' returns max_{var} sumation of the factors in the collection
    output format is a Factor
    '''
    sf = self.join()
    ix = sf.args.index(var)
    newset = sf.args
    newset.remove(var)
    newFactor = Factor(newset, self.proj_op(sf.func, ix))
    return newFactor

  def assign_maximizers(self, known_arg_set):
    '''assigns argmax_{unknown} summation of factors projected to known assignments
    modifies AI.MAP and AI.MAP_update_time
    '''
    AI = self.AI
    fc = FCollection(AI)
    for f in self.flist:
      fc.add_factor(f.project(AI, known_arg_set))
    sf = fc.join()
    ix = sf.func.argmax()
    assignment = np.unravel_index(ix, sf.func.shape)
    for var,new_value in zip(sf.args,assignment):
```

```
      AI.assign_MAP_value(var,new_value)

  def __repr__(self):
    output = ["FCollection allvarset=%s, numfactors=%d)" % (str(self.allvarset), len(sel
    for f in self.flist:
      output.append(repr(f))
    return '\n'.join(output)

class ApproximateInference(ExactInference):
  ''' Provides methods to compute the approximate Inference on general graphical models.

  let G = (X+F, E) is the factor graph we are working with. Here is how this class is us

  Initial Run

  * initialize the class
  * initialize the factors
  * construct the graph by adding edges via add_nontree_edge(u,v)
  * give a computation tree (spanning tree) via add_tree_edge (u,v)
      Note that add_spanning_tree_edge(u,v) is equivalent to
      add_nontree_edge(u,v) and add_tree_edge(u,v)
  * run update_boundaries()
  * run update_cluster_functions()
  * run MAP_update()

  Dynamic changes and maintainence
```

```
221     * make changes on the graph by changing factors or edges
222     * run MAP_update()
223     * look MAP_updates variable to see what variables are changed in this round
224     '''
225
226     def __init__(self, N, num_of_variables, computation_limit = 2**20, sumprod=False):
227       ExactInference.__init__(self, N, num_of_variables)
228
229       self.dlimit = computation_limit
230       self.sumprod = sumprod
231
232       if sumprod:
233         semi_ring = SemiRing(np.multiply, np.sum)
234       else:
235         semi_ring = SemiRing(np.add, np.max)
236       self.semi_ring = semi_ring
237       self.Factors = [self.semi_ring.unit] * N
238
239       self.ccol = [None] * N
240       self.Hcol = [None] * N
241       self.MAP_lookup_time = np.zeros(self.num_of_variables, 'int', 'C')
242       self.num_of_approximated_variables = np.zeros(N, 'int', 'C') - 1
243       self.minibuckets_approximated_variables = 0
244       self.energy_ub = 0 # final value obtained after minibucket approximations
245       self.energy_lb = 0 # this is modified through MAP configurations
246
```

```
247      self.marginals = []
248
249      self.MAP_value = self.final_value
250      self.likelihood = self.final_value
251
252    def assign_MAP_value(self, var,new_value):
253      self.MAP_lookup_time[var] = self.current_time
254      if new_value != self.MAP[var]:
255        self.MAP_update_time[var] = self.current_time
256        self.MAP[var] = new_value
257        self.MAP_updated_variables.append(var)
258
259    def MAP_fix_for_single_state_variables(self):
260      for var in range(self.num_of_variables):
261        if self.Variables[var] < 2:
262          self.MAP[var] = 0
263
264    def single_cf_update(self, rcn):
265      '''computes the cluster function of the rc-node rcn
266      '''
267      id = rcn
268      self.cf_update_time[id] = self.current_time
269
270      nbvar_set = self.nonboundary_variables[id]
271      nbvar = list(nbvar_set)
272      nbvar.sort()
```

```
    t = len(nbvar)

    fc = FCollection(self)
    if self.isFactor(id):
      cfVi = list(self.neighbors[id])
      cfVi.sort()
      fc.add_factor(Factor(cfVi, self.Factors[id]))

    for ch in self.children(rcn):
      for f in self.ccol[ch].flist :
        fc.add_factor(f)

    cols = []
    num_of_approximated_variables = 0
    max_cost = 0

    for i,var in zip(range(t),nbvar):

      projected_fc, newfc = fc.project(var)
      par = projected_fc.partitioning()
      cols.append(projected_fc)

      if len(par) > 1: num_of_approximated_variables += 1

      for col in par:
        max_cost = max(max_cost, len(col.allvarset)) # TODO: count dimensions
```

