Journal of Machine Learning Research 12 (2011) 3147-3186 Submitted 9/10; Revised 5/11; Published 11/11

Adaptive Exact Inference in Graphical Models

Özgür Sümer OSUMER@CS.UCHICAGO.EDU
Department of Computer Science
University of Chicago
1100 E. 58th Street
Chicago, IL 60637, USA

Umut A. Acar UMUT@MPI-SWS.ORG
Max-Planck Institute for Software Systems
MPI-SWS Campus E 1 4
D-66123 Saarbruecken, Germany

Alexander T. Ihler IHLER@ICS.UCI.EDU
Donald Bren School of Information and Computer Science
University of California, Irvine
Irvine, CA 92697 USA

Ramgopal R. Mettu METTU@ECS.UMASS.EDU
Electrical and Computer Engineering Department
University of Massachusetts, Amherst
151 Holdsworth Way
Amherst, MA 01003, USA

Editor: Neil Lawrence

Abstract

Many algorithms and applications involve repeatedly solving variations of the same inference problem, for example to introduce new evidence to the model or to change conditional dependencies. As the model is updated, the goal of *adaptive inference* is to take advantage of previously computed quantities to perform inference more rapidly than from scratch. In this paper, we 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. After a linear time preprocessing step, our approach enables updates to the model and the computation of any marginal in time that is logarithmic in the size of the input model. Moreover, in contrast to max-product 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. To evaluate the practical effectiveness of our algorithms, we implement and test them using synthetic data as well as for two real-world computational biology applications. Our experiments show that adaptive inference can achieve substantial speedups over performing complete inference as the model undergoes small changes over time.

Keywords: exact inference, factor graphs, factor elimination, marginalization, dynamic programming, MAP computation, model updates, parallel tree contraction

1. Introduction

Graphical models provide a rich framework for describing structure within a probability distribution, and have proven to be useful in numerous application areas such as computational biology, statistical physics, and computer vision. Considerable efforts have been made to understand and minimize the computational complexity of inferring the marginal probabilities or most likely state of a graphical model. However, in many applications we may need to perform repeated computations over a collection of very similar models. For example, hidden Markov models are commonly used for sequence analysis of DNA, RNA and proteins, while protein structure requires the definition of a factor graph defined by the three-dimensional topology of the protein of interest. For both of these types of models, it is often desirable to study the effects of mutation on functional or structural properties of the gene or protein. In this setting, each putative mutation gives rise to a new problem that is nearly identical to the previously solved problem.

The changes described in the examples above can, of course, be handled by incorporating them into the model and then performing inference from scratch. 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. *Adaptive inference* refers to the problem of handling changes to the model (e.g., to model parameters and even dependency structure) more efficiently than performing inference from scratch. Performing inference in an adaptive manner requires a new algorithmic approach, since it requires us to balance the computational cost of the inference procedure with the reusability of its calculations. As a simple example, suppose that we wish to compute the marginal distribution of a leaf node in a Markov chain with n variables. Using the standard sum-product 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 such a setting, it is worth using 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.

In this paper, we focus on developing efficient algorithms for performing exact inference in the adaptive setting. Specifically, we present techniques for two basic inference tasks in general graphical models: marginalization and finding maximum *a posteriori* (MAP) configurations. 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 (Darwiche, 2009) 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. Furthermore, we show that after such updates, the time necessary to compute marginal distributions is logarithmic in the size of the model, and the time to update a MAP configuration is roughly proportional to the number of variables whose values have changed.

1.1 Related Work

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 algorithms (Koenig et al., 2004), in which solutions can be efficiently updated by reusing previously searched parts of the solution space. The problem of performing adaptive inference in graphical models was first considered by Delcher et al. (1995). In their work, they introduced a logarithmic time method for updating marginals under changes to observed variables in the model. Their algorithm relies on the input model being tree-structured, and can only handle changes to observations in the input 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. However, their algorithm addresses only updates to "observations" in the model, and cannot update dependencies in the input model. Additionally, 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. Adaptive exact inference using graph-cut techniques has also been studied by Kohli and Torr (2007). Although the running time of their method does not depend on the tree-width of the input model, it is restricted to pairwise models with binary variables or with submodular pairwise factors. Adaptivity for approximate inference has also been studied by Komodakis et al. (2008); in this work, adaptivity is achieved by performing "warm starts". That is, a change to model is simply made at the final iteration of approximate inference and the algorithm is restarted from this state and allowed to continue until convergence.

The preprocessing technique used by Delcher et al. (1995) is inspired by a method known as *parallel tree contraction*, devised by Miller and Reif (1985) to evaluate expressions on parallel architectures. In parallel tree contraction we must evaluate a given expression tree, where internal nodes are arithmetic operations and leaves are input values. The parallel algorithm of Miller and Reif (1985) 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, and it has also been used to develop efficient parallel implementations of inference (Pennock, 1998; Namasivayam et al., 2006; Xia and Prasanna, 2008).

An interesting property of tree contraction is that it can also be made to be *adaptive* to changes in the input (Acar et al., 2004, 2005). In particular, the techniques of self-adjusting computation (Acar, 2005; Acar et al., 2006, 2009a; Hammer et al., 2009) show that tree contraction can, for example, be used to derive an efficient and reasonably general data structure for dynamic trees (Sleator and Tarjan, 1983). In this paper we apply similar techniques to develop a new algorithm for adaptive inference that can handle arbitrary changes to the input model and can be used for both marginalization and for computing MAP configurations.

1.2 Contributions

In this paper, we present a new framework for adaptive exact inference, building upon the work of Delcher et al. (1995). 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. Our framework for adaptive inference requires a preprocessing step in which we build a balanced representation of the input elimination tree in $O(d^{3w}n)$ time where w is the width of the input elimination tree T. We show that this balanced representation, which we call a *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.

As in standard approaches for exact inference in general graphs, our algorithm has an exponential dependence on the tree-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.

