Journal of Machine Learning Research 9 (2008) 651-682 Submitted 1/06; Revised 10/07; Published 4/08

Graphical Models for Structured Classification, with an Application to Interpreting Images of Protein Subcellular Location Patterns

Shann-Ching Chen SHANNCC@ANDREW.CMU.EDU
Department of Biomedical Engineering
Carnegie Mellon University
Pittsburgh, PA 15213, USA

Geoffrey J. Gordon GGORDON@CS.CMU.EDU
Machine Learning Department
Carnegie Mellon University
Pittsburgh, PA 15213, USA

Robert F. Murphy MURPHY@CMU.EDU
Departments of Biological Sciences, Biomedical Engineering, and Machine Learning
Carnegie Mellon University
Pittsburgh, PA 15213, USA

Editor: Nir Friedman

Abstract

In structured classification problems, there is a direct conflict between expressive models and efficient inference: while graphical models such as Markov random fields or factor graphs can represent arbitrary dependences among instance labels, the cost of inference via belief propagation in these models grows rapidly as the graph structure becomes more complicated. One important source of complexity in belief propagation is the need to marginalize large factors to compute messages. This operation takes time exponential in the number of variables in the factor, and can limit the expressiveness of the models we can use. In this paper, we study a new class of potential functions, which we call decomposable k-way potentials, and provide efficient algorithms for computing messages from these potentials during belief propagation. We believe these new potentials provide a good balance between expressive power and efficient inference in practical structured classification problems. We discuss three instances of decomposable potentials: the associative Markov network potential, the nested junction tree, and a new type of potential which we call the voting potential. We use these potentials to classify images of protein subcellular location patterns in groups of cells. Classifying subcellular location patterns can help us answer many important questions in computational biology, including questions about how various treatments affect the synthesis and behavior of proteins and networks of proteins within a cell. Our new representation and algorithm lead to substantial improvements in both inference speed and classification accuracy.

Keywords: factor graphs, approximate inference algorithms, structured classification, protein subcellular location patterns, location proteomics

1. Introduction

In standard supervised classification problems, the label of each test instance is independent of the labels of all other instances. In some problems, however, we may receive multiple test instances at a time, along with side information about dependences among the labels of these instances. For

example, if each instance is a handwritten character, the side information might be that the string of characters forms a common English word; or, if each instance is a microscope image of a cell with a certain protein tagged, the side information might be that several cells share the same tagged protein. To solve such a *structured classification problem* in practice, we need both an expressive way to represent our beliefs about the structure, as well as an efficient probabilistic inference algorithm for classifying new groups of instances.

Unfortunately, the goal of having an expressive language is in direct conflict with the goal of having an efficient inference algorithm: while Markov random fields or factor graphs can represent arbitrary dependences among instances, inference rapidly becomes intractable as the graph structure becomes more complicated (see, e.g., Koller and Friedman, 2007). Simple graphs such as pairwise links arranged in chains or trees lead to efficient inference, but these structures may not allow us to express our beliefs accurately or completely. On the other hand, if we try to couple large groups of labels (either directly, by specifying a factor that links to a large number of labels, or indirectly, by using a graph with large loops), the cost of inference grows exponentially.

To speed up inference, we can move to approximate algorithms such as loopy belief propagation (see, e.g., Koller and Friedman, 2007). Loopy belief propagation handles large loops efficiently, but it does nothing to speed up the task of working with single large factors. In fact, in practical problems, the operation of marginalizing a large factor can easily become the main bottleneck for inference, preventing us from using more-expressive models.

Therefore, in this paper, we study a new class of potential functions, which we call decomposable k-way potentials. Computing messages for these potentials is much more efficient than for general potentials, even though the new potentials can express distributions that cannot be represented by groups of smaller potentials. Accordingly, we believe these new potentials provide a better balance between expressive power and efficient inference than was previously available.

We discuss three instances of decomposable potentials: the associative Markov network potential, the nested junction tree, and a new type of potential which we call the voting potential. We use these potentials to classify images of protein subcellular location patterns in groups of cells. Classifying protein subcellular location patterns is important as a step in solving many practical computational biology problems, particularly in the area of systems biology: for example, it can help in designing high-throughput screening systems for drug discovery, or in conducting experiments to determine the effect of various treatments on the synthesis and behavior of proteins and networks of proteins within a cell. Our new representation and algorithm lead to substantial improvements in both inference speed and classification accuracy.

