2008

Adaptive Inference and Its Applications to Protein Modeling

Bo Jiang
University of Massachusetts Amherst

ADAPTIVE INFERENCE AND ITS APPLICATION TO PROTEIN MODELING

A Thesis Presented

by

BO JIANG

Submitted to the Graduate School of the
University of Massachusetts Amherst in partial fulfillment
of the requirements for the degree of

MASTER OF SCIENCE

September 2008

Electrical and Computer Engineering

ADAPTIVE INFERENCE AND ITS APPLICATION TO PROTEIN MODELING

A Thesis Presented

by

BO JIANG

Approved as to style and content by:

Ramgopal R. Mettu, Chair

Dennis L. Goeckel, Member

Weibo Gong, Member

C. V. Hollot, Department Head
Electrical and Computer Engineering

TABLE OF CONTENTS

LIST OF FIGURES

Figure **Page**

CHAPTER 1

INTRODUCTION

Inference problems naturally arise in many scientific and engineering areas. For some applications, the inference is done once and for all. We can call such inference static. For many other applications, however, we need to perform inference under dynamically changing conditions. For example, it might be desirable to assess the effects of a large number of possible interventions. This poses the problem of adaptive inference, namely, how to quickly incorporate changes and compute new inference results from old ones.

In this thesis, we present an adaptive inference algorithm with application to computational biology. The contributions of this thesis include an algorithm for dynamically updating optimal configuration and its theoretical analysis, the application of adaptive inference to protein modeling, a modified dead-end elimination method and a C++ library for performing adaptive inference.

In Chapter 2, we will discuss the use of graphical models to solve inference problems. We will first review the classical sum-product type algorithms [9]. Then the adaptive inference algorithm in [1] is introduced. In [1], the focus is on the computation of marginals. In this thesis, we will also address the problem of adaptively updating minimizers in the context of the min-sum algorithm, which is a variant of the sum-product algorithm.

Chapter 3 considers the application of the adaptive inference algorithm to computational biology. A central problem in computational biology is the prediction of the three-dimensional conformation of a protein, given its sequence of amino acids.

Usually the backbone is assumed to be known [5, 6, 16] and the problem reduces to side-chain packing. The need for adaptiveness is motived by the study of ligand binding and allostery, where we would like to know how the global conformation changes when some local changes are introduced. Chapter 3 will briefly review the side-chain packing problem and then introduce the adaptive side-chain packing problem.

Chapter 4 presents some implementation details and the experimental results. The adaptive inference algorithm is tested first on a synthetic benchmark and then on real proteins. The results for both tests are presented. Finally, Chapter 5 concludes the thesis.

CHAPTER 2

GRAPHICAL MODELS AND ADAPTIVE INFERENCE

In probability theory, a graphical model is a graph in which each node represents a random variable, and the absence of an edge between two nodes represents the conditional independence of the corresponding variables. Thus a graphical model captures the structural information of the joint probability distribution. More generally, a graphical model provides a graphical representation of the structure of any function f of a set of variables $x_1, x_2, \ldots, x_n$. For the purpose of this thesis, we will focus on a particular type of graphical model, namely the factor graph.

Given a graphical model, some useful information can be efficiently computed by exploiting the structural information. The information extraction process is referred to by the general term *inference*. Classical examples include the sum-product algorithm and its variants [9]. In some applications, for example, the structural study of proteins in Section 3.4, we need to extract information under dynamically changing conditions. This poses the problem of adaptive inference. In [1], Acar et al proposed an adaptive inference scheme, which provides a solution to this problem in the special case of factor trees.

In this chapter, we start with a discussion of factor graphs and the classical sum-product type algorithms in Section 2.1. Then we introduce the data structure for adaptive inference in Section 2.2. The two basic operations, query and update, are discussed in Sections 2.3 and 2.4, respectively. Finally Section 2.5 concludes this chapter with a comparison of the complexities of the classical and adaptive inference schemes.

2.1 Factor Graphs and Sum-product Algorithm

This section provides some background information about factor graphs and the sum-product algorithm. Section 2.1.1 introduces the concept of a factor graph. Section 2.1.2 discusses the classical sum-product algorithm for computing marginal distributions. Section 2.1.3 discusses the min-sum algorithm, a variant of the sum-product algorithm, with an emphasis on computing the consistent variable configurations.

2.1.1 Factor Graph

Throughout this thesis, a *variable* is assumed to take on finitely many values, and a *factor* is a function of finitely many such variables. We now give a formal definition of factor graphs [9].

Definition 1 (Factor graph)**.** A *factor graph* is a bipartite graph (X, F, E), where $X = \{x_1, \ldots, x_n\}$ is a set of variables, $F = \{f_1, \ldots, f_m\}$ a set of factors, and $E \subset X \times F$ a set of edges such that $e = (x_i, f_j) \in E$ if and only if x_i is an argument of factor f_j.

Definition 2 (Factor tree)**.** A *factor tree* is a factor graph with no cycles.

A factor graph serves as a graphical representation of the factorization of a function. Suppose, for example, a function f factors as follows,

$$f(x, y, z, u, v) = f_1(u) \cdot f_2(x, u, v) \cdot f_3(v) \cdot f_4(x, y) \cdot f_5(y, z) \cdot f_6(z)$$

Then the corresponding factor graph is shown in Fig. 2.1. Note that it is acyclic, so it is also a factor tree.

Although a factor graph is initially used to represent the factorization of a function into factors, we see in Fig. 2.1 that the graph itself does not specify multiplication as the operation which combines the "factors" $f_1, \ldots, f_6$. Therefore, it is perfectly

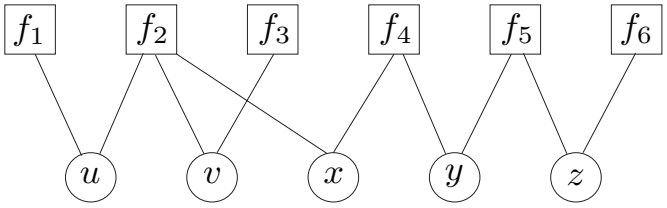


Figure 2.1. An example of factor graph.

suitable to represent the decomposition of f, for instance, into the sum of several additive terms,

$$f(x, y, z, u, v) = f_1(u) + f_2(x, u, v) + f_3(v) + f_4(x, y) + f_5(y, z) + f_6(z)$$

We will make use of this observation when we model the energy of proteins in Section 3.2.

2.1.2 Sum-product Algorithm

Given the factorization of a function $f(X) = f(x_1, \ldots, x_n)$ and its factor graph representation, various computations can be done efficiently. In particular, we consider the case that the factor graph is in fact a tree. In this case, the marginals, defined for each variable x by

$$f_x(x) = \sum_{X \setminus \{x\}} f(X) \tag{2.1}$$

can be computed using the *sum-product algorithm* [9] very efficiently. The sum-product algorithm can be described by the following message-passing protocol: For each $e = (u, v) \in E$, u sends to v a message $\mu_{u \to v}$ upon receiving messages from all its

neighbors other than v and similarly for $\mu_{v\to u}$ from v to u. Now suppose $u = x$ is a variable and $v = f$ a factor, then the messages $\mu_{u\to v}$ and $\mu_{v\to u}$ are defined as follows:

$$\mu_{x\to f}(x) = \prod_{g\in N_x\setminus\{f\}} \mu_{g\to x}(x) \tag{2.2}$$

$$\mu_{f\to x}(x) = \sum_{X\setminus\{x\}} \left(f(X) \cdot \prod_{y\in N_f\setminus\{x\}} \mu_{y\to f}(y) \right) \tag{2.3}$$

where N_v is the set of neighbors of v. After all the $2|E|$ messages have been passed, the marginal $f_x(x)$ is computed by multiplying all the messages received by node x,

$$f_x(x) = \prod_{f\in N_x} \mu_{f\to x}(x) \tag{2.4}$$

Although it is specified by the above protocol that a node send a message immediately it is able to do so, this is not necessary. The same correct marginals will be obtained, as long as all the $2|E|$ messages are sent in accordance with the causality constraints, i.e. a node sends a message to a neighbor only after it has received messages from all the other neighbors. In particular, we can specify a node as the root, thus making the factor tree into a rooted tree, and pass messages first from bottom up and then top down.

2.1.3 Min-sum Algorithm

The sum-product algorithm introduced in Section 2.1.2 is not confined to the operations of addition "+" and multiplication "×". In fact, it is valid for any semiring $(R, \oplus, \otimes)$, where $\oplus$ corresponds to "addition" and $\otimes$ to "multiplication" [9]. Of particular interest is the "min-sum" semirings, where minimization plays the role of "addition", and summation that of "multiplication". Therefore, if we want to

compute the "min-marginals" of a function f that is the sum of several additive terms, the corresponding message-passing protocol will take the following form

$$\mu_{x\to f}(x) = \sum_{g\in N_x\backslash\{f\}} \mu_{g\to x}(x) \tag{2.5}$$

$$\mu_{f\to x}(x) = \min_{X\backslash\{x\}} \left(f(X) + \sum_{y\in N_f\backslash\{x\}} \mu_{y\to f}(y) \right) \tag{2.6}$$

By associating each variable node with the additive identity function, which is identically zero, we can combine the above equations into a single equation

$$\mu_{u\to v}(X_u \cap X_v) = \min_{X_u\backslash X_v} \left(f_u(X_u) + \sum_{w\in N_u\backslash\{v\}} \mu_{w\to u}(X_w \cap X_u) \right) \tag{2.7}$$

where X_v is the set of variables involved at node v. For a variable node $v = x$, $X_v = \{x\}$. For a factor node $v = f$, X_v is just the argument of the factor. Note that every node is treated equally in this form, and the sum-product algorithm applies to not only trees but "junction trees" as well, as we will discuss in Section 3.3.