```
299         newFactor = col.proj(var)
300         newfc.add_factor(newFactor)
301
302       fc = newfc
303
304     self.ccol[id] = fc
305     self.Hcol[id] = cols
306     self.node_cost[id] = max_cost
307     self.num_of_approximated_variables[id] = num_of_approximated_variables
308
309   def update_cluster_functions(self):
310     import lib.stat as stat
311     stat.timerstart()
312     ExactInference.update_cluster_functions(self)
313     energy_ub = self.MAP_value()
314     stat.timer('ucf')
315
316   def final_value(self):
317     total_value = np.zeros(1) + self.semi_ring.unit
318     for rcn in self.rootnodes():
319       fc = self.ccol[rcn]
320       if fc is not None:
321         for f in self.ccol[rcn].flist:
322           self.semi_ring.join(total_value, f.func, total_value)
323     return total_value
324
```

```
  def single_message_update(self, msgfrom, towards, curmsg, dependent_msgs):
    rcn = msgfrom

    fc = FCollection(self)
    if self.isFactor(msgfrom):
      cfVi = list(self.neighbors[msgfrom])
      cfVi.sort()
      fc.add_factor(Factor(cfVi, self.Factors[msgfrom]))

    for ch in self.children(rcn):
      cid = ch
      if cid == towards :
        continue
      for f in self.ccol[cid].flist :
        fc.add_factor(f)

    for msg in dependent_msgs:
      for f in self.message_downward[msg].flist :
        fc.add_factor

    if towards is None:
      if self.isFactor(msgfrom):
        node_set = set(self.neighbors[msgfrom])
      else:
        node_set = set([msgfrom])
      msgvar_set = node_set & self.nonboundary_variables[msgfrom]
```

```
351     else :
352       msgvar_set = self.boundary_variables[towards] & \
353           self.nonboundary_variables[msgfrom]
354
355     msgvar = list(msgvar_set)
356     msgvar.sort()
357     t = len(msgvar)
358
359     num_of_approximated_variables = 0
360     for i,var in zip(range(t),msgvar):
361
362       projected_fc, newfc = fc.project(var)
363       par = projected_fc.partitioning()
364
365       if len(par) > 1:
366         num_of_approximated_variables += 1
367
368       for col in par:
369         newFactor = col.proj(var)
370         newfc.add_factor(newFactor)
371
372       fc = newfc
373
374     return fc
375
376   def MAP_update(self) :
```

```
    '''returns a list of variables that have changed in the MAP configuration
    '''
    stack = [el for el in self.rootnodes()]
    self.MAP_updated_variables = []
    counter = 0
    visited = []
    checked = []
    while len(stack) > 0:
      counter += 1
      rcn = stack.pop()
      id = rcn
      visited.append(id)
      need_to_be_updated = self.cf_update_time[id] > self.lastMAP_time
      for var in self.boundary_variables[id]:
        need_to_be_updated |= self.MAP_update_time[var] > self.lastMAP_time

      if (need_to_be_updated) :

        checked.append(id)
        nbvar_set = self.nonboundary_variables[id]
        nbvar = list(nbvar_set)
        nbvar.sort()
        t = len(nbvar)
        cols = self.Hcol[id]