Preliminary versions of portions of this work have been presented previously (Chen and Murphy, 2006; Chen et al., 2006a,b). These papers describe applications of decomposable potentials and the corresponding fast inference algorithms for segmenting and classifying images of protein subcellular location patterns. But, none of these papers describe the idea of decomposable potentials of Section 4 or the nested inference algorithm of Section 5 in full generality. They also do not cover some of the instances of decomposable potentials described in Section 6; and, the experiments of Sections 8–9 have not been reported previously.

2. Factor Graphs

The factor graph representation of a probability distribution (Kschischang et al., 2001) describes the relationships among a set of variables x_i using local factors or potentials φ_j. Each factor depends

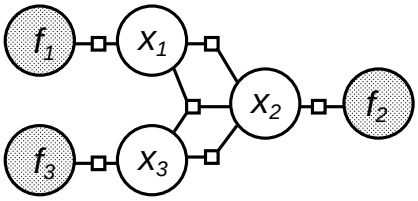


Figure 1: A probability distribution represented as a factor graph. Small squares denote potential functions; for example, this factor graph contains a potential which connects the variables x_1, x_2, and x_3, and another which connects x_1 and f_1.

on only a subset of the variables, and the overall probability distribution is the product of the local factors, together with a normalizing constant Z:

$$P(x) = \frac{1}{Z} \prod_{\text{factors } j} \varphi_j(x_{V(j)}).$$

Here $V(j)$ is the set of variables that are arguments to factor j; for example, if φ_j depends on x_1, x_3, and x_4, then $V(j) = \{1,3,4\}$ and $x_{V(j)} = (x_1, x_3, x_4)$.

Each variable x_i or factor φ_j corresponds to a node in the factor graph. Fig. 1 shows an example: the large nodes represent variables, with shaded circles for observed variables and open circles for unobserved ones. The small square nodes represent factors, and there is an edge between a variable x_i and a factor φ_j if and only if φ_j depends on x_i, that is, when $i \in V(j)$. (By convention the graph only shows factors with two or more arguments. Factors with just a single argument are not explicitly represented, but are implicitly allowed to be present at each variable node.)

The inference task in a factor graph is to combine the evidence from all of the factors to compute properties of the distribution over x represented by the graph. Naively, we can do inference by enumerating all possible values of x, multiplying together all of the factors, and summing to compute the normalizing constant. Unfortunately, the total number of terms in the sum is exponential in the number of random variables in the graph. So, usually, a better way to perform inference is via a message-passing algorithm called belief propagation (BP). Here we briefly review the basics of BP in factor graphs; for more details, see Kschischang et al. (2001).

The basic BP algorithm works on a factor graph which is tree-shaped (i.e., has no cycles). It sends messages from every variable to each of its neighboring factors, and from every factor to each of its neighboring variables. Messages to or from a variable x_i will be vectors whose length is equal to the number of values that x_i can take on.

For a variable x_i with neighboring factors $\varphi_1, \varphi_2, \ldots, \varphi_k$, suppose that x has received messages $m_{j \to i}(x_i)$ from its neighbors φ_j for $j \in \{1 \ldots (k-1)\}$. (To simplify notation, we have numbered x_i's neighbors consecutively starting from 1. This is not a loss of generality since we can always temporarily permute our factor indices to make it so.) Suppose also that x_i has local evidence represented by the one-argument factor $\varphi_i^{\text{loc}}(x_i)$. Then we can compute x_i's message to φ_k as

$$m_{i \to k}(x_i) = \varphi_i^{\text{loc}}(x_i) \prod_{j=1}^{k-1} m_{j \to i}(x_i). \tag{1}$$

That is, we take the componentwise product of all of the messages from factors $1 \ldots k-1$, multiply in the local evidence, and send the result to φ_k. The normalization of the message is arbitrary, so for convenience or numerical precision we may multiply each component of the message by an arbitrary constant.

