ADAPTIVE UPDATES FOR MAP CONFIGURATIONS WITH APPLICATIONS TO BIOINFORMATICS

Umut A. Acar
Toyota Tech. Institute
Chicago, IL
umut@tti-c.org

Alexander T. Ihler
U.C. Irvine
Irvine, CA
ihler@ics.uci.edu

Ramgopal R. Mettu
Univ. of Massachusetts
Amherst, MA
mettu@ecs.umass.edu

Özgür Sümer
Univ. of Chicago
Chicago, IL
osumer@cs.uchicago.edu

ABSTRACT

Many applications involve repeatedly computing the optimal, maximum *a posteriori* (MAP) configuration of a graphical model as the model changes, often slowly or incrementally over time, e.g., due to input from a user. Small changes to the model often require updating only a small fraction of the MAP configuration, suggesting the possibility of performing updates faster than recomputing from scratch. In this paper we present an algorithm for efficiently performing such updates under arbitrary changes to the model. Our algorithm is within a logarithmic factor of the optimal and is asymptotically never slower than re-computing from-scratch: if a modification to the model requires m updates to the MAP configuration of n random variables, then our algorithm requires $O(m \log(n/m))$ time; re-computing from scratch requires $O(n)$ time. We evaluate the practical effectiveness of our algorithm by considering two problems in genomic signal processing, CpG region segmentation and protein sidechain packing, where a MAP configuration must be repeatedly updated. Our results show significant speedups over recomputing from scratch.

Index Terms— dynamic programming, MAP configuration, model updates, CpG region segmentation, protein sidechain conformation

1. INTRODUCTION

Finding the optimal sequence of hidden states, or MAP configuration, from given observations is a classic estimation task typically solved using variants of dynamic programming [1]. In many application of the technique, we often perform repeated computations over a collection of similar problems. For example, in genomic signal processing problems that require determining the effects of mutation (*computational mutagenesis*), each putative mutation gives rise to a new problem that is very similar to the previously solved problem. This suggests that we can update our results significantly faster than recomputing from scratch.

Motivated by this observation, previous work on adaptive inference [2, 3] considered the problem of updating marginals as the model changes. These approaches, however, consider marginal computations rather than MAP computations; moreover, they provide only for efficient "queries" to user-specified variables, but as we do not know *a priori* which variable configurations will be changed we cannot use this framework directly.

In this paper we describe an algorithm for efficiently updating the MAP configurations as the underlying model changes. For a graphical model with n nodes, if a modification requires that the MAP configuration be updated at m positions, then our algorithm takes expected $\Theta(m \log(n/m))$ time (where expectations are taken over internal randomization). Re-running dynamic programming, in contrast, could require up to $\Theta(n)$ time regardless of the number of changes actually introduced (Sec. 2). Consequently, our algorithm provides fast updates when the number of required updates is small, e.g., $\Theta(\log n)$ time when a constant number of updates is required, and never requires more than recomputing from scratch, i.e., takes $\Theta(n)$ time when $m = n$.

We apply our algorithm to two problems from computational biology. For CpG island detection, we use an HMM model to segment regions containing a high frequency of CG nucleotides, and use our adaptive algorithm to track updates to the island(s) as mutations are introduced. In protein sidechain packing, we use a higher-order model to compute the effect of small structural changes on a protein's minimum-energy sidechain conformation. In both cases our algorithm can perform updates orders of magnitude faster than recomputing from scratch.

2. BACKGROUND

Graphical models provide a useful formalism for describing structure within a probability distribution or energy function defined over variables $X = [x_1, \ldots, x_n]$. This structure can then be used to design efficient estimation algorithms, for example finding the maximum *a posteriori* (MAP) or most likely configuration of the variables. We assume discrete-valued x_i throughout the paper.