        known_arg_set = self.boundary_variables[id]
```

```
403         for i in range(t-1,-1,-1):
404           known_arg_set = self.boundary_variables[id] | set(nbvar[i+1:])
405           fc = cols[i]
406           fc.assign_maximizers(known_arg_set)
407
408         for ch in self.children(rcn):
409           stack.append(ch)
410
411     self.lastMAP_time = self.current_time
412     self.MAP_num_visits = counter
```

CHAPTER 9

CONCLUDING REMARKS

In this thesis, we have presented an adaptive framework for performing exact inference that efficiently handles changes to the input factor graph and its associated elimination tree. Our approach to adaptive inference requires a linear preprocessing step in which we construct a cluster-tree data structure by performing a generalized factor elimination; the cluster tree offers a logarithmic depth balanced representation of an elimination tree annotated with certain statistics. We can then make arbitrary changes to the factor graph or elimination tree, and update the cluster tree in logarithmic time in the size of the input factor graph. Moreover, we can also calculate any particular marginal in time that is logarithmic in the size of the input graph, and update MAP configurations in time that is roughly proportional to the number of entries in the MAP configuration that are changed by the update.

As with all methods for exact inference, our algorithms carry a constant factor that is exponential in the width of the input elimination tree. Compared to traditional methods, this constant factor is larger for adaptive inference; however the running time of critical operations are logarithmic, rather than linear, in the size of the graph in the common case. In our experiments, we establish that for any fixed treewidth and variable dimension, adaptive inference is preferable as long as the input graph is sufficiently large. For reasonable values of these input parameters, our experimental evaluation shows that adaptive inference can offer a substantial speedup over traditional methods. Moreover, we validate our algorithm using two real-world computational biology applications concerned with sequence and structure variation in proteins.

We have then applied this framework to dual-decomposition solvers in a way that it balances the intrinsic trade-offs between parallelism and convergence rate. For the choice of subproblems, we use a cover tree to obtain rapid convergence, but use a balanced *cluster tree* data structure to enable efficient subgradient updates. In addition to the benefits of

adaptivity, the cluster tree is also amenable to a high degree of parallelism. Moreover, it can be used to efficiently update optimal configurations during each subgradient iteration. We randomly generated models and demonstrated that our approach is up to two orders of magnitude faster than other methods as the model size becomes large. We also show that for the real-world problem of stereo matching, our approach is roughly twice as fast as other methods.

At a high level, our cluster-tree data structure is a replacement for the junction tree in the typical sum-product algorithm. A natural question, then, is whether our data structure, can be extended to be used as subroutine for approximate inference techniques other than dual-decomposition. The approach does appear to be amenable to methods that rely on approximate elimination (e.g., [18]), since these approximations can be incorporated into the cluster functions in the cluster tree. Iterative approximate methods that use fixed point updates (e.g., [52, 53, 58]), however, may be more difficult, since they often make a large number of changes to messages in each successive iteration.

Another interesting direction is to tune the cluster tree construction based on computational concerns. While deferred factor elimination gives rise to a balanced elimination tree, it also incurs a larger constant factor dependent on the tree width. Our benchmarks show that this overhead can be pessimistic, but it is also possible to tune the number of deferred factor eliminations performed, at the expense of increasing the depth of the resulting cluster tree. It would be interesting to incorporate additional information into the deferred elimination procedure used to build the cluster tree to reduce this constant factor. For example, we can avoid creating a cluster function if its run-time complexity is high (e.g., it dimension or the domain sizes of its variables are large), preferring instead a cluster tree that has a greater depth but will yield overall lower costs for queries and updates.

REFERENCES

[1] U. Acar, A. T. Ihler, R. R. Mettu, and Ö. Sümer. Adaptive Bayesian inference. In *Advances in Neural Information Processing Systems 20*. MIT Press, 2008.

[2] U. A. Acar. *Self-Adjusting Computation*. PhD thesis, Department of Computer Science, Carnegie Mellon University, May 2005.