Similarly, suppose that a factor φ_j has neighbors $x_1, x_2, \ldots, x_k$ (again numbered consecutively without loss of generality) and has received messages $m_{i \to j}(x_i)$ for $i \in \{1 \ldots (k-1)\}$. Then we can compute φ_j's message to x_k as

$$m_{j \to k}(x_k) = \sum_{x_1} \sum_{x_2} \cdots \sum_{x_{k-1}} \varphi_j(x_1, \ldots, x_k) \prod_{i=1}^{k-1} m_{i \to j}(x_i). \tag{2}$$

Unlike Equation 1, in Equation 2 we must marginalize out variables other than x_k by summing over their possible values.

BP works by picking an arbitrary node of the graph as root and then making two passes over the tree: first it passes messages inward from the leaves to the root, and then outward from the root to the leaves. When the message passing finishes, the posterior marginal probability of any random variable x_i is just the componentwise product of its local potential and all of its incoming messages:

$$P(x_i) = \varphi^{\text{loc}}(x_i) \prod_{\{j | i \in V(j)\}} m_{j \to i}(x_i). \tag{3}$$

2.1 Working with General Graphs

The above discussion assumed that our factor graph was tree-shaped. For graphs with loops, we have two alternatives: first, we can collapse groups of variable nodes together into combined nodes, which can turn our graph into a tree and allow us to run BP as above. Second, we can run an approximate inference algorithm that doesn't require a tree-shaped graph. We can also combine these alternatives if desired, grouping variable nodes to reduce the number of loops in the graph so that our approximate inference becomes more accurate.

When we combine a set of variable nodes, the new node represents all possible settings of all of the original nodes. E.g., if we collapse a variable x_1 that has settings T, F with a variable x_2 that has settings A, B, C, then the combined variable x_{12} has settings TA, TB, TC, FA, FB, FC. When we collapse a set of variables, we need to alter the neighboring factor nodes: any factor adjacent to any of the original nodes becomes a neighbor of the new combined node, and its list of arguments is extended if necessary to include all variables in the collapsed set.

The advantage of collapsing variable nodes is that it can allow us to simplify the structure of our graph: if in the collapsed graph there are two factor nodes with the same set of arguments, then we can combine them by multiplying their potentials elementwise. For example, Fig. 2 shows the result of collapsing x_1 and x_2 in the factor graph of Fig. 1. The potentials φ_{23} and φ_{123} from the original graph have the same set of neighbors in the new graph, and so can be combined into one factor node. Similarly, the local potentials φ_1^{loc} and φ_2^{loc} can be combined with the factor φ_{12} to form a new local potential at the collapsed node x_{12}. Notice that the new factor graph is tree-shaped, even though the original one had loops.

When removing loops from a factor graph, we may wish to include each original variable in more than one combined node. We are free to do so as long as we adjust our potentials to enforce the constraint that all copies of the variable must agree on its value. That is, when any two copies

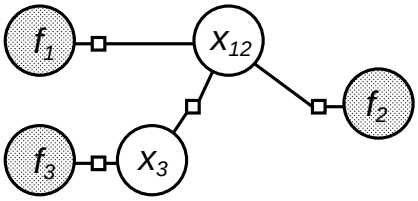


Figure 2: Another factor graph that can represent the same probability distribution as the graph of Fig. 1. The new graph was derived by collapsing together the nodes for variables x_1 and x_2 from Fig. 1.

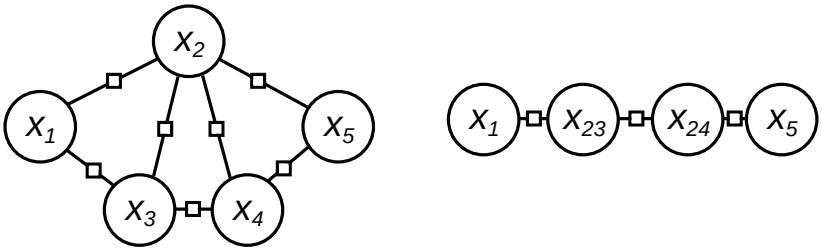


Figure 3: A factor graph with multiple loops (left) and a tree derived from the factor graph by collapsing pairs of nodes (right).

disagree, at least one potential must be zero. The cost of storing or working with such a hard-constraint potential depends only on the number of *distinct* variables in its argument list.

For example, in the graph of Fig. 3, we could form a tree by collapsing x_2, x_3, and x_4 into a single node. The resulting graph would have variables x_1, x_{234}, and x_5, and factors φ_{1234} and φ_{2345}. But, we can form a tree with smaller factors if we group x_2 with x_3 and also separately with x_4: the resulting graph has variables x_1, x_{23}, x_{24}, and x_5, and factors φ_{123}, φ_{234}, and φ_{245}. The potential φ_{234} encodes the constraint that the settings of x_{23} and x_{24} must agree on the assignment to x_2.