Since d and w can often be bounded by reasonably small constant factors, we know that there exists some n beyond which we would achieve speedups, but where exactly the speedups materialize is important in practice. To evaluate the practical effectiveness of our approach, we implement the proposed algorithms and present an experimental evaluation by considering both synthetic data (Section 6.1) and real data (Sections 6.2 and 6.3). 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. Given that the asymptotic difference between linear and logarithmic run-times can be large, it is not surprising that our approach yields speedups for large models. The reason for the observed speedups in the smaller graphs is due to the fact that constant factors hidden by the asymptotic bounds associated with the exponential bounds are small (because they involve fast floating point operations) and because our worst-case bounds are often not attained for relatively small graphs (Section 6.1.5).

In addition, we also show the applicability of our framework to two problems in computational structural biology (Sections 6.2 and 6.3). 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

(Canutescu et al., 2003), our algorithm is nearly 7 times faster than computing minimum-energy conformations from scratch.

Several elements of this work have appeared previously in conference versions (Acar et al., 2007, 2008, 2009b). In this paper we unify these into a single framework and improve our algorithms and our bounds in several ways. Specifically, we present deterministic versions of the algorithms, including a key update algorithm and its proof of correctness; we derive upper bounds in terms of the tree-width, the size of the model, and the domain size; and we give a detailed experimental analysis.

1.3 Outline

The remainder of the paper is organized as follows. In Section 2, we give the definitions and notation used throughout this paper, along with some background on the factor elimination algorithm and tree decompositions. In Section 3, we describe our algorithm and the cluster tree data structure and how they can be used for marginalization. Then, in Section 4, we describe how updates to the underlying model can be performed efficiently. In Section 5, we extend our algorithm to compute and maintain MAP configurations under model changes. In Section 6, we show experimental results for our approach on three synthetic benchmarks and two applications in computational biology. We conclude with a discussion of future directions in Section 7.

2. Background

Factor graphs (Kschischang et al., 2001) 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 1a). We denote the adjacency relationship in graph G by $\sim_G$, and let $X_{f_j}=\left\{x_i\in X : x_i{\sim_G}f_j\right\}$ be the set of variables adjacent to factor f_j. For example, in Figure 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 paper 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 the configuration of the variables that maximizes g, that is,

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

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

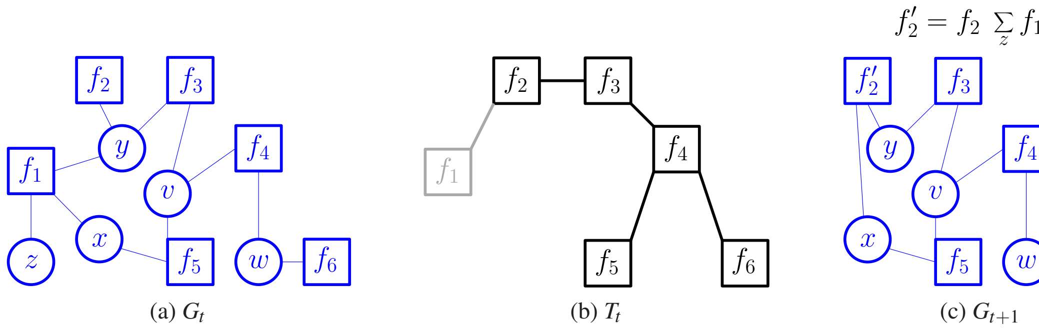


Figure 1: *Factor elimination.* Factor elimination takes a factor graph G_1 and an elimination tree T_1 as input and sequentially eliminates the leaf factors in the elimination tree. As an example, to eliminate f_1 in iteration t, we first marginalize out any variables that are only adjacent to the eliminated factor, and then propagate this information to the unique neighbor in T_t, that is, $f_2' = f_2 \sum_z f_1$.

2.1 Factor Elimination

There are various essentially equivalent algorithms proposed for solving marginalization problems, including belief propagation (Pearl, 1988) or sum-product (Kschischang et al., 2001) for tree-structured graphs, or more generally bucket elimination (Dechter, 1998), recursive conditioning (Darwiche and Hopkins, 2001), junction-trees (Lauritzen and Spiegelhalter, 1988) and factor elimination (Darwiche, 2009). 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 1b.

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

$$\lambda_j = \sum_{\mathcal{V}_j} f_j, \qquad\qquad f_k' = f_k \lambda_j.$$

For reasons that will be explained in Section 3.1, 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}, that is, $X_{f_k'} := X_{f_k} \cup X_{f_j} \setminus \mathcal{V}_j$. Figure 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., $\mathcal{V}_1 = \{z\}$)

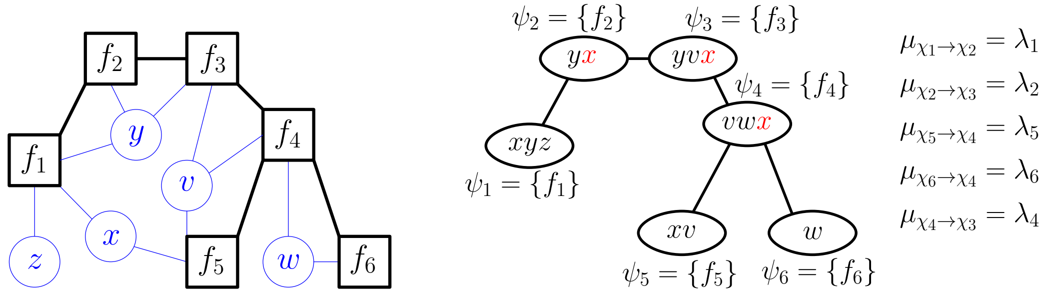


Figure 2: *Factor trees and tree decompositions.* A tree-decomposition (right) that is equivalent to a given elimination tree (left) can be obtained by first replacing each factor with a hyper-node that contains the variables adjacent to that factor node and then adding variables to the hyper-nodes so that the running intersection property is satisfied.

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 at a factor f_j such that $x_i \sim_G f_j$, 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 (Kask et al., 2005; Darwiche, 2009) 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.