Factor graphs [4] describe the factorization structure of a function $g(X)$ using a bipartite graph consisting of *factor* nodes and *variable* nodes. Specifically, suppose such a graph G consists of factor nodes $F = \{f_1, \ldots, f_m\}$ and variable nodes $X = \{x_1, \ldots, x_n\}$, and let $X_j \subseteq X$ denote the neighbors of factor node f_j. Then, G is said to be consistent with a function $g(\cdot)$ if and only if $g(X) = \prod_j f_j(X_j)$. For example, a hidden Markov model (HMM) corresponds to a factor graph with $f_t(x_t, x_{t-1}) = p(x_t|x_{t-1})p(o_t|x_t)$, the product of the transition probability distribution and the likelihood of the observed values o_t at each time t. In the case of a singly-connected graph such as an HMM or a tree, the optimal sequence of states $[x_1^* \ldots x_n^*]$ can be found using dynamic programming [1]. This proceeds in two passes, first computing a *cost to go* function recursively from leaves to root (in an HMM, for example, n to 1), then selecting the optimal values outward from root to leaves (1 to n).

Changes to the model, however, may take up to linear time to incorporate. If we are unlucky, the change may occur at a leaf which is $O(n)$ distance from the root. In this case, we must recompute $O(n)$

Acar is partially supported by a gift from Intel.
Mettu is supported by a National Science Foundation CAREER Award (IIS-0643768).

cost to go messages before we can determine whether the root configuration has changed, and $O(n)$ selections to determine whether any configurations have changed. Thus, even when very little about the problem or solution has changed, we may need to do as much work as to re-solve the problem from scratch.

3. EFFICIENTLY UPDATING MAP CONFIGURATIONS

To perform efficient updates under modifications, we construct a *cluster tree* that represents certain intermediate computations. For a change to the model that induces m changes to the MAP configuration, we give an algorithm that uses the cluster tree to efficiently updates the MAP configuration in $O(m \log n/m)$ time. Our algorithm requires no *a priori* knowledge of m or the positions in which the MAP configuration requires updates.

3.1. Cluster Trees and MAP configurations

A *cluster tree* is a representation of the original factor graph constructed by an automated, partially random clustering procedure. At a high level, the cluster tree is an elimination ordering for the input model that preserves intermediate computations, using little more time and space than dynamic programming. In particular, we show the following:

Theorem 1. *Given a factor graph G with n nodes, the cluster tree and associated MAP configuration can be computed in expected $O(n)$ time. Furthermore the cluster tree has size $O(n)$, and expected depth $O(\log n)$.*

For a factor graph $G = (X + F, E)$, a *cluster C* is simply a set of connected vertices in G. To construct the cluster tree, we require as input a spanning tree T of G, and proceeds in rounds, replacing nodes of T with degree at most 2 by a “cluster” associated with their neighbor(s), which are then joined by an edge. When a node v is replaced, we denote its cluster by $\bar{v}$; any clusters adjacent to v are removed and become the children of $\bar{v}$ in the cluster tree. Each cluster thus represents a connected subtree of the original graph. Fig. 1 shows a clustering and its associated cluster tree. In [3], we established that the construction of a cluster tree with similar properties for the purpose of computing marginals for any given variable in the model. In the following discussion we show this definition can be modified for the computation of MAP configurations.

For a cluster $\bar{v}$, define the *boundary* $\partial\bar{v}$ of $\bar{v}$ as the set of edges with exactly one endpoint in $\bar{v}$, and the *boundary variables* $X_{\bar{v}}$ of $\bar{v}$ as the set of variables (variable nodes) incident to the boundary edges. The boundary variables $X_{\bar{v}}$ form a Markov blanket for the subtree represented by $\bar{v}$, i.e., given the configuration of these variables, optimization of the nodes in $\bar{v}$ does not depend on the rest of the graph. Moreover, because the clusters are nested, the Markov blanket can be computed recursively in the cluster tree: if the children of $\bar{v}$ are $\{\bar{u}_1, \ldots, \bar{u}_k\}$, the boundary of $\bar{v}$ is

$$\partial\bar{v} = E(v) \triangle \partial\bar{u}_1 \triangle \ldots \triangle \partial\bar{u}_k, \qquad (1)$$

where $E(v)$ are the edges incident to v in the factor graph and $\triangle$ is the set symmetric difference operator. Using the boundary, it is easy to compute the boundary variables. The *cluster function* of a cluster $\bar{v}$, written $\varphi_{\bar{v}}$, is the partial maximization of the factors within that cluster, over all variables except the boundary variables. This too can be computed recursively

$$\varphi_{\bar{v}}(X_{\bar{v}}) = \max_{\overline{X}_{\bar{v}}} \; \psi_v(X_v) \prod_{i=1}^{k} \varphi_{\bar{u}_i}(X_{\bar{u}_i}). \qquad (2)$$