[3] U. A. Acar, G. Blelloch, R. Harper, J. Vittes, and M. Woo. Dynamizing static algorithms with applications to dynamic trees and history independence. In *ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 2004.

[4] U. A. Acar, G. E. Blelloch, K. Tangwongsan, and D. Turkoglu. Robust kinetic convex hulls in 3d. In *European Symposium on Algorithms (ESA)*, 2008.

[5] U. A. Acar, G. E. Blelloch, and J. L. Vittes. An experimental analysis of change propagation in dynamic trees. In *Workshop on Algorithm Engineering and Experimentation*, 2005.

[6] U. A. Acar, A. Cotter, B. Hudson, and D. Turkoglu. Dynamic well-spaced point sets. In *ACM Symposium on Computational Geometry (SCG)*, 2010.

[7] U. A. Acar, B. Hudson, and D. Turkoglu. Kinetic mesh refinement in 2d. In *ACM Symposium on Computational Geometry (SCG)*, 2011.

[8] U. A. Acar, A. T. Ihler, R. R. Mettu, and Ö. Sümer. Adaptive Bayesian inference in general graphs. In *Proceedings of the 24th Annual Conference on Uncertainty in Artificial Intelligence*, pages 1–8, 2008.

[9] U. A. Acar, A. T. Ihler, R. R. Mettu, and Ö. Sümer. Adaptive updates for maintaining MAP configurations with applications to bioinformatics. In *Proceedings of the IEEE Workshop on Statistical Signal Processing*, pages 413–416, 2009.

[10] R. Achanta, A. Shaji, K. Smith, A. Lucchi, P. Fua, and S. Süsstrunk. SLIC Superpixels. Technical report, EPFL, 2010.

[11] P. Bhatotia, A. Wieder, R. Rodrigues, U. A. Acar, and R. Pasquini. Incoop: Mapreduce for incremental computations. In *ACM Symposium on Cloud Computing (SoCC)*, 2011.

[12] A. A. Canutescu, A. A. Shelenkov, and R. L. Dunbrack Jr. A graph-theory algorithm for rapid protein side-chain prediction. *Protein Sci*, 12(9):2001–2014, Sep 2003.

[13] W. Chu, Z. Ghahramani, and D. Wild. A graphical model for protein secondary structure prediction. In *Proc. 21st International Conference on Machine Learning*, 2004.

[14] G. F. Cooper. The computational complexity of probabilistic inference using bayesian belief networks. *Artificial Intelligence*, 42(2-3):393 – 405, 1990.

[15] B. D'Ambrosio. Parallelizing probabilistic inference: Some early explorations. In *UAI*, pages 59–66, 1992.

[16] A. Darwiche. *Modeling and Reasoning with Bayesian Networks*. Cambridge, 1st edition, 2009.

[17] A. Darwiche and M. Hopkins. Using recursive decomposition to construct elimination orders, jointrees, and dtrees. In *Trends in Artificial Intelligence, Lecture Notes in AI*, pages 180–191. Springer-Verlag, 2001.

[18] R. Dechter. Bucket elimination: A unifying framework for probabilistic inference. In M. I. Jordan, editor, *Learning in Graphical Models*, pages 75–104. MIT Press, 1998.

[19] A. L. Delcher, A. J. Grove, S. Kasif, and J. Pearl. Logarithmic-time updates and queries in probabilistic networks. *JAIR*, 4:37–59, 1995.

[20] R. L. Dunbrack Jr. Rotamer libraries in the 21st century. *Curr Opin Struct Biol*, 12(4):431–440, 2002.

[21] D. Frigioni, A. Marchetti-Spaccamela, and U. Nanni. Fully dynamic algorithms for maintaining shortest paths trees. *Journal of Algorithms*, 34:251 – 281, 2000.

[22] D. Frishman and P. Argos. Knowledge-based protein secondary structure assignment. *Proteins: Structure, Function and Genetics*, 23:566–579, 1995.

[23] A. Globerson and T. Jaakkola. Fixing max-product: Convergent message passing algorithms for MAP LP-relaxations. In *Advances in Neural Information Processing Systems (NIPS)*, 2007.

[24] Matthew A. Hammer, Umut A. Acar, and Yan Chen. CEAL: a C-based language for self-adjusting computation. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 2009.

[25] L. Hong and G. Chen. Segment-based stereo matching using graph cuts. In *Proceedings of the 2004 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'04)*, pages 74–81, 2004.

[26] J. K. Johnson, D. M. Malioutov, and A. S. Willsky. Lagrangian relaxation for MAP estimation in graphical models. In *In Allerton Conference Communication, Control and Computing*, 2007.

[27] V. Jojic, S. Gould, and D. Koller. Fast and smooth: Accelerated dual decomposition for MAP inference. In *Proceedings of International Conference on Machine Learning (ICML)*, 2010.

[28] W. Kabsch and C. Sander. Dictionary of protein secondary structure: pattern recognition of hydrogen-bonded and geometrical features. *Biopolymers*, 22(12):2577–2637, 1983.

[29] H. Kamisetty, E. P. Xing, and C. J. Langmead. Free energy estimates of all-atom protein structures using generalized belief propagation. In *Proc. 11th Ann. Int'l Conf. Research in Computational Molecular Biology*, pages 366–380, 2007.

[30] K. Kask, R. Dechter, J. Larrosa, and A. Dechter. Unifying cluster-tree decompositions for reasoning in graphical models. *Artificial Intelligence*, 166:165–193, 2005.