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 Darwiche and Hopkins, 2001 and Kask et al., 2005), 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 $\chi = \left\{X_{f_1}, X_{f_2}, \ldots, X_{f_m}\right\}$ 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 2. The factor graph (light 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 $\psi_j = \left\{f_j\right\}$ 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,

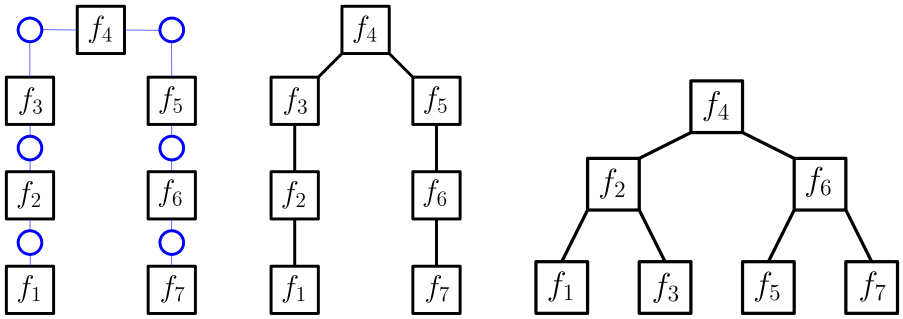
$$\mu_{\chi_i \to \chi_j} = \sum_{\chi_j \setminus \chi_i} f_i \prod_{(\chi_k, \chi_i) \in E' \setminus \left\{(\chi_j, \chi_i)\right\}} \mu_{\chi_k \to \chi_i}. \tag{1}$$

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 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 paper, 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.

3. Computing Marginals with Deferred Factor Elimination

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



(a) Chain graph (b) Unbalanced elimination tree (c) Balanced elimination tree

Figure 3: *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.

for repeated inference tasks. For example, an HMM typically used for sequence analysis yields a chain-structured factor graph as shown in Figure 3a. The obvious elimination tree for this graph is also chain-structured (Figure 3b). 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 3c, 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. Pseudocode is given in Figure 4.

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 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 9 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

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: *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.

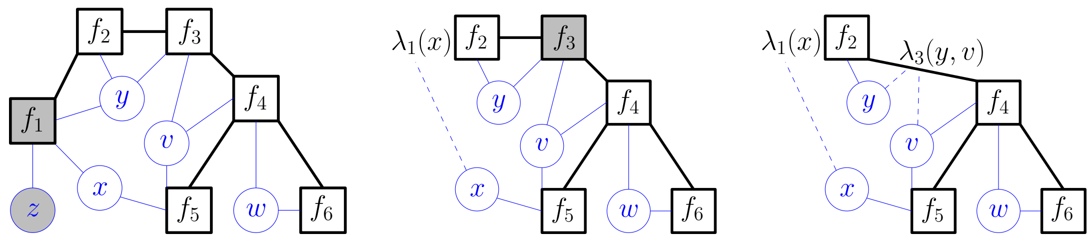
updated. Thus, in the remainder of the paper we separate the discussion of updates applied to the input model from updates that are applied to the input elimination tree. As we will see in Section 4, the former prove to be relatively easy to deal with, while the latter require a reorganization of the cluster tree data structure.

3.1 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 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-one (leaf) factor f_j, we attach λ_j to the neighbor vertex f_k. When a degree-two 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



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

Figure 5: *Deferred factor elimination.* (a) An elimination tree T_1 (bold elges), with variable dependencies shown with light edges 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-two node f_3, we replace it with λ_3 attached to the edge (f_2, f_4), giving tree T_3 shown in (c).

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 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

$$\lambda_j = \sum_{\mathcal{V}_j} f_j \prod_{\lambda_k \in \mathcal{C}_{T_t}(f_j)} \lambda_k. \tag{2}$$

The cluster λ_j is referred as a *root cluster* if $\deg_{T_t}(f_j) = 0$, a *degree-one cluster* if $\deg_{T_t}(f_j) = 1$, and a *degree-two cluster* if $\deg_{T_t}(f_j) = 2$. Figure 5 illustrates the creation of degree-one and degree-two clusters, and the associated changes to the elimination tree and factor graph. We first eliminate f_1 by replacing it with degree-one 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-two factor f_3 by replacing it with degree-two 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 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-two factors are also eliminated. As discussed earlier in Section 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-one 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 shown in Section 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 5, $\lambda_1(x) = \mu_{\chi_1 \to \chi_2}(x)$).

A degree-two cluster λ_j produced after eliminating a degree-two 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 3.2). 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 $\mu_{\chi_j \to \chi_c}$, that is, $\mu_{\chi_d \to \chi_k} = \sum_{\chi_k \setminus \chi_j} \lambda_i \mu_{\chi_j \to \chi_c}$. These intermediate functions are in fact the mechanism that allows us avoid long sequences of message passing. For example in Figure 5, λ_3 can be used to compute the message $\mu_{\chi_3 \to \chi_4}$ using only $\mu_{\chi_2 \to \chi_3}$, that is, $\mu_{\chi_3 \to \chi_4}(x, v) = \sum_y \mu_{\chi_2 \to \chi_3}(x, y) \lambda_3(y, v)$.

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 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-two vertices takes $\Theta(d^{3w})$ time and produces a cluster of size $\Theta(d^{2w})$.*

Proof Each degree-one cluster has size $O(d^w)$ because it is equal to a sum-product message in the equivalent tree-decomposition. For a degree-two 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-one clusters of f_j, their argument set is a subset of χ_j, so the product in Equation (2) 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 (2) becomes $O(d^{2w})$. This bound is achieved when there is a degree-two cluster and χ_j and χ_k are disjoint.

When a degree-two 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-one clusters is a subset of χ_j. This cluster can have degree-two clusters on edges (f_i, f_j) and (f_j, f_k), and in this case, computation of a degree-two 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 Equation (2) is bounded, that is, for any factor f, $|C_T(f)| = O(1)$. 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 (2) 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,

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

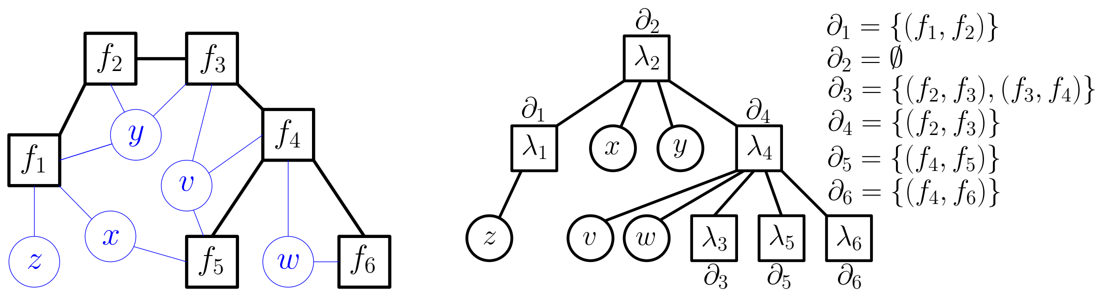
we can bring the complexity of computing Equation (2) down to $O(d^w)$ for each factor. ■

3.2 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 6). 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 (2). 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 (2), 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



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

Figure 7: *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.