In many applications, for instance, the protein side-chain packing problem to be discussed in Section 3.1, rather than the min-marginals, we are interested in the actual minimizers

$$\hat{X} = (\hat{x}_1, \ldots, \hat{x}_n) = \arg\min f(x_1, \ldots, x_n) \tag{2.8}$$

which we call an *global minimum configuration* (GMC). Note that the GMC is by no means unique for a given f. When it is unique, it can be obtained by computing the individual minimizers of the min-marginals

$$\hat{x}_i = \arg\min f_{x_i}(x_i) \tag{2.9}$$

where $\hat{x}_i$ is the unique minimizer for f_{x_i}. When it is not unique, however, care must be taken to insure the consistency of individual minimizers. One way to do this is

to compute the minimizers in a two-phase procedure similar to that described at the end of Section 2.1.2. In the top-down phase, when a node receives the message from its parent, it compute the minimizers for the involved variables

$$\hat{X}_u = \arg\min_{X_u} \left(f_u(X_u) + \sum_{w \in N_u} \mu_{w \to u}(X_w \cap X_u) \right) \tag{2.10}$$

in such a way that $\hat{X}_u$ is consistent with that of its parent. If we are only interested in the GMC and do not care about the min-marginals, a parent node needs only to send to its children the minimizers of its variables. More precisely, given a node u and its parent v, we can decompose the variables at u into two groups, $X_u = Y_u \cup Z_u$, where $Y_u = X_u \cap X_v$ and $Z_u = X_u \setminus X_v$. Then in the top-down process, v only needs to pass to u the minimizers $\hat{Y}_u$, and u will compute $\hat{Z}_u$ according to

$$\hat{Z}_u = \arg\min_{Z_u} \left(f_u(\hat{Y}_u, Z_u) + \sum_{w \in N_u \setminus v} \mu_{w \to u}(\hat{Y}_u, X_w \cap Z_u) \right) \tag{2.11}$$

When $Z_u = \emptyset$, no computation is performed.

The above procedure is best illustrated by an example. Fig. 2.2 illustrates the bottom-up phase of min-sum algorithm for the example corresponding to Fig. 2.1, of which node f_1 is taken as the root. Fig. 2.3 then shows the top-down phase for computing the GMC.

2.2 A Data Structure for Adaptive Inference

As we have mentioned at the beginning of this chapter, some applications require the computation of marginals, the GMC, etc, under dynamically changing conditions. The classical sum-product type algorithms are not designed for this kind of task, in the sense that if we change the function associated with a single node, it will take $O(n)$ time to update the extracted information, where n is the size of the graph. This

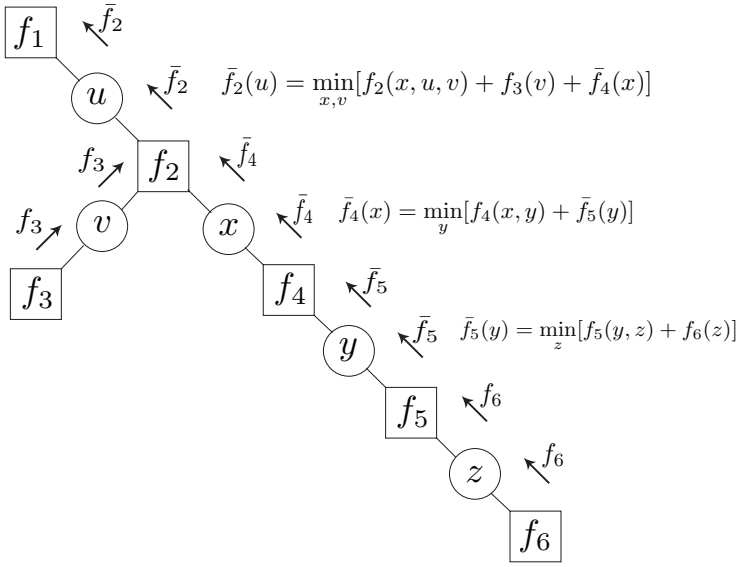


Figure 2.2. Bottom-up message-passing.

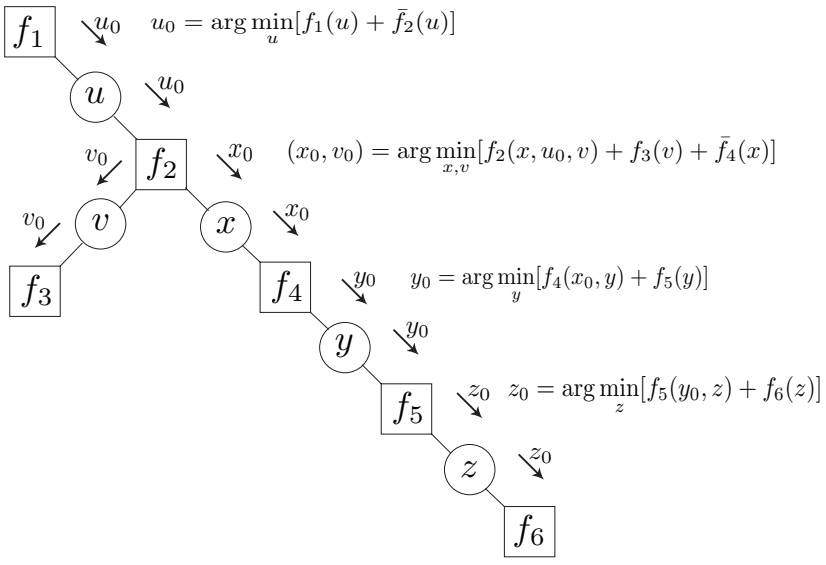


Figure 2.3. Top-down message-passing for computing GMC.

is because the tree can be highly unbalanced. For example, if the tree is a chain, and a leaf node is updated, then in the worst case we will have to propagate the change all the way to the other leaf node. In [1], Acar et al proposed an adaptive inference scheme for trees, which updates information in $O(\log n)$ time. The idea is to balance the input tree in such a way that messages can be passed hierarchically.

Section 2.2.1 discusses tree contraction, which transforms the input tree into a balanced tree. Section 2.2.2 discusses the local orientation of clusters, which is some structural information inherited from the input tree that is needed for marginal computation. Section 2.2.3 discusses the upward message passing in the cluster tree, which is similar to the message passing in the classical sum-product algorithm.

2.2.1 Tree Contraction

The pre-processing technique proposed in [1] is *tree contraction*. Tree contraction is used to construct a balanced representation of the input tree, which they call a *cluster tree*. There are three basic operations, *rake*, *compress* and *finalize*. A *finalize* operation removes a degree-0 node; a *rake* operation removes a degree-1 node; and a *compress* operation removes a degree-2 node while connecting its two neighbors. When a node is removed, the incident edges are also removed. Its neighbors immediately before the removal are also recorded for later use.

The cluster tree is constructed by applying rake and compress operations in rounds until there is only one node left, which is then removed by the finalize operation. In each round, the rake operation is applied to each degree-1 node and the compress operation to an independent set of degree-2 nodes randomly chosen by the method described below. In each round, we flip coins for each degree-2 node. Then a degree-2 node is chosen if and only if (1) it flips head and (2) each of its two neighbors either has degree more than two, or has degree two and flips tail. This way, each round

removes an expected constant fraction of nodes, making balanced the cluster tree to be constructed below.

When tree contraction terminates, we construct the cluster tree as follows. For each node v of the factor tree, a node $\bar{v}$, called a *cluster*, is added to the cluster tree. Here we make the convention that an unbarred letter v always refers to a node in the factor graph, and the same barred letter $\bar{v}$ to the corresponding cluster. If a node v is raked and u is its neighbor immediately before the rake operation, then an edge is added between clusters $\bar{v}$ and $\bar{u}$. If v is compressed, and u and w are its neighbors immediately before the compress operation, then an edge is added between $\bar{v}$ and $\bar{u}$ or $\bar{w}$, according as u or w is removed first during the contraction.

Here we have conceptually divided the clustering process into two phases for the sake of clearness, though in the implementation the cluster tree is built on the fly as the nodes are removed.

For the example in Fig. 2.1, the contraction process is illustrated in Fig. 2.4. In the first round, f_1, f_3 and f_6 are raked, and f_4 is compressed. In the second round, u, v and z are raked. This process continues until x is finalized in the fifth round. The resulting cluster tree is shown in Fig. 2.5. Note that the two neighbors of f_4 immediately before its removal are x and y, of which y is removed first, so $\bar{y}$ is the parent of $\bar{f}_4$ in the cluster tree.

2.2.2 Local Orientations of Clusters

To facilitate the computation of marginals, we need to additional structural information of the factor tree. Let node r be the one removed by the finalize operation, which corresponds to the root $\bar{r}$ of the cluster tree. Let u be a neighbor of a non-root node v at v's removal. If u is closer to r than v is, then $\bar{u}$ is said to be the *in* neighbor of $\bar{v}$, denoted by $\texttt{in}(v)$; otherwise, $\bar{u}$ is said to be the *out* neighbor of $\bar{v}$, denoted by $\texttt{out}(v)$. The orientations can be computed recursively by the algorithm

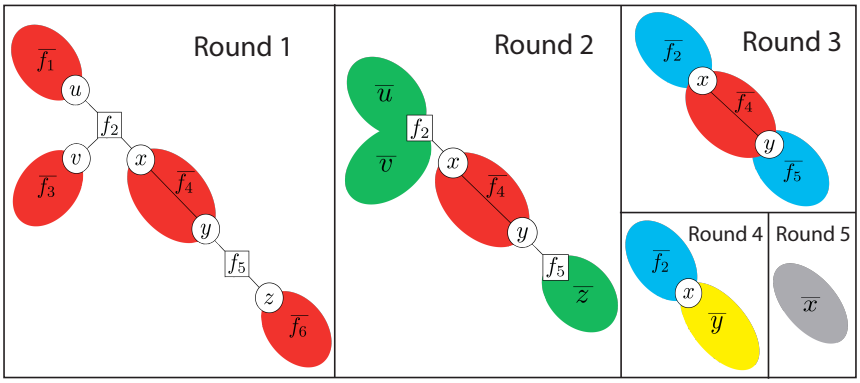


Figure 2.4. Tree contraction.