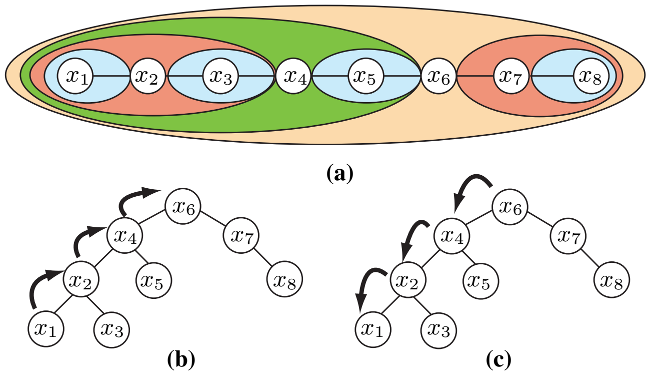

Fig. 1. **(a)** An example clustering of a Markov chain (with factor node suppressed for space). **(b)-(c)** The MAP can be computed on the cluster tree by computing cluster functions in an upward pass, then assigning values in a downward pass. Note that the ancestors of any subtree include its Markov blanket.

where $\overline{X}_{\bar{v}} = (X_v \cup X_{\bar{u}_1} \cup \ldots \cup X_{\bar{u}_k}) \setminus X_{\bar{v}}$, and the ψ_u refer generically to the factors of $g(\cdot)$, equal to $f_j(X_j)$ if $v = f_j$, and a constant if $v = x_i$. Using these recursive definitions, we can compute all the cluster functions in the cluster tree in $O(n)$ time by beginning with the leaves and working our way upward, as illustrated in Fig. 1(b). In standard dynamic programming, the time to compute individual cost-to-go functions requires time exponential in the treewidth of the model. Computing cluster functions incurs a slightly larger constant factor that is exponential in the “cut size” [3] of the input spanning tree. For example, for variables of dimension d in an HMM, our approach requires $O(d^3)$ time to compute cluster functions, versus $O(d^2)$ for dynamic programming. As we show in Sec. 4, this additional constant factor does not affect the performance of our algorithm, since the model size, n, is the dominant parameter.

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 forth. When this top-down recursion reaches cluster $\bar{v}$, we choose the optimal configuration for the variables in $\overline{X}_{\bar{v}}$ using

$$\overline{X}^*_{\bar{v}} = \arg\max_{\overline{X}_{\bar{v}}} \psi_v(X_v) \prod_{i=1}^{k} \varphi_{\bar{u}_i}(X_{\bar{u}_i}) \delta(X_{\bar{v}} = X^*_{\bar{v}}). \qquad (3)$$

where $\delta(\cdot)$ is the Kronecker delta, ensuring that $X_{\bar{v}}$ takes on value $X^*_{\bar{v}}$. By the recursive nature of the computation, we are guaranteed that the optimal configuration $X^*_{\bar{v}}$ is selected before reaching the cluster $\bar{v}$. This can be proven inductively: assume that $X_{\bar{v}}$ has an optimal assignment when the recursion reaches the cluster $\bar{v}$. We are thus conditioning on the Markov blanket for $\bar{v}$, and can optimize the subtree of $\bar{v}$ independently. The value in (3) is thus the optimal configuration for $\overline{X}_{\bar{v}}$, which by definition includes the boundary variables $X_{\bar{u}_i}$ for each of $\bar{v}$'s children. This property can be seen in Fig. 1(c).

3.2. Updating MAP Configurations Under Changes

In this section we describe how MAP configurations may be efficiently maintained in the face of changes to the underlying model. The key to our approach is the ability to modify the underlying factor graph and efficiently update the MAP configurations. More specifically, we can modify the factor graph by changing the factors themselves or by changing the structure of the graph by inserting/deleting edges or nodes.