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 7b. 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 2 *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 1, each such elimination takes $O(d^{3w})$ time. ■

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-one 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 hierarchical clustering process will prove to be useful in Section 4, 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 7a. 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 7b.

As we show with the following lemma, the cluster tree that results from hierarchical 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($\mathcal{H}$, x_i)

Let $x_i, \lambda_1, \ldots, \lambda_k$ be the path from x_i to the root λ_k of cluster tree $\mathcal{H}$
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 8: *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 3 *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-two nodes and c degree-3 or more nodes, that is, $m = a + b + c$ 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-one and degree-two 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-two 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 $O(\log n)$. ∎

3.3 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 8 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-one 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

$$M_{f_j} = \sum_{Y \setminus X_{\lambda_j}} f_j M_{f_a} M_{f_b} \prod_{f \in C_j \setminus \{f_{j-1}\}} f, \tag{3}$$

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

$$g^i(x_i) = \sum_{Y \setminus \{x_i\}} M_{f_1} \prod_{f \in C_1} f \tag{4}$$

where Y is the set of variables that appear in the summands. Combining this approach with Lemmas 1 and 3, we have the following theorem.

Theorem 4 *Consider a factor graph G with n nodes and let T be an elimination tree with width w. Then, Equation (4) holds for any variable x_i and can be computed in $O(d^{2w} \log n)$ time.*

Proof The correctness of Equation (4) 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-two clusters in $C_j \setminus \{f_{j-1}\}$, 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 (1) 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 1. ■

We note that it is also possible to speed-up successive marginal queries by caching the downward marginalization functions in Equation (3). 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 marginalization 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 4.1, 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.

4. Updates

The preceding sections 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 section, 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

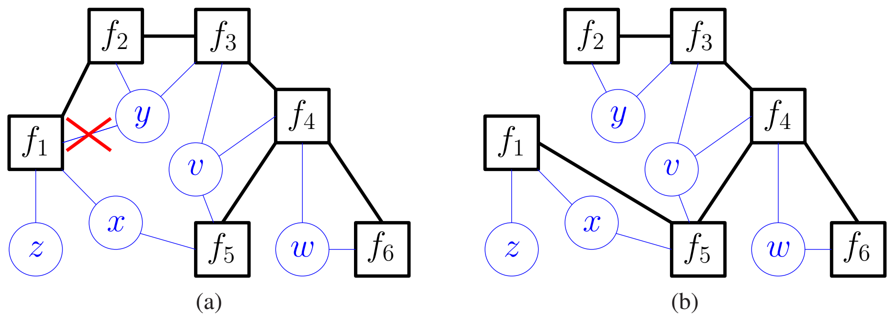


Figure 9: *Modifying the Elimination Tree.* If the factor graph in (a) is modified by removing the edge (y, f_1), we can reduce the width of the elimination tree (from 3 to 2) by replacing the edge (f_1, f_2) by (f_1, f_5).

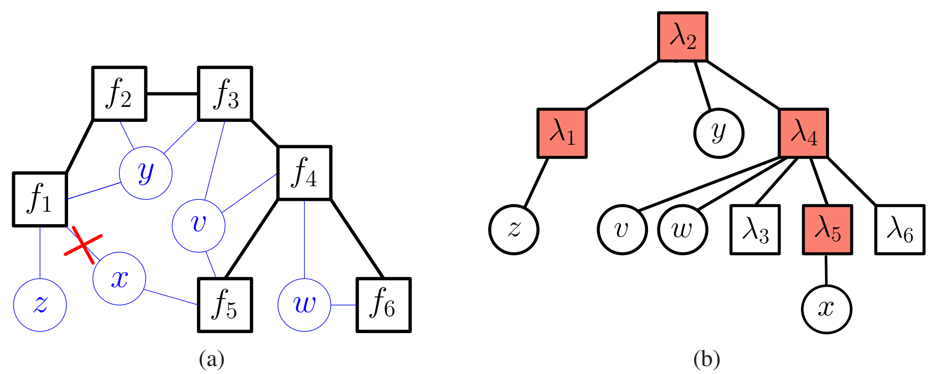


Figure 10: *Modifying the arguments of factors.* If the factor graph in (a) is modified by removing the edge (x, f_1), we update two paths in the cluster tree, as shown in (b), from both x and λ_1 to the root. The position in which x is eliminated is found by bottom-up traversing of the factors adjacent to x.

elimination tree upon changes to factors. Figure 9 illustrates such an example, in which changing a dependency in the factor graph makes it possible to reduce the width of the elimination tree.