A factor graph that has been reduced to a tree is equivalent to a *junction tree*. A junction tree is a connected acyclic graph whose nodes are labeled with sets of variables in a way that satisfies the *running intersection property*: if a variable is present at two nodes A and B, it is also present at all nodes along the (unique) path connecting A and B in the tree. The factor nodes in a tree-shaped factor graph correspond to nodes of the junction tree, with labels equal to their argument sets. The collapsed variable nodes correspond to edges of the junction tree. The groups of original variables at each collapsed variable node (such as $\{x_2, x_4\}$ in the figure) are called *separators*, since conditioning on all of the variables in a separator is sufficient to separate the factor graph into two or more disconnected pieces. The running intersection property ensures that we can define potentials that constrain all copies of a variable to agree.

If we collapse our factor graph all the way to a tree, we can do inference with the exact BP algorithm from above. If we leave some loops, a similar BP algorithm can still work: we can initialize all messages to be uniform, then start at an arbitrary node and use the same formulas as before

for message calculation (Equations 1–2). However, we may have to update each message several times before the marginals converge. The version of the algorithm that updates messages repeatedly until convergence is called loopy BP, or LBP. Inference with LBP is approximate because it can double-count evidence: messages to a node i from two nodes j and k can both contain information from a common neighbor l of j and k. Several researchers have empirically demonstrated that when LBP converges, the posterior marginal probabilities from Equation 3 often approximate the true marginals well (Murphy et al., 1999; McEliece et al., 1998; Zhang and Chang, 2004). If LBP oscillates between some steady states and does not converge, we can stop the process after some number of iterations; in this case, the approximate posteriors will usually be inaccurate. Although oscillations can be avoided by using "momentum" (Murphy et al., 1999), which replaces the messages that were sent at time t with a weighted average of the messages at times t and $t-1$, in some cases the approximate posteriors are still inaccurate (Murphy et al., 1999). The convergence of LBP depends on the exact graph structure and on the type and strength of the factors involved (Pearl, 1988; Heskes, 2004). Recently, researchers have developed sufficient conditions for the convergence of LBP (Weiss, 2000; Tatikonda and Jordan, 2002; Ihler et al., 2005), and a measurement of message errors has been proposed (Ihler et al., 2005).

For either exact or loopy BP, the runtime for each pass over the factor graph is exponential in the number of distinct original variables included in the largest factor. So, inference can become prohibitively expensive if our factors are too large, either because they were too large in the original graph or because we merged too many variables.

3. Factor Graphs and Structured Classification

To use belief propagation to solve structured classification problems, we need two things: a local classifier for individual instances, and a factor graph which encodes our prior beliefs about likely arrangements of instance labels. The local classifier tells us the likelihood of individual test examples under each possible class assignment, while the factor graph tells us how to trade off evidence at one example against evidence at another. We can learn local classifiers in a number of ways; for the cell image classification experiments below, we use standard support vector machines, together with a post-processing step that allows us to interpret the SVM outputs as probabilities.

There are also a number of ways to construct factor graphs that encode our beliefs about likely label vectors. In our experiments below, we construct a factor graph in two steps: first we use domain-specific heuristics to identify pairs of examples whose labels are likely to be the same, and use these pairs to build a *similarity graph* with an edge between each such pair of examples. Then, we use this similarity graph to decide what potentials to add to our factor graph. (In the protein subcellular location pattern classification problem, the similarity graph edges come from either physical proximity or similarity in appearance.) Given the similarity graph, we compare factor graphs built from several different types of potentials; the following sections introduce these potentials and discuss their advantages and disadvantages.