Perhaps surprisingly, the time required to perform an update is proportional to the number of modifications in MAP configuration plus the number of model changes and the cost gracefully degrades to linear in the number of nodes in the factor graph, ensuring that changes to many factors or configurations 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.

3.2.1. Modifications to Factors

Let G be a factor graph and let T be its cluster tree. Suppose that we modify some factor nodes $v_1, \ldots, v_l$ in G, e.g., by changing relative probabilities (factor values). We update the cluster tree and MAP configurations in two passes. In the bottom-up pass, we update cluster functions and the boundary variables of the ancestors of clusters $v_1, \ldots v_l$ by recomputing the cluster functions as described in Sec. 3.1. We additionally mark each such cluster *dirty* to indicate that it has been modified. In the top-down phase, we search for changes to and update the optimal configuration for the variables of each cluster. We make this high-level description precise and prove that updates may be performed efficiently in the following theorem.

Theorem 2. *Suppose that we modify l factors of a cluster tree, and that any MAP configuration of the new model differs from our previous result on at most m variables. Define $\gamma = \min(l + rm, n)$, where r is the maximum degree of any node in G. Then, the m variables and their new MAP configuration can be found in expected $O(\gamma(1 + \log \frac{n}{\gamma}))$ time.*

Proof Sketch: To update the cluster tree, we first change the cluster functions above each modified factor f_j, on the path from f_j to the root. Then, to find the new MAP, we revisit our decision for the configuration of any variables along these routes.

Consider how we can rule out any changes in the MAP configuration of a subtree rooted at $\bar{v}$. Assume that we have found all changed configurations above $\bar{v}$. The decision at $\bar{v}$ is based on the cluster functions from its children and the configuration of its boundary variables. The boundary variables form a Markov blanket for the cluster $\bar{v}$: if none of these variables have changed, and nothing internally has changed, then the configuration for all nodes in $\bar{v}$ remains valid.

Thus, the following procedure finds a new MAP: beginning at the root, we move downward along the paths to each modified f_j, checking for a MAP change. At any changed node, we also check that node's children. At any unchanged node, however, we recurse only on any children which were degree-two when removed, and were at the time of removal adjacent to a changed node. (This connectivity information can be tracked easily by augmenting the cluster tree.) As a property of the cluster tree's construction, there can be at most one such path of "unchanged but checked" nodes per child of a changed node.

Now suppose that m nodes have changed value. The total number of paths checked is then at most $l + rm$. These paths are of expected height $O(\log n)$, and every node is checked at most once, ensuring that the total number of nodes visited is at most $O(\gamma(1 + \log \frac{n}{\gamma}))$ where $\gamma = \min(l + rm, n)$. □

As an example, consider the graph in Fig. 1(b)-(c). We update the cluster functions in the upward pass, then check for changes back down. If, for example, x_6 and x_4 are unchanged, we can reason that x_5 must also be unchanged.

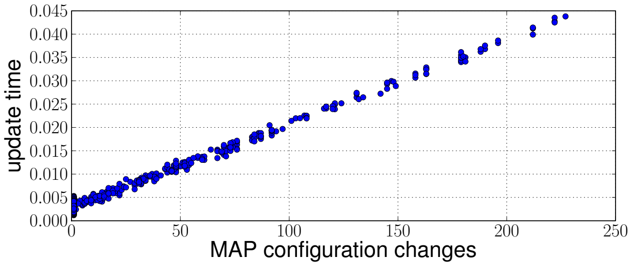

Fig. 2. **Update times.** Time to update MAP configurations for a DNA sequence with CpG islands. Our algorithm takes time proportional to the number of changes to the MAP configuration, rather than the size of the HMM.

3.2.2. Modifications to Structure

It is also possible, using essentially the same procedure, to modify the *structure* of the graph, adding or removing nodes and edges. Suppose that we modify G by inserting/deleting a total of l edges and nodes as desired to obtain a new graph G'. As with updates to the factors, these changes can be made in $O(\gamma(1 + \log \frac{n}{\gamma}))$ time, with γ defined as before.