4.1 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 Section 3, 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 cluster tree. Since the cluster tree is of depth $O(\log n)$ by Lemma 3, 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

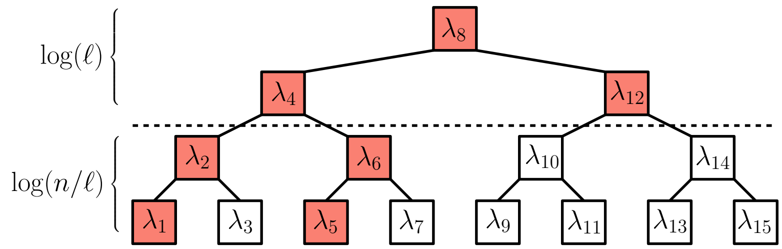


Figure 11: *Batch updates.* After modifying $\ell = 3$ factors, f_1, f_5 and f_{12}, we update the corresponding clusters and their ancestors in a bottom-up fashion. The total number of nodes visited is $O(\ell \log(\frac{n}{\ell}) + 2^{\log(\ell)}) = O(\ell \log(\frac{n}{\ell}))$.

also change. Since x_i is eliminated (i.e., 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 10 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 upwards through the cluster tree. This situation is illustrated in Figure 11. 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,

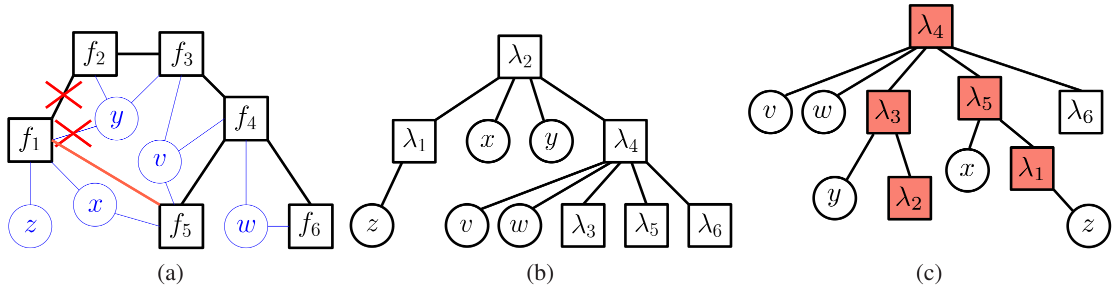


Figure 12: *Updating the elimination tree.* Suppose we modify the input factor graph by removing (y, f_1) from the factor graph and replacing (f_1, f_2) by (f_1, f_5) in the elimination tree as shown in (a). The original cluster tree (b) must be changed to reflect these changes. We must revisit the decisions made during the hierarchical clustering for the affected factors (shaded).

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.

Theorem 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 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.*

4.2 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 12 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 Section 3; 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.

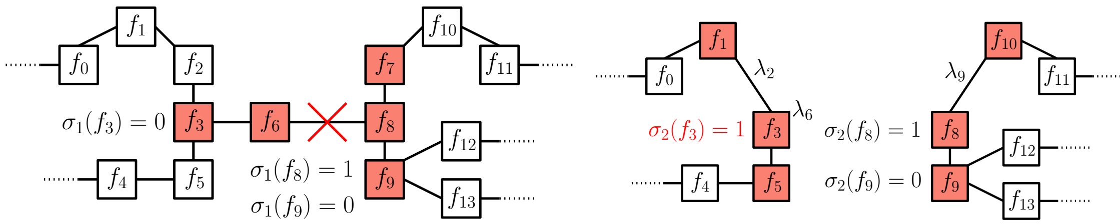
Observe that, to update any particular round of the hierarchical clustering, for any factor f_k we must be able to efficiently determine whether its associated cluster must be recomputed due to the insertion or deletion of an edge (f_i, f_j). A trivial way to check this would be to compute a new hierarchical 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 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. An example is shown in Figure 13. 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 two 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:



(a) Round 1 (b) Round 2

Figure 13: *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.

- 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 one 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 zero, 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 6 *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) cannot 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) cannot 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 $f \in P_i$ implies 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 zero 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, so 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 zero, 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 7 *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, that is,*

- *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 6 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 6, M_i is a maximal independent set over degree-one and degree-two in 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 6, 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 7 shows that our update method correctly modifies the cluster tree, and thus marginals can be correctly computed. Note that, by Lemma 3, 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 8 *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_i, f_j) in the elimination tree. Then, $|A_i| \leq 12$.*

Proof First, we observe that the edge (f_i, 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 sub-tree throughout the elimination procedure. For the remained of the proof, we focus on the component associated with f_i, and show that it has at most six 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 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 two 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 13. 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 one 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 13 has $|B_i| = 1$, so we can increase $|B_i|$ by one.

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

To prove this, we argue that if there are more than six affected nodes in the connected component, our algorithm eliminates at least two factors. Since affected nodes form a sub-tree that interacts with the rest of the tree on at most two factors, what remains to be shown is that in any tree with at least four nodes, the size of a maximal independent set over the nodes with degree at most two is at least two. To see this, observe that every tree has two leaves, and if the size of the tree is at least four, the distance between these two leaves is at least two or the tree is star-shaped. In either case, any maximal independent set must include at least two 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 9 *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 8 and the number of rounds is $O(\log n)$ by Lemma 3, the result follows. ■

We can easily generalize these results to multiple edge insertions and deletions by considering each connected component resulting from a modification separately. As we discussed in Section 4.1, 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 10 *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. Maintaining MAP Configurations

The previous sections 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 section, 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.

5.1 Computing MAP Configurations Using a Cluster Tree

In Section 3, 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 (2) for computing a cluster λ_j to be

$$\lambda_j = \max_{\mathcal{V}_j} f_j \prod_{\lambda_k \in \mathcal{C}_T(f_j)} \lambda_k \tag{5}$$

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.

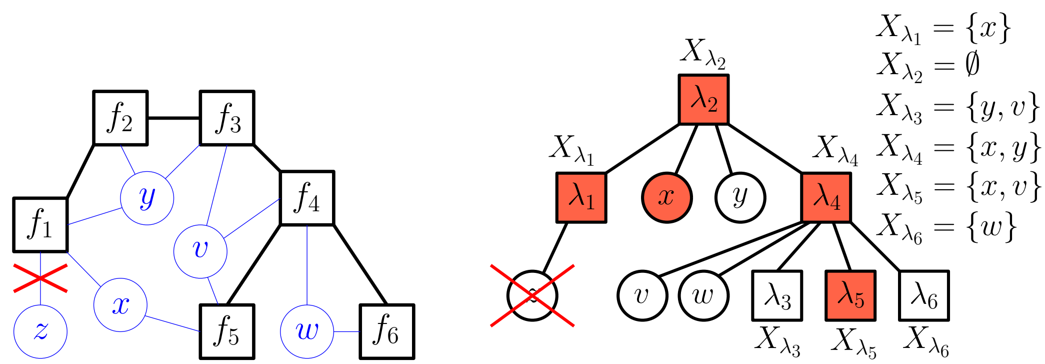


Figure 14: *Updating a MAP configuration.* Factor f_1 is modified and no longer depends on z on the factor graph (left). We first update the clusters on the path from modified clusters to the root, namely, λ_1 and λ_2. Then, we check for changes to the MAP configuration using a top-down traversal in 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

$$\mathcal{V}_j^* = \arg\max_X \, \delta(X_{\lambda_j} = X_{\lambda_j}^*) \, f_j \prod_{\lambda_k \in \mathcal{C}_T(f_j)} \lambda_k \tag{6}$$

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) is thus the optimal configuration for $\mathcal{V}_j$ (which by definition includes the Markov blanket) for each child cluster λ_k; see Figure 14 for an example.