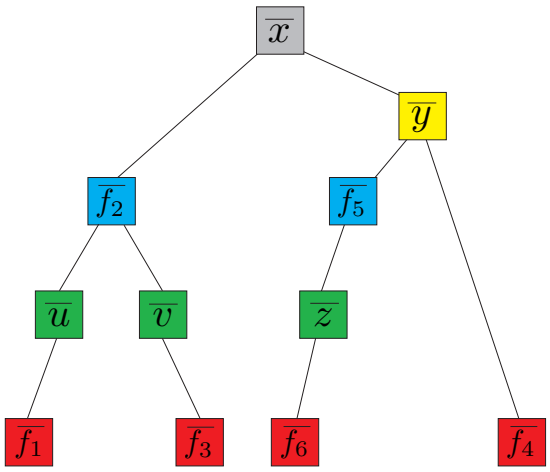


Figure 2.5. Cluster tree for the tree contraction in Fig. 2.4.

```
1: if v is raked with neighbor u at its removal then
2:    in(v̄) ⇐ ū, out(v̄) ⇐ NIL
3: else if v is compressed with neighbors u and w then
4:    compute orientation of u and w.
5:    if ū = in(w̄) or w̄ = out(ū) then
6:       in(v̄) ⇐ ū, out(v̄) ⇐ w̄
7:    else
8:       in(v̄) ⇐ w̄, out(v̄) ⇐ ū
9:    end if
10: end if
```

Figure 2.6. Algorithm for computing local orientations.

in Fig. 2.6. For the raked nodes, the orientations can be set during tree contraction. For compressed nodes, the orientations can be computed only after tree contraction is completed.

2.2.3 Bottom-up Message-passing in Cluster Tree

After tree contraction, further computations such as computing the marginals, can be accomplished by message passing in the cluster tree. This is, as before, divided into two phases, a bottom-up phase and a top-down phase. The bottom-up phase will be described here while the top-down phase will be deferred to Section 2.3. As a convention, we will use f_v to refer to the factor at node v in the factor graph, which, we recall, is set to the identity 1_v of the "multiplication" $\otimes$ if v is a variable node. We will also use X_v, as in Section 2.1.3, to represent the set of variables involved at node v.

To each cluster $\bar{v}$, we associate a function $g_{\bar{v}}$, called the *cluster function*, and a set of variables $A_{\bar{v}}$, called the *cluster variables*.

If v is the root of cluster tree, then $A_{\bar{v}} = \emptyset$; otherwise, $A_{\bar{v}}$ is computed by

$$A_{\bar{v}} = \left(X_v \cup \bigcup_{\bar{u} \in C_{\bar{v}}} A_{\bar{u}} \right) \cap \left(\bigcup_{u \in N_v} X_u \right) \tag{2.12}$$

where N_v is the set neighbors at v's removal, not its neighbors in the original graph as in Section 2.1.3. Note that $|N_v| = 1$ if v is raked and $|N_v| = 2$ if it is compressed. Here $C_{\bar{v}}$ is the set of children of $\bar{v}$ in the cluster tree.

For the sum-product semiring, the cluster function is given by

$$g_{\bar{v}}(A_{\bar{v}}) = \sum_{\sim A_{\bar{v}}} \left(f_v(X_v) \prod_{\bar{u} \in C_{\bar{v}}} g_{\bar{u}}(A_{\bar{u}}) \right) \tag{2.13}$$

where $\sim A_{\bar{v}}$ is the complement of $A_{\bar{v}}$. For the min-sum semiring, this becomes

$$g_{\bar{v}}(A_{\bar{v}}) = \min_{\sim A_{\bar{v}}} \left(f_v(X_v) + \sum_{\bar{u} \in C_{\bar{v}}} g_{\bar{u}}(A_{\bar{u}}) \right) \tag{2.14}$$

As is the case for the construction of cluster tree, the process of bottom-up message-passing can be merged with the tree contraction process.

2.3 Queries

After the bottom-up message-passing described in Section 2.2.3 , we obtain, for example, the minimum value of the summation in the case of the min-sum algorithm, which is of limited interest. If we are to get some other useful information, such as the marginals and the GMC, we need to perform some top-down message-passing, from the root to the node whose information we are querying.

2.3.1 Query for Marginals

Using the local orientations of clusters, the marginals can be computed by message-passing according to Eq. 2.15, which is a type of top-down message passing in the cluster tree. The message from cluster $\bar{u}$ to cluster $\bar{v}$ is computed as follows,

$$m_{\bar{u} \to \bar{v}} = \begin{cases} \sum_{\sim A_{\bar{v}}} \left(m_{\mathtt{in}(\bar{u}) \to \bar{u}} \cdot f_u(X_u) \cdot \prod_{\bar{w} \in C_{\bar{u}} \setminus \{\bar{v}, \mathtt{out}(\bar{v})\}} g_{\bar{w}}(A_{\bar{w}}) \right), & \text{if } \bar{u} = \mathtt{in}(\bar{v}) \\ \sum_{\sim A_{\bar{v}}} \left(m_{\mathtt{out}(\bar{u}) \to \bar{u}} \cdot f_u(X_u) \cdot \prod_{\bar{w} \in C_{\bar{u}} \setminus \{\bar{v}, \mathtt{in}(\bar{v})\}} g_{\bar{w}}(A_{\bar{w}}) \right), & \text{if } \bar{u} = \mathtt{out}(\bar{v}) \end{cases} \tag{2.15}$$

Whenever $\texttt{out}(\cdot)$ is undefined, i.e. `NIL`, the corresponding message is set to the identity.

When a node $\bar{v}$ has received messages from its neighbors, the marginal at that node is given by

$$f_{\bar{v}}(A_{\bar{v}}) = \sum_{\sim A_{\bar{v}}} m_{\texttt{in}(\bar{v})\to\bar{v}} \cdot m_{\texttt{out}(\bar{v})\to\bar{v}} \cdot f_v(X_v) \cdot \prod_{w\in C_{\bar{v}}} g_{\bar{w}}(A_{\bar{w}}) \tag{2.16}$$

For the min-sum semiring, the min-marginals can be computed similarly using

$$m_{\bar{u}\to\bar{v}} = \begin{cases} \min\limits_{\sim A_{\bar{v}}} \Big(m_{\texttt{in}(\bar{u})\to\bar{u}} + f_u(X_u) + \sum\limits_{\bar{w}\in C_{\bar{u}}\setminus\{\bar{v},\texttt{out}(\bar{v})\}} g_{\bar{w}}(A_{\bar{w}}) \Big), & \text{if } \bar{u} = \texttt{in}(\bar{v}) \\ \min\limits_{\sim A_{\bar{v}}} \Big(m_{\texttt{out}(\bar{u})\to\bar{u}} + f_u(X_u) + \sum\limits_{\bar{w}\in C_{\bar{u}}\setminus\{\bar{v},\texttt{in}(\bar{v})\}} g_{\bar{w}}(A_{\bar{w}}) \Big), & \text{if } \bar{u} = \texttt{out}(\bar{v}) \end{cases} \tag{2.17}$$

and

$$f_{\bar{v}}(A_{\bar{v}}) = \min_{\sim A_{\bar{v}}} \left(m_{\texttt{in}(\bar{v})\to\bar{v}} + m_{\texttt{out}(\bar{v})\to\bar{v}} + f_v(X_v) + \sum_{w\in C_{\bar{v}}} g_{\bar{w}}(A_{\bar{w}}) \right) \tag{2.18}$$

2.3.2 Query for Min-sum Configuration

In the min-sum algorithm, what are also interesting other than the min-marginals are the variable values (states) that actually achieve the minimum. The set of variables

$$X_{\bar{v}} = X_v \cup \bigcup_{\bar{u}\in N_{\bar{v}}} A_{\bar{u}} \tag{2.19}$$

involved at a cluster $\bar{v}$ is partitioned into two subsets $A_{\bar{v}}$ and $Z_{\bar{v}} = X_{\bar{v}} \setminus A_{\bar{v}}$. Once the configuration $\hat{A}_{\bar{v}}$ of $A_{\bar{v}}$ is known, we can compute the configuration of $Z_{\bar{v}}$, if it is nonempty, by

$$\hat{Z}_{\bar{v}} = \arg\min_{Z_{\bar{v}}} \left(f_v(X_v)\Big|_{\hat{A}_{\bar{v}}} + \sum_{u\in C_{\bar{v}}} g_{\bar{u}}(A_{\bar{u}})\Big|_{\hat{A}_{\bar{v}}} \right) \tag{2.20}$$

where $f_v(X_v)\Big|_{\hat{A}_{\bar{v}}}$ means the variables $X_v \cap A_{\bar{v}}$ are set to the value specified by $\hat{A}_{\bar{v}}$ and similarly for $g_{\bar{u}}(A_{\bar{u}})\Big|_{\hat{A}_{\bar{v}}}$.

Since $A_{\bar{v}} \subset X_{\bar{u}}$, where $\bar{u}$ is the parent cluster of $\bar{v}$, we can always query $\bar{u}$ for the configuration $\hat{A}_{\bar{v}}$. Since $A_{\bar{v}} = \emptyset$ if and only if $\bar{v}$ is the root cluster $\bar{r}$, the configurations can be computed following the path from root $\bar{r}$ to cluster $\bar{v}$. The message passed here from parent to child is the states of the cluster variables of the child cluster. If the downward passing continues until all leafs are reached, we will obtain a consistent GMC when it terminates.