The primary difference between factor and structural changes is that, in addition to the ancestors of the changes, there are potentially a small number of additional nodes who are affected by the change. These correspond to nodes that are in close proximity to an affected node during the clustering process; again, tracking this information requires only some additional bookkeeping during clustering. The growth of this set is limited by the local nature of decisions during the clustering process, and can be shown to be bounded by a constant at each round [5]. Thus the number of nodes visited remains at most $O(\gamma(1 + \log \frac{n}{\gamma}))$.

4. APPLICATIONS TO COMPUTATIONAL BIOLOGY

We implemented our algorithm by using a combination of Python and C++ and applied it to two computational biology applications: genomic signal processing with hidden Markov models (HMM), and protein structure determination using higher-order statistical models. For both applications, we show that our adaptive approach yields significant speedups that would be valuable in practice.

4.1. Sequence Analysis with Hidden Markov Models

HMMs are a widely-used tool to analyze DNA sequences [6]; typically an HMM is trained using a sequence with known function or annotations, and new sequences are analyzed by inferring hidden states in the resulting HMM. In this context our algorithm for updating MAP configurations can be used to study the effect of changes to to the model and observations on hidden states of the HMM. We can use an HMM to identify "CpG" islands in DNA [7], which are regions of DNA with higher distributions of cytosine and guanine that are believed to regulate upstream gene expression. For each nucleotide in the sequence being analyzed, our HMM has a hidden state that represents whether that nucleotide is in a CpG island or not. A MAP configuration of the hidden states in this model then identifies CpG islands in the given sequence. In mutagenesis applications, we would like to repeatedly update the inferred location of CpG islands as we modify the sequence DNA sequence (e.g. due to methylation or evolution). This has potential application, for example, in studying the evolution of gene expression.

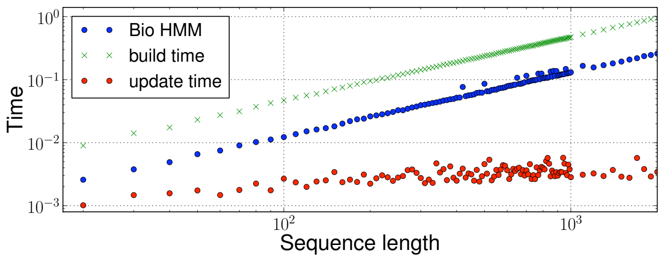

Fig. 3. **Runtime of our algorithm.** A log-log plot of the time to update MAP configurations as the HMM size increases; we see a logarithmic trend for our algorithm.

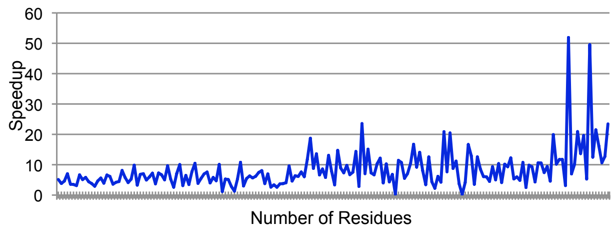

Fig. 4. **Speedup for SCWRL benchmark.** The x-axis shows proteins sorted by size; the speedup varies due to the diversity of protein folds in SCWRL, but on average our approach is about 8 times faster than computation from scratch.

Since repeated modifications to the DNA sequence typically cause small updates to the CpG-islands (MAP) configurations, we expect to gain significant speedup by using our algorithm. To test this hypothesis, we compared the time to update MAP configurations in our algorithm against the Bio.HMM package in BioPython; we used training parameters from [6]. We used a 3000-base pair stretch of DNA from chromosome 21 of the human genome (contig NT.030188), which contains three CpG islands, and observed the effect of 10,000 random mutations. After each mutation we updated the respective MAP configurations (i.e. indicating CpG island position); we observed the islands shift, disappear and reappear. We see in Fig. 2 that the time to update a MAP configuration scales with number of actual updates, rather than the length of the sequence. Similarly, Fig. 3 shows that as the sequence size increases, update times grows logarithmically, with our algorithm running about an order of magnitude faster than computation from scratch.