Theorem 11 *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 (5) and (6) takes $O(d^{3w})$ by Lemma 3. Since the algorithm visits each node twice, once bottom-up using Equation (5) and once top-down using Equation (6) the total cost is $O(nd^{3w})$. ■

5.2 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 Section 4, we recompute each cluster along the path using Equation (5). 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 14 shows an example of how a MAP configuration changes after a factor (e.g., 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 12 *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 Section 4. 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: 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). 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 1, we can show that $\left|\mathcal{V}_j\right| \leq w$. 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 Section 4 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 $O(\ell \log n)$ 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. 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- or max-product algorithms. For the experiments, we implemented our proposed approach as well as the sum- and max-product algorithms in Python.[1] 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). To evaluate our algorithm, we performed experiments with both synthetic data (Section 6.1) and real-world applications (Sections 6.2 and 6.3).

First, we evaluate the practical effectiveness of our proposed approach by considering synthetically generated graphs to compute marginals (Section 6.1.3) and MAP configurations (Section 6.1.4). 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.

Second, we evaluate the effectiveness of our approach for two applications in computational biology. The first 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 sequence. The second application evaluates our approach on 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.

6.1 Experiments with Synthetic Data

For our experiments with synthetically generated data, 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-

1. The source code of our implementation can be obtained by contacting the authors.

product algorithm. Since we perform exact inference, the sum-product algorithm offers an adequate basis for comparison.

6.1.1 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:

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

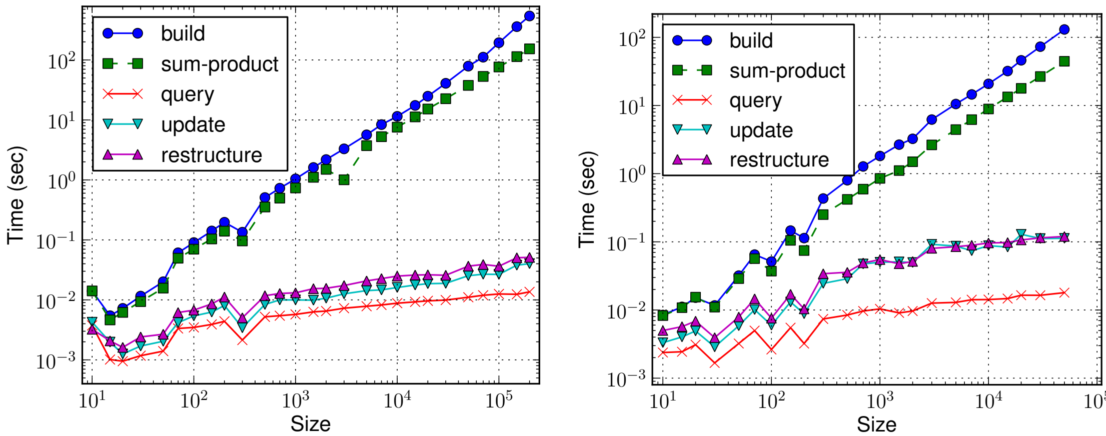
Here, p is a parameter that when set to 1 results in a linear chain. More generally, the parameter p determines how far back a node is connected while growing the random tree. The i^{th} node is expected to connect as far back as the j^{th} node where $j = i - 1/p$, due to the truncated geometric distribution. 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, that is, each entry is randomly chosen from e^Z where Z is a Gaussian distribution with zero mean and unit variance.

6.1.2 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.



(a) tree-structured factor graphs

(b) loopy factor graphs with tree-width 3

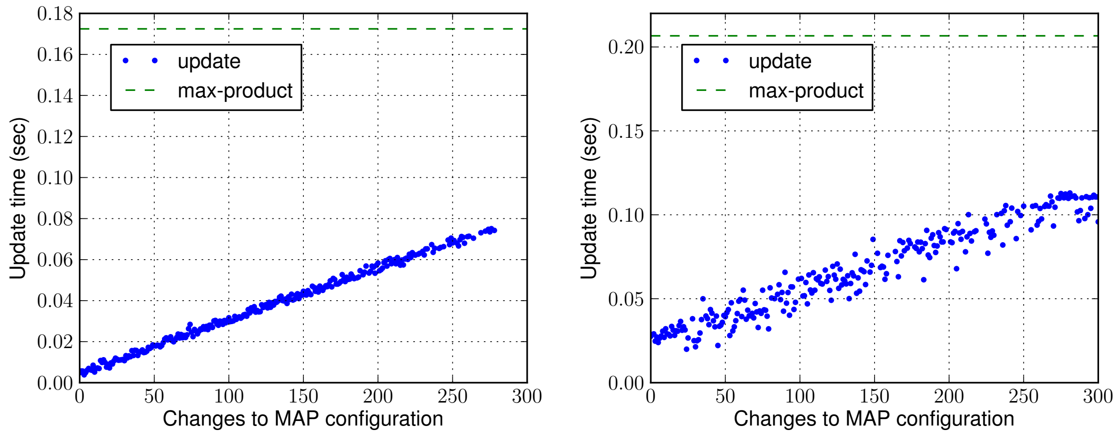
Figure 15: *Marginalization queries and model updates.* We measure the running times for naive sum-product, building the cluster tree, computing marginal queries, updating factors, and restructuring (adding and deleting edges to the elimination tree) for tree-structured and loopy factor graphs. Building 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.