2.4 Updates

The main purpose of using a tree contraction based algorithm is to make the algorithm adaptive to changes, i.e. it can efficiently update the current results to get the new results when some input information changes. There are two categories of input changes, of which one involves topological changes of the factor graph, e.g. addition and deletion of edges, and the other only changes the numerical values of input factors. In [1], both categories are considered in the context of computing marginals. In this section, we will focus on the latter, i.e. updates of factors without changing the graph topology.

2.4.1 Updating Cluster Functions

If we are to update a certain factor at a node v, we first replace the old factor $f_v(X_v)$ with the new one $\tilde{f}_v(X_v)$. Then we propagate the change upward to the root cluster according to Eq. 2.13, i.e. we recompute the cluster functions on the path from $\bar{v}$ to the root. Now the cluster tree is ready for query as described in Section 2.3.

2.4.2 Updating GMC

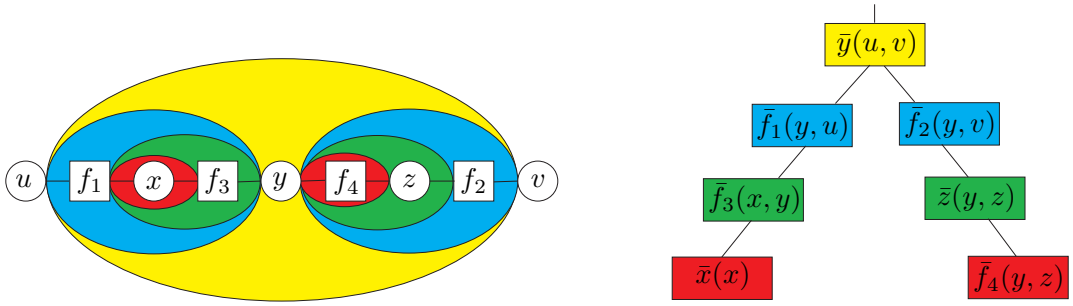
Sometimes it is useful to maintain the GMC as part of the state of the cluster tree. For example, in protein modeling, the GMC gives the protein conformation, which is what people are most interested in. We can always do a top-down tree traversal to recompute the GMC, but it can be done more efficiently as we will see presently.

```
Config(v̄)
1: recompute the configuration at v̄.
2: for each child ū of v̄ do
3:    if ū is dirty, or Â_ū have changed then
4:       call Config(ū)
5:    end if
6: end for
```

Figure 2.7. Algorithm for configuration update.

The key observation here is the Markov property. The optimal configuration of a subgraph, given the configuration of its neighboring set, is independent of the configurations of the rest of the graph. This property manifests itself in the cluster tree as follows. The configuration of variables involved in a subtree rooted at $\bar{v}$ are independent of the rest of the variables, given the configuration at the parent of $\bar{v}$. Therefore, if the configuration at a cluster $\bar{v}$ remains the same after the update as before the update, there is no need to recompute the configuration of a subtree rooted at a child cluster of $\bar{v}$, unless the subtree itself contains some dirty cluster whose factor function has been updated from the first place. Thus we need to mark each cluster on the path as *dirty*, when updating cluster functions. Then we update the GMC iteratively by the algorithm in Fig. 2.7, starting from the root cluster $\bar{r}$.

As an example, a portion of a factor tree and its corresponding portion of cluster tree are shown in Fig. 2.8. The boundary of this subtree consists of u and v. If the configurations $\hat{u}$ and $\hat{v}$ are given, then the optimal configurations of x, y and z are independent of the rest of the factor tree. In the cluster tree, this translates to that the configuration of the subtree rooted at $\bar{y}$ is guaranteed to be unchanged, if $\hat{A}_{\bar{y}} = \{\hat{u}, \hat{v}\}$ does not change in some updating process where none of the factor $f_1 \sim f_4$ changes. However, if, for instance, f_4 has been updated, we will always check all the cluster along the path $\bar{y} - \bar{f}_2 - \bar{z} - \bar{f}_4$, irrespective of whether $\hat{A}_{\bar{y}}$ has changed or not.



(a) Factor tree. (b) Cluster tree.

Figure 2.8. Markov property.

2.5 Comparison of Complexities

In this section, we will compare the theoretical complexities of the classical sum-product (or min-sum) and the adaptive (cluster-tree based) algorithms.

Let T be a factor tree with n nodes with maximum degree k, and the domain size of each variable be at most d. The traditional sum-product algorithm passes $n-1$ messages in order to find the marginal at a given node, and $2(n-1)$ messages to find all the marginals. Thus the complexity is always $O(n)$, i.e. linear in the size of the graph. If the constant involving d is written out explicitly, the complexity is then $O(d^k n)$. In [1], it is shown that the cluster tree can be built in $O(d^{k+2}n)$ time. Due to the additional factor d^2, the time it takes to build the cluster tree grows quickly as d increases. This may impose some limitation on the practical usage of this algorithm, as we will see later.

In the adaptive algorithm, the query process takes $O(kd^{k+2}\log n)$ time per node, as does updating cluster functions [1]. If we need to update factors repeatedly and query the marginals at a small fraction of nodes after each update, it is advantageous to use the adaptive algorithm instead of running sum-product each time. If all the marginals are needed, the adaptive algorithm can also produce the desired result in linear time by a top-down tree traversal, though it will have an additional factor of d^2 in the running time.

Interestingly, the cluster-tree framework allows us to update the GMC in time roughly proportional to the number of changed minimizers after an update, even though we do not know a priori which minimizers are going to change. The precise statement is given by the following theorem.

Theorem 1 (Updating GMC)**.** *Let a factor tree be given with n nodes, maximum degree k, domain size d, and its cluster tree. If l factors are modified, resulting in m changes in variable configurations, then the algorithm in Fig. 2.7 updates the GMC in $O(d^k\beta(1+\log\frac{n}{\beta}))$ time, where $\beta = \min\{n, l+km\}$.*

Proof. Consider a path from the root $\bar{r}$ to a cluster $\bar{v}$ where the configuration is recomputed for $\bar{v}$ but not for any of its children. Denote by P the collection of such paths. Observe that each path p in P satisfies at least one of the following properties:

1. The path p contains at least one of the l factors that are modified initially.

2. The path p involves at least one of the m variables whose states are changed.

The number of paths having Property 1 is at most l. Since each node of the factor tree has degree at most k, there are at most km paths having Property 2. Therefore, the size of P is upper bounded by $\beta = \min\{n, l+km\}$.

Now we bound the total number N of clusters involved in P. To this end, we order the paths in P as follows. Select an arbitrary path p_1 from P. After $p_1, \ldots, p_{i-1}$ have been selected, we select p_i to be a path that shares the fewest number of clusters with $p_1, \ldots, p_{i-1}$. Denote the number of clusters involved in $p_1, \ldots, p_i$ by N_i and the depth of the tree by D. Then the N_i's satisfy the following inequalities.

$$\begin{aligned} N_1 &\leq D \\ N_i - N_{i-1} &\leq D - s, \text{ for } k^{s-1} < i \leq k^s, s = 1, 2, \ldots \end{aligned}$$

Therefore,

$$\begin{aligned}
N = N_{|P|} &= N_1 + \sum_{i=2}^{|P|} (N_i - N_{i-1}) \\
&\leq D + \sum_{i=2}^{\beta} (D - \lceil \log_k i \rceil) \\
&\leq D\beta - \sum_{i=2}^{k^{\lceil \log_k \beta \rceil}} \lceil \log_k i \rceil \\
&= D\beta - \sum_{j=0}^{\lceil \log_k \beta \rceil - 1} \sum_{a=1}^{k^j (k-1)} \lfloor \log_k (k^j + a) \rfloor \\
&= D\beta - \lceil \log_k \beta \rceil k^{\lceil \log_k \beta \rceil} + \frac{k^{\lceil \log_k \beta \rceil} - 1}{k-1} \\
&\leq D\beta - \beta \log_k \beta + \beta
\end{aligned}$$

Since $D = O(\log n)$, we see that $N = O(\beta(1 + \log \frac{n}{\beta})$. Since the computation at each cluster is $O(d^k)$, the theorem follows. □

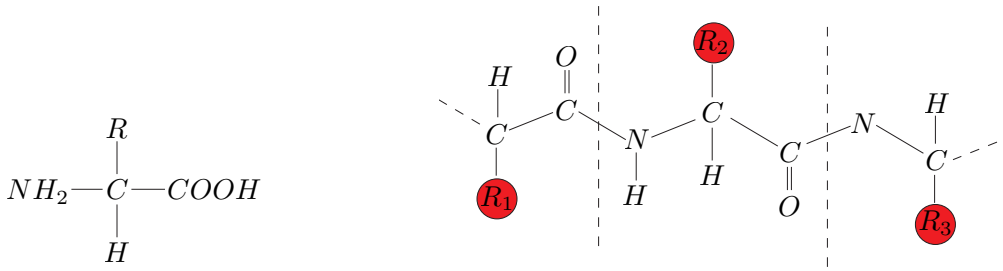
CHAPTER 3

APPLICATION TO PROTEIN MODELING

Previous work [5, 16, 17] has shown that proteins can be reasonably modeled by a graphical model, from which some useful information can be extracted by inference. In this chapter, we discuss the application of adaptive inference to protein modeling. Section 3.1 gives some background about proteins and introduces the side-chain packing problem. Section 3.2 then gives a particular way of modeling proteins through energy minimization. Section 3.3 shows how to convert a model on a general graph to a model on a tree, making the inference algorithm developed in Chapter 2 applicable. Section 3.4 discuss the possibility of exploiting the adaptiveness of our scheme to study protein structures.