4.2. Protein Structure Analysis with Factor Graphs

The molecular structure of proteins defines the biochemical basis for biological processes. Determining protein structure experimentally, however, is difficult and computational methods are often employed. For the *protein sidechain packing problem*, we are given a three-dimensional backbone for a protein P consisting of n amino acids, each of which can take on one of a discrete number of states called *rotamers* [8]. Our goal then is to find a conformation (i.e. choice of rotamers) with minimum physical energy. Recently, graphical models have been used to accurately predict sidechain conformations [9]; these models are similar to an HMM, but with additional long-range interactions depending on the amino acid sequence and backbone structure. To construct the model we represent amino acids with variable nodes, whose states correspond to rotamers. Then, we define factors for the *a priori* likelihood of a rotamer according to its interactions with the backbone, and pairwise factors betwen rotamers using their pairwise energies[1]. In the resulting model, a MAP configuration corresponds to a minimum-energy sidechain conformation.

Conformational changes in proteins occur due to, for example, ligand binding and allostery in proteins; these changes typically involve changes to a small set of amino acid sidechains. Such conformational change commonly regulates activity in many enzymes, transcription factors and signaling proteins. Our algorithm for adaptive MAP computation is a natural choice in such applications, since it can handle arbitrary factor graphs.

For our experiments, we used the SCWRL [8] benchmark, which contains about 180 proteins of varying sizes (31–910 residues) and backbone folds. With some optimization (i.e. reducing the state space of the models using dead-end elimination), we were able to obtain accurate models with cut size of about 6 on average. For the cluster tree corresponding to each protein, we selected a set of 10 randomly chosen amino acids for modification. We then measured the time taken for inference from scratch versus our algorithm. For each protein we applied updates to a random subset of a selected set amino acids by choosing a random rotameric state for each. We then recorded the time taken to compute a new minimum-energy conformation. We ran 1000 such updates for each protein, and found that our algorithm was on average 8.18 times faster than computation from scratch (see Fig. 4). As expected, we found that the minimum-energy conformation actually does not change substantially after the updates, with typically no more than 5 additional amino acids requiring a new rotamer in the computed minimum-energy conformation.

[1]We compute probability of a conformation C_P by using a natural correspondence between energy functions and probabilistic models, given by the Boltzmann distribution: $p(C_P) \propto \exp(-\beta E(C_P))$.

5. REFERENCES

[1] D. P. Bertsekas, *Dynamic Programming and Optimal Control*, vol. 1, Athena Scientific, Belmont, MA, 1995.

[2] A. L. Delcher, A. J. Grove, S. Kasif, and J. Pearl, "Logarithmic-time updates and queries in probabilistic networks," *J. AI Res.*, vol. 4, pp. 37–59, 1995.

[3] U. A. Acar, A. T. Ihler, R. R. Mettu, and Ö. Sümer, "Adaptive Bayesian inference in general graphs," in *UAI*, 2008, pp. 1–8.

[4] F. Kschischang, B. Frey, and H.-A. Loeliger, "Factor graphs and the sum-product algorithm," *IEEE Trans. Inform. Theory*, vol. 47, no. 2, pp. 498–519, Feb. 2001.

[5] U. A. Acar, G. E. Blelloch, R. Harper, J. L. Vittes, and M. Woo, "Dynamizing static algorithms with applications to dynamic trees and history independence," in *ACM-SIAM SODA*, 2004.

[6] R. Durbin, S. Eddy, A. Krogh, and G. Mitchison, *Biological sequence analysis*, Press, Cambridge U., 11th edition, 2006.

[7] Gardiner-Garden M. and Frommer M., "CpG in vertebrate genomes," *J Mol Biol*, vol. 196, no. 2, pp. 261–282, July 1987.

[8] A. A. Canutescu, A. A. Shelenkov, and R. L. Dunbrack Jr., "A graph-theory algorithm for rapid protein side-chain prediction," *Protein Sci.*, vol. 12, no. 9, pp. 2001–2014, Sep 2003.

[9] C. Yanover and Y. Weiss, "Approximate inference and protein folding," in *NIPS*, 2002, pp. 84–86.