6.1.3 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. We consider graphs with tree-width one (trees) and three, with 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 15 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



(a) tree-structured

(b) loopy

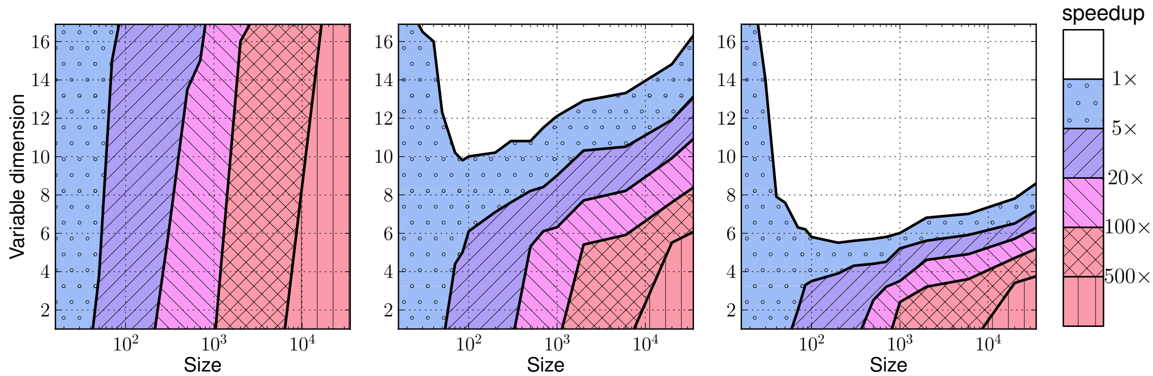
Figure 16: *Updates to MAP configurations.* We report the time required to update a MAP configuration after a single change is made to the input model, in both tree-structured and loopy factor graphs, with 300 variables. Our algorithm takes time that is roughly linear in the number of changed entries, unlike the standard max-product algorithm, which takes time that is linear in the size of the model.

to four orders of magnitude faster than a from-scratch inference with the sum-product algorithm. Update and restructuring operations are costlier than the query operation, as predicted by our complexity bounds on updates ($O(d^{3w} \log n)$, Theorem 5) and queries ($O(d^{2w} \log n)$, Theorem 4). 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.

6.1.4 MAP CONFIGURATIONS

We also tested the approach for computing and maintaining MAP configurations, as outlined in Section 5. 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 16 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 MAP on 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 were substantially faster than computing a MAP configuration from scratch in all cases, for both tree-structured and loopy graphs.



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

Figure 17: *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 and variable dimensions 2–16.

6.1.5 Efficiency Trade-offs and Constant Factors

Our experiments with the computations of marginals and MAP configurations (Sections 6.1.3 and 6.1.4) 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 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 15, 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 17 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 17a, 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 17 and 15 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, that is, $\beta \gg \alpha$, 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 15) suggests that a similar case holds. Specifically, in Theorem 2 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 15(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

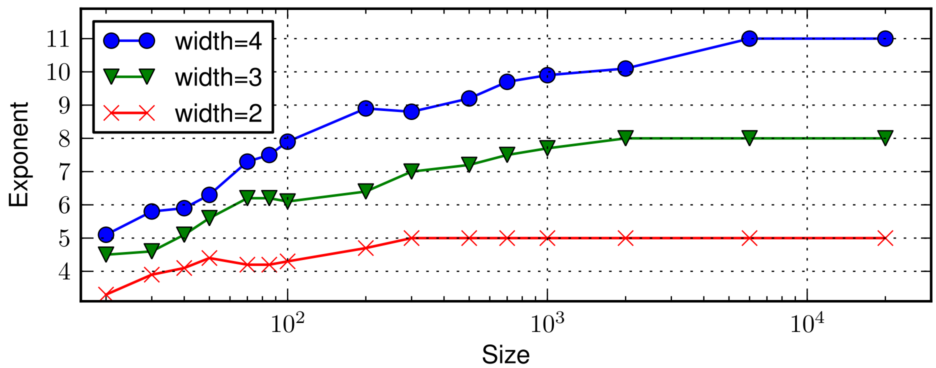


Figure 18: *Cost of cluster computation.* The maximum exponent e during the computation of clusters, which takes $O(d^e)$ time, is plotted as a function of the input size. As can be seen, the exponent starts relatively small and increases to reach the theoretical maximum of three times the tree-width 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, that is, it is not tight except in larger graphs.

our build time about 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 10 and 4. According to Theorems 10 and 4, 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, for example, 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 17 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 18 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.

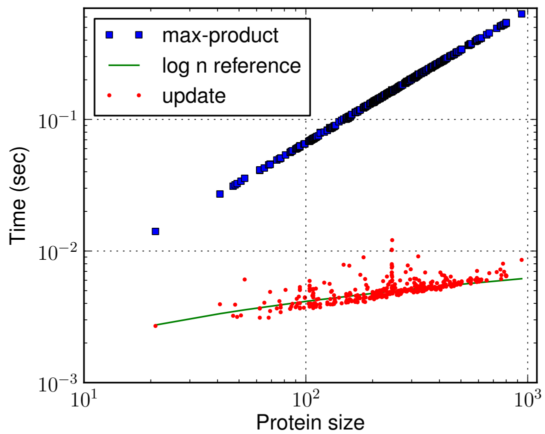
6.2 Sequence Analysis with Hidden Markov Models

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 configuration 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 (Frishman and Argos, 1995), 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 strands in the given sequence. This general approach has been studied and refined (Chu et al., 2004; Martin et al., 2005), and is capable of accurately predicting secondary structure. In the context of secondary structure 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 at any application where biological sequences are represented by HMMs (e.g., DNA or RNA sequence, exon-intron chains, CpG islands) and we want to study the effects of changes to these sequences.