3.1 The Side-chain Packing Problem

In chemistry, an amino acid is a molecule containing both an amino group $-\mathtt{NH_2}$ and a carboxyl group $-\mathtt{COOH}$. There are twenty different types of naturally occurring amino acids; all of these amino acids have the general structure in Fig. 3.1(a), with the amino and the carboxyl groups bonded to the same carbon, denoted C_α. Different amino acids are differentiated by the sets of "side-chain" atoms bonded to C_α, denoted by R in Fig. 3.1(a). A protein is one or more chains of amino acids joined together by *peptide bonds*, which are the $\mathtt{C}-\mathtt{N}$ bonds intersected by the vertical dashed lines in Fig. 3.1(b). The linear peptide chain then folds into a 3-dimensional conformation, which plays a crucial role in determining the function of the protein.



(a) Amino acid. (b) Protein segment.

Figure 3.1. Amino acid and protein segment.

Experimental methods such as X-ray or NMR are typically used to determine protein structures. These methods can find some or all of the necessary structural information, but are usually very time-consuming and must be supplemented by computational methods. The computation is usually divided into two phases, backbone prediction and side-chain prediction. In this thesis, we assume the backbone is given and only consider the side-chain prediction, also known as the side-chain packing (SCP) problem.

In most of the methods for solving the side-chain packing problem, the state space of each residue is discretized into a finite number of conformation candidates called rotamers[5, 6, 16, 17]. Then the problem becomes selecting one rotamer for each residue to minimize the energy of the protein. In other words, we would like to find $S^* = (s_1^*, \ldots, s_N^*)$ such that

$$E(S^*) = \min E(s_1, \ldots, s_N) \tag{3.1}$$

Here E is the total energy of the protein and s_i the state of the ith side-chain. In practice, some simple forms of the energy function E are used, which are decomposed into sums of functions of fewer variables. One commonly used form is the sum of pairwise interactions (Section 3.2). Unfortunately, however, even in this simplified model, the side-chain packing problem is still NP hard [2, 13].

3.2 Energy Minimization

In this section, we will introduce a specific model for the energy $E(S)$. The basic assumption is that $E(S)$ can be reasonably approximated by the sum of singleton scores for each residue and pairwise scores for each residue pair.

$$E(S) = \sum_i S_i(s_i) + \sum_{i<j} P_{ij}(s_i, s_j) \tag{3.2}$$

Let $E(a, b)$ be the energy between two atoms a and b. Let D_u, SC_u and BB_u be the sets of rotamers, side-chain atoms and backbone atoms for residue u, respectively. Then the singleton scores and pairwise scores are computed as follows [6, 16]

$$S_i(s_i) = -K \log \left(\frac{\Pr_i(s_i \mid \phi, \psi)}{\max_{s \in D_i} \Pr_i(s \mid \phi, \psi)} \right) + \sum_{|i-j|>1} \sum_{s \in SC_i} \sum_{b \in BB_j} E(s, b) \tag{3.3}$$

$$P_{ij}(s_i, s_j) = \sum_{a \in SC_i} \sum_{b \in SC_j} E(a, b) \tag{3.4}$$

where $\Pr_i(s \mid \phi, \psi)$ is the probability of rotamers s of residue i, given the backbone dihedral angles ϕ and ψ, and K is a weighting factor which we set to 6.

Now we can construct the so-called *geometric neighborhood graph* for the protein as follows [16]. Each residue of the protein is represented by a node and an edge is added between two nodes i and j if and only if

$$\max_{s_i \in D_i, s_j \in D_j} P_{ij}(s_i, s_j) > 0 \tag{3.5}$$

which means residues i and j interact with each other. The specific definitions of the pairwise atomic energy will be discussed in Section 4.3.2.

3.3 Tree Decomposition

The geometric neighborhood graph can now be converted into a factor graph, which, in general, contains cycles. The min-sum algorithm can then be applied to

find the approximate minimum energy configuration [17]. In order to find the exact minimum energy configuration, we transform the geometrical neighborhood graph into a tree through *tree decomposition* [15, 16].

Definition 3 (Tree decomposition)**.** Given a graph $G = (V, E)$, a *tree decomposition* of G is a pair (X, T) such that

(1) X is a cover of the set V, i.e. a collection of subsets of V whose union is V.

(2) For each edge $e = (u, v) \in E$, there is a $C \in X$ such that $u, v \in C$.

(3) T is a tree such that X is its vertex set and such that for each $v \in V$, elements of X containing v span a subtree of T.

The *tree width* of a tree decomposition (X, T) is $\max_{C \in X}(|C| - 1)$.

It has been shown that it is NP-complete to find the minimum width decomposition of a given graph [3]. We will use the minimum degree heuristic to compute the tree decomposition.

Given a tree decomposition (X, T) corresponding to the geometrical neighborhood graph of a protein, we assign a *potential* to each hypernode $C \in X$ as follows:

1. Initialize $p(C) = 0$, for $\forall C \in X$.

2. For each residue r, select a covering hypernode $C \in X$ such that $r \in C$, and let $p(C) \Leftarrow p(C) + S_r$, where S_r is the singleton score function for residue r.

3. For each residue pair i, j, select a hypernode $C \in X$ such that $i, j \in C$, and let $p(C) \Leftarrow p(C) + P_{ij}$, where P_{ij} is the pairwise score function for i and j.

By construction, the total energy E of the protein is equal to the sum of the potentials. Equipped with potentials, the junction tree is a generalization of the factor graph. We can apply the algorithms in Chapter 2 to the junction tree T, where the potentials play the role of factor functions. The minimizers then give a energy-minimizing conformation of the protein.

3.4 Adaptive Side-chain Packing

The inference framework can be potentially useful in studying allostery, protein-ligand binding or even inter-protein interactions. The basic idea here is that any minor change to a protein can be rapidly incorporated into the model.

In the study of ligand binding, we are interested in how the binding ligands change the three-dimensional conformations of proteins. The ligand directly affects the rotameric states of the binding site, from which the impact propagates to the whole protein. In protein design, we need to change the amino-acid type and hence the rotameric states of particular residues and examine whether the protein changes into some desired conformation.

In both examples, we are interested in the minimum energy conformations as the conformation of some given site changes dynamically. We call such problems the adaptive side-chain packing. This fits into our framework of updating the GMC in a min-sum cluster tree and it is natural to use our algorithm to study this problem.

CHAPTER 4

IMPLEMENTATION AND EXPERIMENTAL RESULTS

In this chapter, we will discuss the implementation of our algorithm and present some experimental results. Section 4.1 discusses how to organize the local computations in the adaptive inference in a more efficient way that gives a practical speed-up when the variable dimensions are large. Section 4.2 gives the test result of the algorithm on a randomly generated synthetic benchmark. Section 4.3 gives the detailed procedure for generating the graphical model of a protein, following [16] and the test result is given in Section 4.4.

4.1 Ordering in Message Computation

It is shown in [1] that the cluster tree can be built in $O(d^{k+2}n)$ time, and a query for marginal takes $O(kd^{k+2}\log n)$ time; see also Section 2.5. In this section, we will see that a careful organization of the computation can reduce the factor d^{k+2} to $d^{\max\{k,3\}}$, which can give a practical speed-up when the variable dimensions are large.

We now give a closer analysis of the message passing. Eq. 2.13 is repeated here.

$$g_{\bar{v}}(A_{\bar{v}}) = \sum_{\sim A_{\bar{v}}} \left(f_v(X_v) \prod_{\bar{u}\in C_{\bar{v}}} g_{\bar{u}}(A_{\bar{u}}) \right) \tag{4.1}$$

Note that the size of the function table for $g_{\bar{u}}$ is $O(d)$ or $O(d^2)$ according as u is raked or compressed. Since the factor tree has maximum degree k, the size of the function table for $f_v(X_v)$ is $O(d^k)$. If we carry out all the multiplications before the

summation, the worst case occurs when $\bar{v}$ has k children, two of which corresponds to compressed nodes. In this case, the running time for computing $g_{\bar{v}}$ is

$$(k-2)d^k + d^{k+1} + 2d^{k+2} = O(d^{k+2})$$

However, if we interleave the multiplication and summation, the practical running time can be improved. More specifically, let the elements of $C_{\bar{v}}$ be $\bar{u}_1, \bar{u}_2, \ldots, \bar{u}_k$, where u_{k-1} and u_k are compressed. Let $g_{\bar{v}}^{(0)} = f_v(X_v)$, and for $i = 1, 2, \ldots, k$, let

$$g_{\bar{v}}^{(i)}(A_{\bar{v}}^{(i)}) = \sum_{A_{\bar{v}}^{(i-1)} \cap A_{\bar{u}_i}} g_{\bar{v}}^{(i-1)}(A_{\bar{v}}^{(i-1)}) g_{\bar{u}_i}(A_{\bar{u}_i}) \tag{4.2}$$

Then we have $g_{\bar{v}}(A_{\bar{v}}) = g_{\bar{v}}^{(k)}(A_{\bar{v}}^{(k)})$. Assuming $d \geq 2$, the time for the computation at a single node is

$$\sum_{i=1}^{k-2} d^{k+1-i} + 2d^3 + 2d^3 = 2\frac{d^{k+1} - d^3}{d-1} + 4d^3 \leq 4d^k + 4d^3 = O(d^{\max\{k,3\}})$$

Thus we have reduced the per-node complexity from d^{k+2} to $d^{\max\{k,3\}}$ and the number of nodes can be bounded as before.

4.2 Test Results for Synthetic Benchmark

We have implemented the adaptive inference algorithm in C++ and tested it on a randomly generated synthetic benchmark as in [1]. The random factor tree has maximum degree 5. The dimension of each variable is randomly chosen to be 5, 5^2 or 5^3, so that each factor has size between 5 and 5^6. The test is done for both the sum-product and min-sum semirings. In each round of the updating process, only one randomly chosen factor is updated. Fig. 4.1 shows the result for the sum-product semiring, where the queries are for the for marginals. Fig. 4.2 shows the result for

the min-sum semiring, where the queries are for the variable states. In both cases, the time required to build the cluster tree is comparable to one run of sum-product or min-sum. The query and update operations are about two or three orders or magnitude faster. Note that the query in the case of min-sum semiring is much faster than that of sum-product. This is because in Eq. 2.20 we are operating on sections of the factors or cluster functions, which greatly reduces the function sizes when the variable dimensions are large.