[31] S. Koenig, M. Likhachev, Y. Liu, and David Furcy. Incremental heuristic search in artificial intelligence. *Artificial Intelligence Magazine*, 25:99–112, 2004.

[32] N. Komodakis, N. Paragios, and G. Tziritas. MRF optimization via dual decomposition: Message-passing revisited. In *IEEE International Conference on Computer Vision (ICCV)*, 2007.

[33] A. V. Kozlov and J. P. Singh. A parallel lauritzen-spiegelhalter algorithm for probabilistic inference. In *In Proceedings of Supercomputing*, pages 59–66, Washington, DC., 1994.

[34] F. Kschischang, B. Frey, and H.-A. Loeliger. Factor graphs and the sum-product algorithm. *IEEE Trans. Inform. Theory*, 47(2):498–519, February 2001.

[35] S. Lauritzen and D. Spiegelhalter. Local computations with probabilities on graphical structures and their applications to expert systems. *J. Royal Stat. Society, Ser. B*, 50:157–224, 1988.

[36] C. E. Leiserson. The cilk++ concurrency platform. In *ACM Design Automation Conf.*, pages 522–527, 2009.

[37] R. Ley-Wild, M. Fluet, and U. A. Acar. Compiling self-adjusting programs with continuations. In *Proceedings of the International Conference on Functional Programming*, 2008.

[38] J. Martin, J.-F. Gibrat, and F. Rodolphe. Choosing the optimal hidden Markov model for secondary-structure prediction. *IEEE Intelligent Systems*, 20(6):19–25, 2005.

[39] G. L. Miller and J. H. Reif. Parallel tree contraction and its application. In *Proceedings of 26th IEEE Symposium on Foundations of Computer Science*, pages 487–489, 1985.

[40] V. Namasivayam, A. Pathak, and V. Prasanna. Scalable parallel implementation of bayesian network to junction tree conversion for exact inference. In *Information Retrieval: Data Structures and Algorithms*, pages 167–176. PTR, 2006.

[41] J. Pearl. *Probabilistic Reasoning in Intelligent Systems*. Morgan Kaufman, San Mateo, 1988.

[42] D. M. Pennock. Logarithmic time parallel Bayesian inference. In *Proc. 14th Annual Conf. on Uncertainty in Artificial Intelligence*, pages 431–438, 1998.

[43] G. Ramalingam and T. Reps. An incremental algorithm for a generalization of the shortest-path problem. *Journal of Algorithms*, 21:267 – 305, 1996.

[44] J. H. Reif and S. R. Tate. Dynamic parallel tree contraction. In *Proceedings of 5th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 114–121, 1994.

[45] D. Scharstein. High-accuracy stereo depth maps using structured light. In *Proceedings of the 2003 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'03)*, pages 195–202, 2003.

[46] D. Scharstein and R. Szeliski. A taxonomy and evaluation of dense two-frame stereo correspondence algorithms. *International Journal of Computer Vision*, 47:7–42, 2001.

[47] Ö Sümer, U. A. Acar, A. T. Ihler, and R. R. Mettu. Adaptive exact inference in graphical models. *Journal of Machine Learning Research*, 12:3147 – 3186, 2011.

[48] Özgür Sümer, Umut A. Acar, Alexander T. Ihler, and Ramgopal R. Mettu. Fast parallel and adaptive updates for dual-decomposition solvers. In *AAAI*, 2011.

[49] R. Szeliski, R. Zabih, D. Scharstein, O. Veksler, V. Kolmogorov, A. Agarwala, M. Tappen, and C. Rother. A comparative study of energy minimization methods for Markov random fields with smoothness-based priors. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 30:1068–1080, 2008.

[50] H. Trinh. Efficient stereo algorithm using multiscale belief propagation on segmented images. In *British Machine Vision Conference*, 2008.

[51] O. Veksler. Stereo correspondence by dynamic programming on a tree. In *Proceedings of the 2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'05) - Volume 2 - Volume 02*, CVPR '05, pages 384–390, 2005.

[52] M. Wainwright, T. Jaakkola, and A. Willsky. MAP estimation via agreement on (hyper)trees: message-passing and linear programming approaches. *IEEE Trans Info Theory*, 51(11):3697–3717, 2005.

[53] M. J. Wainwright, T. S. Jaakkola, and A. S. Willsky. A new class of upper bounds on the log partition function. *IEEE Trans. Info. Theory*, 51(7):2313–2335, July 2005.

[54] S. J. Weiner, P.A. Kollman, D.A. Case, U.C. Singh, G. Alagona, S. Profeta Jr., and P. Weiner. A new force field for the molecular mechanical simulation of nucleic acids and proteins. *J. Am. Chem. Soc.*, 106:765–784, 1984.

[55] Y. Xia and V. K. Prasanna. Junction tree decomposition for parallel exact inference. In *IEEE International Parallel and Distributed Preocessing Symposium*, pages 1–12, 2008.

[56] C. Yanover and Y. Weiss. Approximate inference and protein folding. In *Proceedings of Neural Information Processing Systems (NIPS)*, pages 84–86, 2002.

[57] J. Yarkony, C. Fowlkes, and A. Ihler. Covering trees and lower-bounds on quadratic assignment. In *Computer Vision and Pattern Recognition (CVPR)*, pages 887–894, june 2010.

[58] J. S. Yedidia, W. T. Freeman, and Y. Weiss. Constructing free energy approximations and generalized belief propagation algorithms. Technical Report 2004-040, MERL, May 2004.