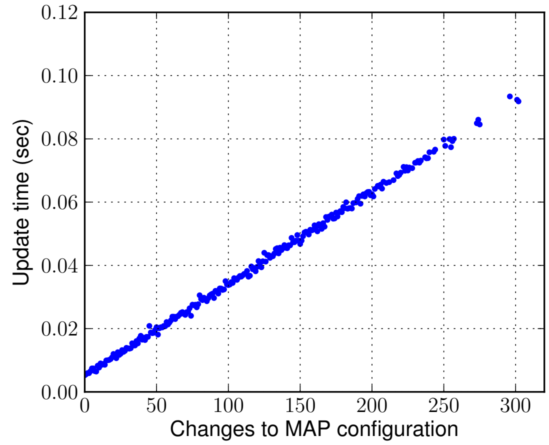
For our experiments, we constructed an HMM for secondary structure prediction by constructing an observed state for each amino acid in the primary sequence, and a corresponding hidden state indicating its secondary structure type. We estimated the model parameters using 400 protein sequences labeled by the DSSP algorithm (Kabsch and Sander, 1983), 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 19(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 19(b)). As in Section 6.1.4 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 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



(a) Update runtimes for 400 proteins



(b) Update times for protein 1QOY

Figure 19: *Secondary structure prediction using HMMs.* We applied our algorithm to perform updates 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 update the MAP configuration is linear in the number of changes to the MAP configuration, rather than in the size of the HMM.

essentially defines its biochemical function. Graphical models constructed from protein structures have been used to successfully predict structural properties (Yanover and Weiss, 2002) as well as free energy (Kamisetty et al., 2007). These models are typically constructed by taking each node as an amino acid whose states represent a discrete set of local conformations called *rotamers* (Dunbrack Jr., 2002), and basing conditional probabilities on a physical energy function (e.g., Weiner et al., 1984 and Canutescu et al., 2003).

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

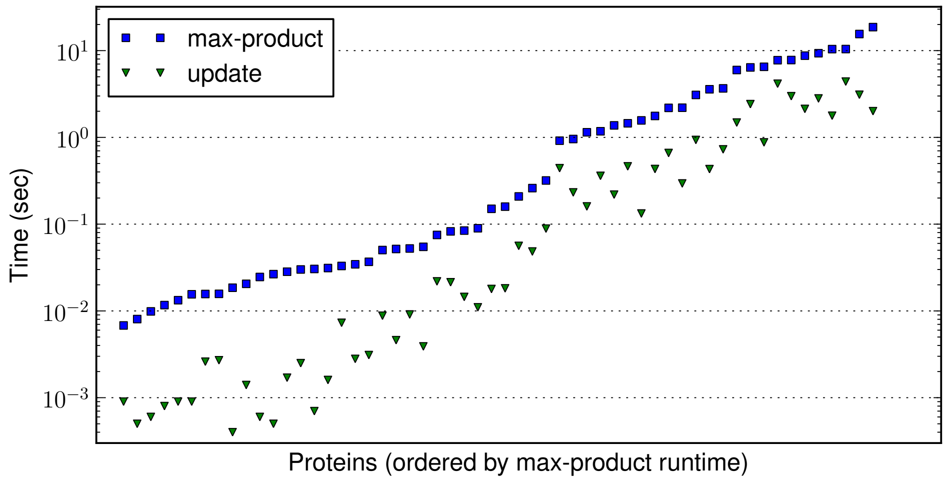


Figure 20: *Adaptive sidechain packing for protein structures.* For 60 proteins from the SCWRL benchmark, we compared the time to adaptively update a MAP configuration against max-product. Since this set of proteins have a diverse set of folds (and thus graph structures), we order the inputs by the time 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.

For each protein, we 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 20. 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 6.1.4 and 6.2. 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.

7. Conclusion

In this paper, 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 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 tree-width 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.

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 perform approximate inference. The approach does appear to be amenable to methods that rely on approximate elimination (e.g., Dechter, 1998), since these approximations can incorporated be into the cluster functions in the cluster tree. Approximate methods that are iterative in nature (e.g., Wainwright et al., 2005a,b and Yedidia et al., 2004), 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. While our benchmarks show that this overhead can be pessimistic, 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., its 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.

Acknowledgments

This research was supported in part by gifts from Intel and Microsoft Research (U. A.) and by the National Science Foundation through award IIS-1065618 (A. I.) and the CAREER award IIS-0643768 (R. M.).

References

U. 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.

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

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.

U. A. Acar, G. Blelloch, and J. Vittes. An experimental analysis of change propagation in dynamic trees. In *Proc. 7th ACM-SIAM W. on Algorithm Eng. and Exp'ts*, 2005.

U. A. Acar, Guy E. Blelloch, and Robert Harper. Adaptive functional programming. *ACM Trans. Prog. Lang. Sys.*, 28(6):990–1034, 2006.

U. A. Acar, A. T. Ihler, R. R. Mettu, and Ö Sümer. Adaptive Bayesian inference. In *Advances in Neural Information Processing Systems 20*. MIT Press, 2007.

U. A. Acar, Guy E. Blelloch, Matthias Blume, Robert Harper, and Kanat Tangwongsan. An experimental analysis of self-adjusting computation. *ACM Trans. Prog. Lang. Sys.*, 32(1):3:1–3:53, 2009a.

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, 2009b.

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.

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

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

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.

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.

A. L. Delcher, A. J. Grove, S. Kasif, and J. Pearl. Logarithmic-time updates and queries in probabilistic networks. *J. Artificial Intelligence Research*, 4:37–59, 1995.

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

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

M. A. Hammer, U. A. Acar, and Y. 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.

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

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.

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

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

P. Kohli and P. H. S. Torr. Dynamic graph cuts for efficient inference in markov random fields. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 29:2079–2088, 2007.

N. Komodakis, G. Tziritas, and N. Paragios. Performance vs computational efficiency for optimizing single and dynamic mrfs: Setting the state of the art with primal-dual strategies. *Comput. Vis. Image Underst.*, 112:14–29, October 2008.

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.

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.

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.

G. L. Miller and J. H. Reif. Parallel tree contraction and its application. In *Proc. 26th IEEE Symp. Found. of Comp. Sci.*, pages 487–489, 1985.

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. Prentice-Hall PTR, 2006.

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

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

D. D. Sleator and R. E. Tarjan. A data structure for dynamic trees. *Journal of Computer and System Sciences*, 26(3):362–391, 1983.

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, 2005a.

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 2005b.

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.

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.

C. Yanover and Y. Weiss. Approximate inference and protein folding. In *Proc. NIPS*, pages 84–86, 2002.

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.