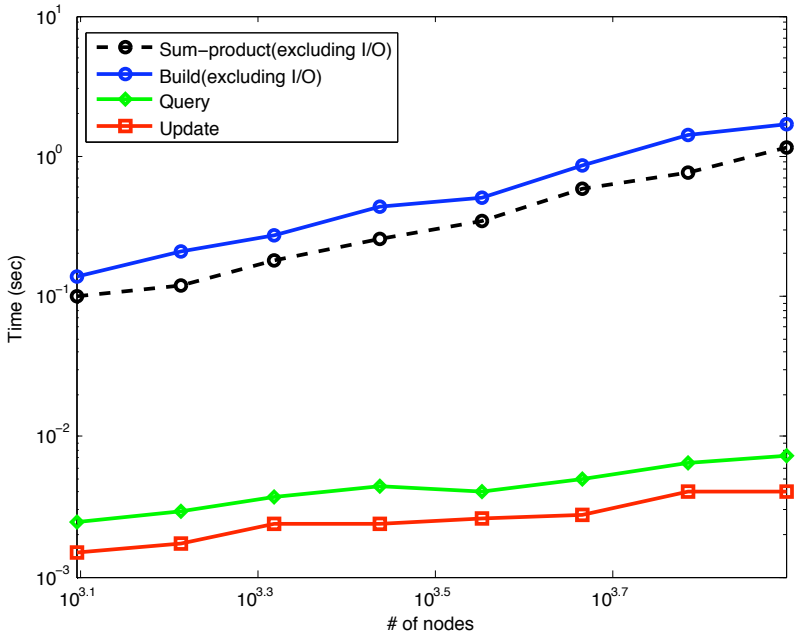


Figure 4.1. Log-log plot of running time for sum-product, building cluster tree, computing queries and updating factors.

The test for doing batch updates are shown in Figs. 4.3 and 4.4. In Fig. 4.3, ten randomly selected factors are updated each round for factor trees of different sizes. We also give an $O(\log n)$ reference line. Though it is not logarithmic, the increase in running time is slow as the size of factor tree increases, similar to the single update case. Fig. 4.4 shows the running time as the number of updated factor per round increases. The running time increases as more factors are updated each round and the rate of increase roughly follows a power-law in the test range. Fig. 4.5 shows

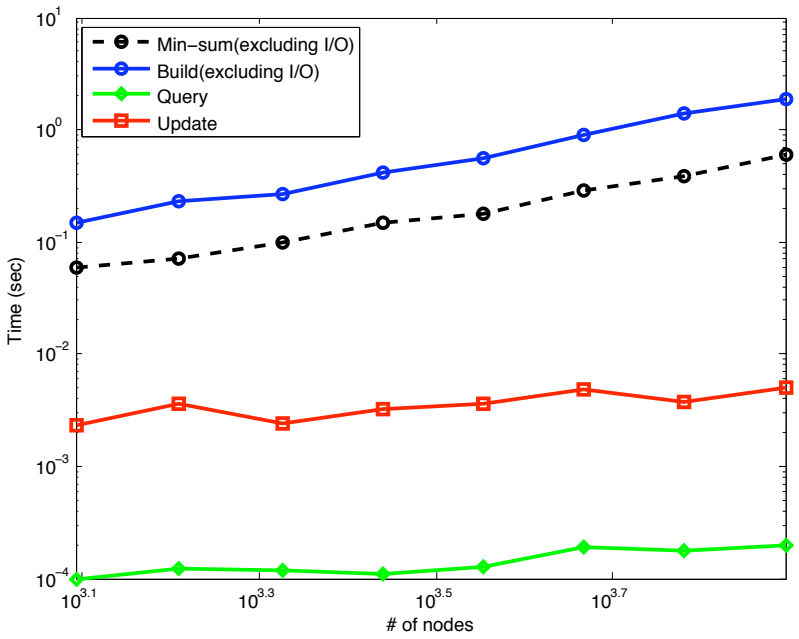


Figure 4.2. Log-log plot of running time for min-sum, building cluster tree, computing queries and updating factors.

that the updating time is approximately linear in the number of variables with state changes, in agreement with our analysis in Section 2.5.

4.3 Generating the Graphical Model of a Protein

In this section, we will describe the procedure to generate the graphical model of a protein, following [16]. Section 4.3.1 discusses how sets of rotamers are chosen for a given amino acid. Section 4.3.2 details the energy computation. Section 4.3.3 discusses dead-end elimination (DEE) and an extension of it.

4.3.1 Initialization of Rotamers

We use the backbone-dependent rotamer library in [5]. For a given residue, we first determine its backbone dihedral angles ϕ and ψ and the rotamers corresponding to these angles are loaded. When we cannot obtain both ϕ and ψ, e.g. at the end of the peptide chain, we use the backbone-independent library [5]. Rotamers are ranked

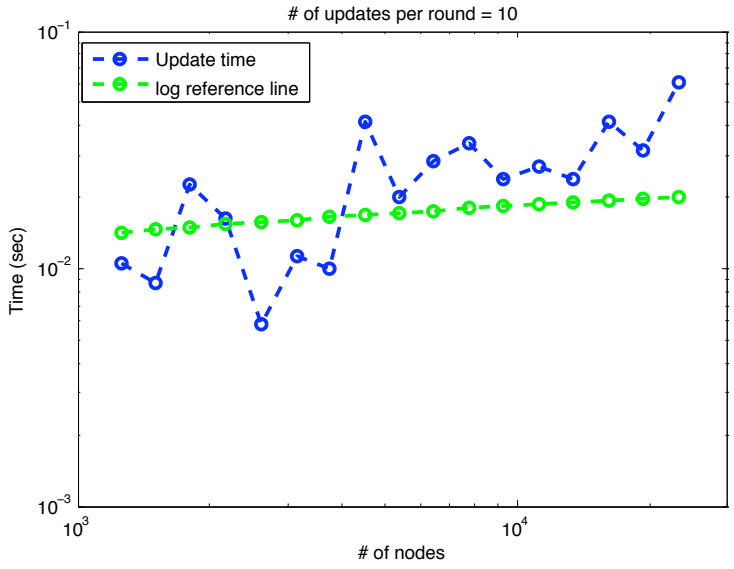


Figure 4.3. Log-log plot of updating time as the factor tree size increases.

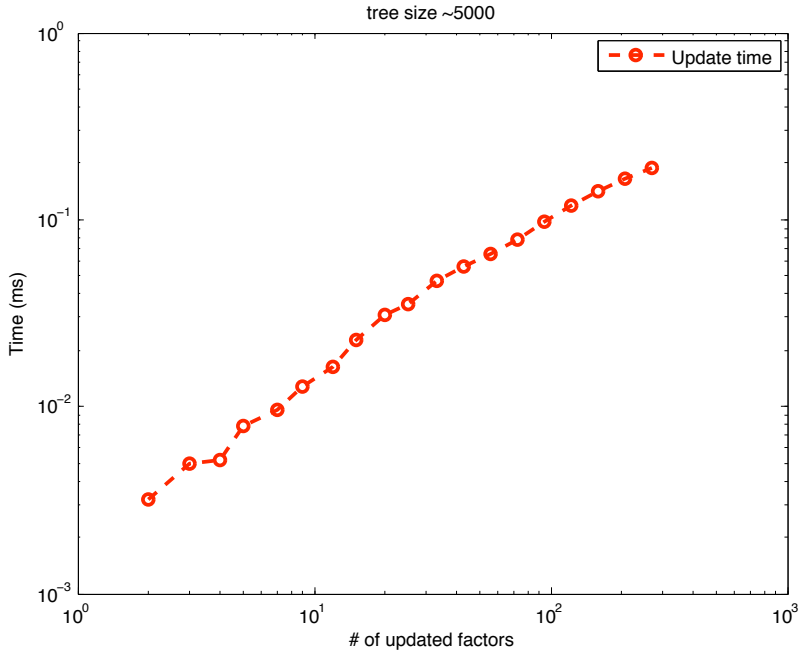


Figure 4.4. Log-log plot of updating time as the number of updated factors per round increases.

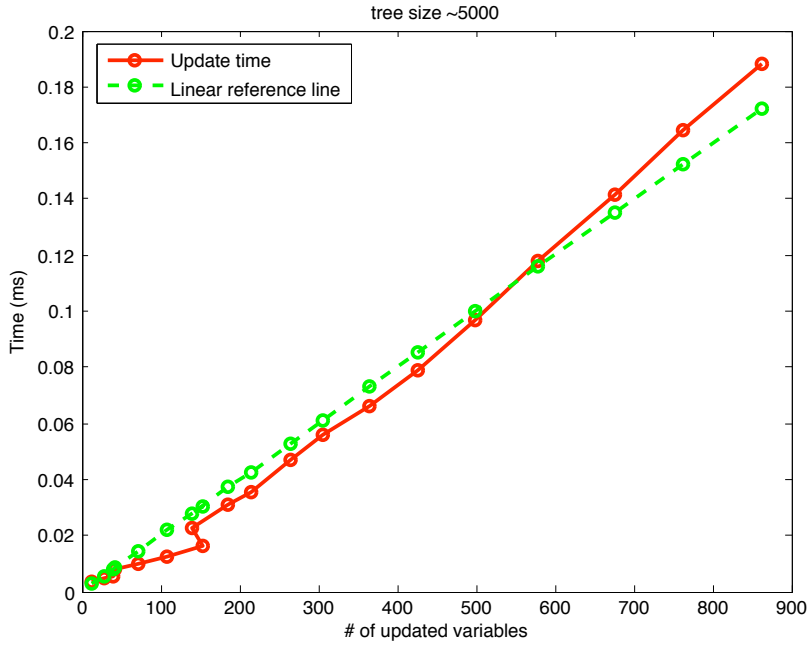


Figure 4.5. Linear plot of updating time as the number of updated variables per round increases.

from the highest probability to the lowest, and those corresponding to the tail are removed. The cutoff probability can be varied. In [6, 16], it is set to 90%.

4.3.2 Energy Computation

After the initialization step, the singleton and pairwise energies are computed according to Eqs. 3.3 and 3.4. The particular form of energy function we will use is the following piecewise linear function [6, 16]

$$E(a,b) = \begin{cases} 0, & r \geq R_{ab} \\ 10, & r \leq 0.8254 R_{ab} \\ 57.273(1 - \frac{r}{R_{ab}}), & \text{otherwise} \end{cases} \tag{4.3}$$

where r is the distance between the two atoms and R_{ab} the sum of their radii. While we use this relatively simple energy function, more sophisticated energy functions such

as AMBER [14] and CHARMM[4] can also be used in combination with a distance threshold.

If we are to compute the energy $E(a, b)$ for all possible combinations of atom pair a, b and then add them up, this will take quadratic time. Note that the energy function in Eq. 4.3 and many others are non-zero only when the distance between the two atoms is below a certain threshold. This property can be exploited to compute the energies very efficiently. As in [16], we use the ANN library in [10] to determine if two atoms are within a certain distance from each other. ANN stands for *Approximately Nearest Neighbor* library. ANN uses sophisticated data structures such as the *k-d* tree that are very efficient in searching with multidimensional key.

Let the totality of side-chain atoms be SC. Here atoms of the same residue corresponding to different rotamers are considered as different atoms. For each $a \in SC$, ANN gives all the neighbors N_a in SC of a within the distance threshold. Then, for each $b \in N_a$, if a, b are from different rotamers of different residues, we add $E(a, b)$ to the corresponding pairwise score $P_{ij}(s_i, s_j)$. The singleton scores can be computed similarly. As in [16], we use the BALL library [8] for basic concepts such as proteins and rotamers.

4.3.3 Construction of the Graph and Dead-end Elimination

Now that the energies have been computed, we can construct the graphical model. However, the graph thus constructed would be too dense to do inference efficiently. Therefore, dead-end elimination (DEE) is applied before the construction of the model in order to reduce the density. Let D_i be the state space of the i^{th} residue. The The Goldstein criterion [7] states: if there exists an $s'_i \in D_i$ such that

$$E(s_i) - E(s'_i) + \sum_{j:j\neq i} \min_{s_j \in D_j} [E(s_i, s_j) - E(s'_i, s_j)] > 0 \tag{4.4}$$

then remove s_i from D_i. Eq. 4.4 means that we can always replace s_i by s_i' to reduce the energy, hence losing no optimality by removing s_i from the search space. The Goldstein criterion can be applied iteratively until no rotamer is removed in an iteration. This will greatly reduce the search space, hence making it possible to search for the minimum energy conformation when the protein size is large.

However, the DEE procedure needs to be slightly modified for the adaptive SCP problems. In the adaptive case, we need to iterate through many possible conformations of a given site. For example, in protein design, we need to change the amino acid type and hence the rotameric state of a particular site. Using the adaptive inference framework, we build the cluster tree only once and then perform updates repeatedly. Here our goal is not to compute a single minimum energy conformation, but rather many min-energy conformations, each conditioned on the local conformation of the given site. Therefore, we must ensure that all such conditional minimum energy conformations are feasible after DEE, where by saying a conformation is feasible, we mean that every rotamer in this conformation is a valid choice for the corresponding residue. In order to generate this property, we show that it suffices to keep the rotamers of the given site from DEE pruning.

Theorem 2 (Modified DEE)**.** *If the rotamers of a given subset of residues be preserved, i.e. not pruned by DEE, then all the minimum energy conformations conditioned on that subset remain feasible after DEE.*

Proof. Note that the Goldstein criterion Eq. 4.4 is equivalent to the following: remove a rotamer s_i only if there exists $s_i' \in D_i$ such that

$$E(s_i) - E(s_i') + \sum_{j:j\neq i} [E(s_i, s_j) - E(s_i', s_j)] > 0 \tag{4.5}$$

for all conformations $(s_1, \ldots, \hat{s}_i, \ldots, s_n) \in (D_1 \times \cdots \times \hat{D}_i \times \cdots \times D_n)$. Here $\hat{}$ means the corresponding component should be removed.

Therefore, a rotamer s_i is preserved if for all $s_i' \in D_i$, there exists a conformation $(s_1, \ldots, \hat{s}_i, \ldots, s_n) \in (D_1 \times \cdots \times \hat{D}_i \times \cdots \times D_n)$ such that

$$E(s_i) - E(s_i') + \sum_{j:j\neq i} [E(s_i, s_j) - E(s_i', s_j)] \leq 0 \tag{4.6}$$

Let n be the size of the protein and $I \subset \{1, 2, \ldots, n\}$ the indices of residues whose rotamers are preserved. By rearranging if necessary, we may assume without loss of generality that $I = \{1, 2, \ldots, m\}$, where $m < n$. Let

$$C^* = (s_1^*, \ldots, s_m^*, s_{m+1}^*, \ldots, s_n^*) \tag{4.7}$$

be the minimum energy conformation conditioned on $s_i = s_i^*$, $i = 1, \ldots, m$, i.e.

$$E(C^*) = \min_{(s_{m+1},\ldots,s_n)\in D_{m+1}\times\cdots\times D_n} E(s_1^*, \ldots, s_m^*, s_{m+1}, \ldots, s_n) \tag{4.8}$$

We need to show that s_j^* is automatically preserved for $j = m+1, \ldots, n$.

Suppose the contrary. Let s_k^* be the first such rotamer being removed. Before s_k^* is removed, C^* is a feasible conformation. For any $s_k' \in D_k \setminus \{s_k^*\}$, consider the conformation $C' = (s_1^*, \ldots, s_k', \ldots, s_n^*)$, i.e. C^* with s_k^* replaced by s_k'. Then C' is also a feasible conformation. By definition of C^*,

$$E(C^*) - E(C') \leq 0 \tag{4.9}$$

After canceling corresponding terms, we have

$$E(s_k^*) - E(s_k') + \sum_{j:j\neq i} [E(s_i^*, s_j^*) - E(s_i', s_j^*)] \leq 0 \tag{4.10}$$

which means s_k^* should be preserved, a contradiction. This completes the proof. $\square$

After DEE, the geometric neighborhood graph is constructed as described at the end of Section 3.2, with the state spaces D_i's replaced by the reduced ones. Then we find a tree decomposition for this graph and run the min-sum algorithm on the junction tree to compute the minimum-energy conformation.

4.4 Test Results for Proteins and Discussions

We first tested our implementation for static SCP on the SCWRL benchmark [6]. Our implementation achieves a performance comparable to that of [16]. The accuracy is slightly worse, but this test serves only as a sanity check here and does not greatly affect our comparison between the static SCP and adaptive SCP.

The test for adaptive SCP goes as follows. For a given protein, we randomly select 10 residues that form a subtree of the largest connected component of the geometric neighborhood graph generated by the static SCP program. Then in the adaptive SCP, all the rotamers of the selected residues are kept from DEE pruning. The rotameric states of the selected residues are randomly changed and the min-energy conformation of the protein is updated.

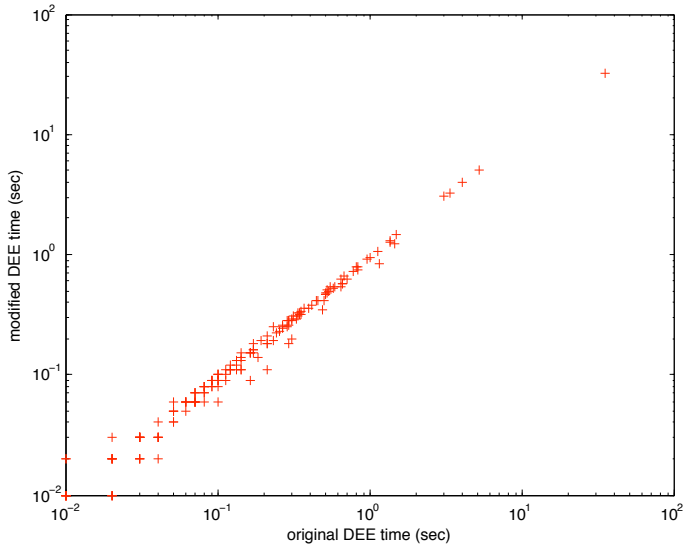


Figure 4.6. Comparison of DEE time.

Fig. 4.6 compares the time for performing the original DEE and the modified one. It is not surprising that both take about the same amount of time, since we only prevented a small fraction of rotamers from being pruned and that should not have a huge impact on the running time. The change from static SCP to adaptive SCP does, however, affect the densities of the resulting graphs and junction trees. The average size of the largest connected components of the geometric neighborhood graph increases from 39 to 49 and the average tree width from 2.8 to 3.6.

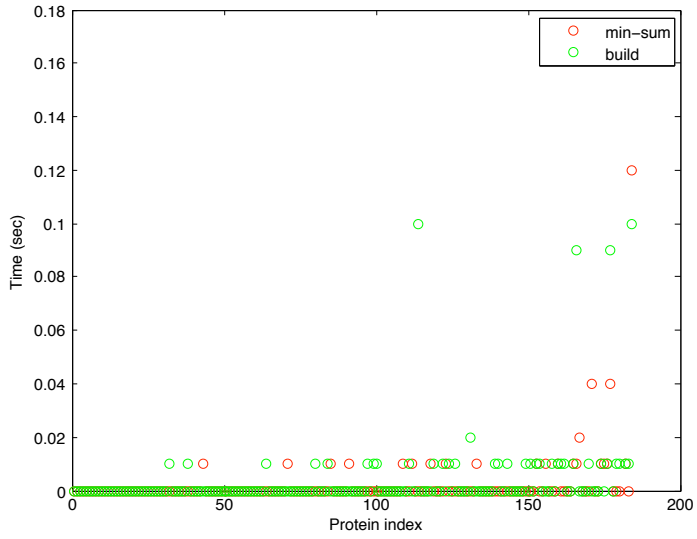


Figure 4.7. Running time for changing states of multiple residues for proteins in SCWRL benchmark.

Fig. 4.7 shows that the time for the original min-sum algorithm and that for building cluster trees are also approximately the same. Fig. 4.8 compares the running time for min-sum and that for updates. In both plots, proteins are indexed in the order of increasing sizes. The average times are 0.0031 second for min-sum, 0.0048 second for building cluster trees and 0.1065 second per 100 updates. Note that for the SCWRL benchmark, the adaptive updates are marginally faster on average than running min-sum from scratch. However, when we examine the running time for individual proteins, we see that for a very large fraction of them, the min-sum time is zero while the updating time is not. This phenomenon may be due to the inaccuracy of

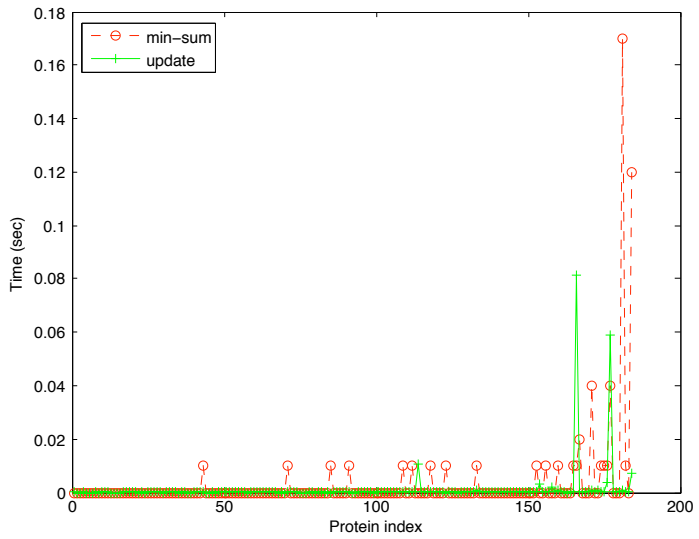


Figure 4.8. Running time for changing states of multiple residues for proteins in SCWRL benchmark.

measurement. For those proteins that give zero min-sum time, the min-sum algorithm terminates within one cycle of the measuring clock, so there are large relative errors in these measurements. When we restrict our test set to the subset of proteins that give nonzero min-sum time, the updates are considerably faster than recomputing from scratch; see Fig. 4.9. We also note that the sizes of the proteins in the SCWRL benchmark are relatively small. All but a few proteins have less than 500 residues and the largest connected components of the geometric neighborhood graphs have less than 150 nodes. Therefore, it is not surprising that we do not observe a huge speed-up for many of them.

Fig. 4.10 gives the average numbers of updated rotameric states for SCWRL proteins. The small numbers of changes might suggest that the rotameric states of the selected sites may not have a great influence on the rotameric states of other residues and the global conformation. However, similar results are observed when we change the states of the allosteric sites of allosteric proteins such as 1ERK. This suggests that the model we used may not be detailed enough to capture such phenomena. If

a better model is found, for example, one that incorporates backbone motions, it is possible to use the adaptive inference to study such phenomena.

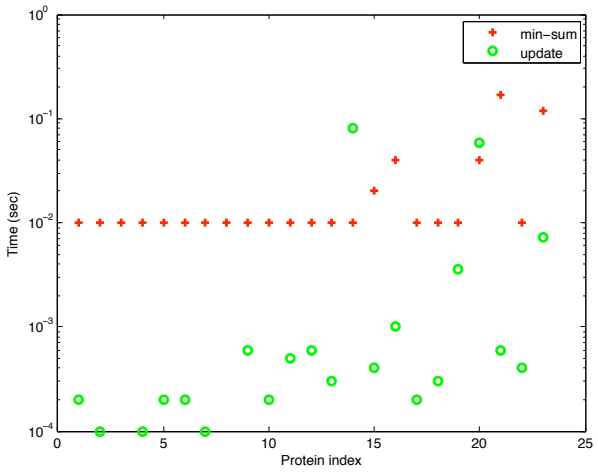


Figure 4.9. Running time for changing states of multiple residues for SCWRL proteins for which min-sum takes nonzero time. The protein indices here are different from that in Figs. 4.7 and 4.8, though still in the order of increasing protein size.

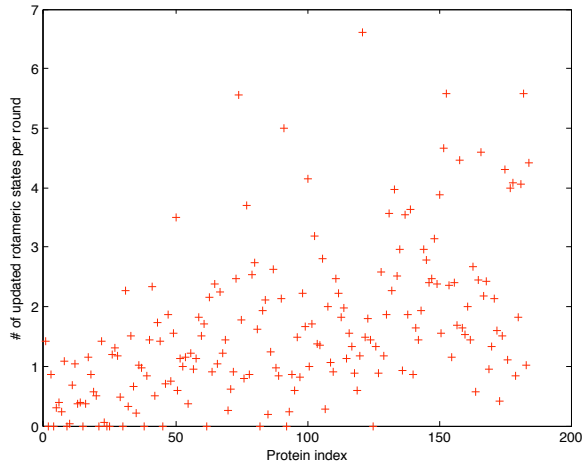


Figure 4.10. Average numbers of updated rotameric states for proteins in the SCWRL benchmark.

CHAPTER 5

CONCLUSIONS

In this thesis, we have presented a C++ library for doing adaptive inference using the cluster tree based framework introduced in [1]. It has been shown in [1] that for an input factor tree of size n, the cluster tree can be built in $O(n)$ time, marginals can be computed in $O(\log n)$ time and updates to factors can be incorporated in $O(\log n)$ time. We have extended this framework by showing that the variable states, e.g. the minimizers, can be computed in $O(n)$ time and maintained in time roughly proportional to the number of changed states. In an experiment on a synthetic benchmark, considerable gains have been observed, in agreement with the theoretical results. This suggests the potential use of the adaptive framework in practice.

We have suggested the potential application of this adaptive inference framework to computational biology, for instance, in the study of ligand binding and allostery. In particular, we have introduced the adaptive side-chain packing problem and a modified dead-end elimination method tailored for it. Experiments have been performed on the SCWRL benchmark, using the protein model in [16]. Although noticeable gains have been observed only for a small fraction of proteins in SCWRL, further analysis shows that this does provide some evidence for the practical usefulness of this framework. The experimental results also suggest that the protein model used are not detailed enough to capture some interesting phenomena, and that it may be essential, for instance, to incorporate backbone motions.

BIBLIOGRAPHY

BIBLIOGRAPHY

[1] U. A. Acar, A. T. Ihler, R. R. Mettu and Ö. Sümer. *Adaptive Bayesian Inference*, 21st Annual Conference on Neural Information Processing Systems, Dec. 2007.

[2] T. Akutsu. *NP-hardness results for protein side-chain packing*, Genome Informatics, Vol. 8, 1997, pp. 180–186.

[3] S. Arnborg, D. G. Corneil and A. Proskurowski. *Complexity of finding embeddings in a k-tree*, SIAM Journal on Matrix Analysis and Applications 8(2), 1987, pp. 277–284.

[4] B. R. Brooks, R. E. Bruccoleri, B. D. Olafson, D. J. States, S. Swaminathan, M. Karplus. "CHARMM: A program for macromolecular energy, minimization, and dynamics calculations". J Comp Chem 4, 1983, pp. 187–217

[5] M. J. Bower, F. E. Cohen and R. L. Dunbrack Jr. *Prediction of protein side-chain rotamers from a backbone-dependent rotamer library: A new homology modeling tool*, J. Mol. Biol. 267, 1268–1282, 1997.

[6] A. Canutescu, A. A. Shelenkov and R. L. Dunbrack Jr. *A graph-theory algorithm for rapid protein side-chain prediction*, Prot. Sci. 12, 2003, pp. 2001–2014.

[7] R. Goldstein. *Efficient rotamer elimination applied to protein side-chains and related spin glasses*. Biophys. J. 66, 1994, pp. 1335–1340

[8] O. Kohlbacher and H. Lenhof. *BALL–Rapid software prototyping in computational molecular biology*. Bioinformatics 16, 9, 2000, pp. 815–824.

[9] F. R. Kschischang, B. J. Frey and H. A. Loeliger. *Factor Graphs and the Sum-Product Algorithm*. IEEE Trans. Info. Theory, Vol. 47, No. 2, Feb. 2001. pp. 498–519.

[10] D. Mount and S. Arya. *ANN: A library for approximate nearest neighbor searching*. In Proceedings of the 2nd CGC Workshop on Computational Geometry, 1997.

[11] K. Murphy. *An Introduction to Graphical Models*. Technical report, Intel Research Technical Report, 2001.

[12] J. Pearl. *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. Morgan Kaufmann, San Francisco, 1988.

[13] N. Pierce and E. Winfree. *Protein design is NP-hard*, Protein Engineering, Vol. 15, No. 10, Oct. 2002, pp. 779–782.

[14] J. W. Ponder and D. A. Case. *Force fields for protein simulations.* Adv. Prot. Chem. 66, 2003, pp. 27–85.

[15] N. Robertson and P. Seymour. *Graph minors. II. Algorithmic aspects of tree-width*, J. Algorithms 7, 1986, pp. 309–322.

[16] J. Xu and B. Berger. *Fast and Accurate Algorithms for Protein Side-Chain Packing*, Journal of the ACM, Vol. 53, No. 4, July 2006. pp. 533–557.

[17] C. Yanover and Y. Weiss. *Approximate Inference and Protein Folding*, Proceedings NIPS, 2002.