3.1 The Potts Potential

The simplest potential function is the Potts potential. The Potts potential is a pairwise (i.e., two-argument) factor which encourages two nodes x_i and x_j to have the same label:

$$\varphi(x_i, x_j) = \begin{cases} \omega & x_i = x_j \\ 1 & \text{otherwise.} \end{cases} \tag{4}$$

Here $\omega > 1$ is an arbitrary parameter which expresses how strongly we believe that x_i and x_j have the same label. If we use one Potts potential for each edge in the similarity graph, the overall probability of a vector of labels x is

$$P(x) = \frac{1}{Z} \prod_{\text{nodes } i} P(x_i) \prod_{\text{edges } i,j} \varphi(x_i, x_j) \tag{5}$$

where we have written Z for a normalizing constant and $P(x_i)$ for the probability which the base classifier assigns to label x_i for node i. Equations 4–5 form what is known as a *Potts model*.

Unfortunately, the Potts model does not perfectly capture our desired intuition about inference from labels of neighboring cells. To see why, consider a two-class prediction problem where node x_i has k_A neighbors of class A and k_B neighbors of class B, and suppose that classes A and B have equal prior probability for x_i given the classifier output. In this situation the ratio of posterior probabilities for classes A and B will be $\omega^{k_A - k_B}$. So for example, if ω is 2, and if x_i has 1 neighbor of class A and 3 of class B, then the ratio of probabilities will be $2^{1-3} = 1/4$. So, class A will be 1/4 as likely as class B, and $P(x_i \text{ has label } A) = 0.2$.

However, if x_i has 7 neighbors of class A and 9 neighbors of class B, the posterior probability of class A will still be 0.2, even though our intuition tells us that the probability should be much closer to 0.5 in this case. The same will hold whenever there are 2 more neighbors of class B than class A, even if the counts are 107 and 109. Worse, as class B's majority gets larger, the ratio of probabilities will approach 0 exponentially fast. So, a sufficiently strong vote from x_i's neighbors will quickly overwhelm any evidence at x_i itself—an undesirable situation.

The source of this problem is that, in Equation 5, the evidence from separate potentials has to combine multiplicatively. So, as long as the evidence from different neighbors acts through separate potentials, we will see an exponential dependence between the number of neighbors of a given class and the probability of that class. We can reduce the severity of this problem by choosing ω to be only a little bit larger than 1. But, to fix the problem we need to move to potential functions that depend on $k > 2$ nodes. Our experimental results, below, will show that potentials that combine evidence additively can perform better than the Potts potential over a wider range of inference problems.

3.2 The Voting Potential

To capture the intuition that we should be less certain about nodes whose neighbors split relatively evenly, we propose a new potential which we call the *voting potential*. In the voting potential, a node's classification is influenced by the *proportion* of classes among its neighbors, rather than the difference in class counts as in the Potts model.

The voting potential has one distinguished argument, called the *center*; the remaining arguments are called *voters*. In the factor graphs for our cell classification experiments below, we use one voting potential per cell j; the center for the jth potential is cell j itself, and the voters are the cells that are adjacent to j in the similarity graph. We will write $N(j)$ for the set of similarity-graph neighbors of cell j, so that the jth potential depends on variables $V(j) = \{j\} \cup N(j)$.

x_1	x_2	x_3	φ
0	0	0	3/4
0	0	1	1/2
0	1	0	1/2
0	1	1	1/4
1	0	0	1/4
1	0	1	1/2
1	1	0	1/2
1	1	1	3/4

Table 1: An example of the voting potential with parameter $\lambda = 2$ for $n = 2$ classes. The center node x_1 has two neighbors, x_2 and x_3.

We define the voting potential as follows:

$$\varphi_j(x_{V(j)}) = \frac{\lambda/n + \sum_{i \in N(j)} I(x_i, x_j)}{|N(j)| + \lambda}. \tag{6}$$

Here n is the number of classes, λ is a smoothing parameter, and I is an indicator function:

$$I(x_i, x_j) = \begin{cases} 1 & \text{if } x_i = x_j \\ 0 & \text{otherwise.} \end{cases}$$

(The normalization constant in the denominator of Equation 6 is irrelevant to inference, and is included only for ease of interpretation.) An example of the voting potential is given in Table 1.

The voting potential function combines the evidence from all of node x_j's neighbors into a summary vote which then influences x's classification. The parameter λ controls how much weight we put on a vote from a small number of neighboring cells. We can interpret it as the size of an additional set of fictitious neighbors whose votes are distributed uniformly; this trick limits the influence of a vote from a small number of neighbors, and is called Laplace smoothing. For example, looking at the first and fifth rows of the table, we can see that when both of x_1's neighbors are 0, x_1 is 3 times as likely to be 0 as 1, since there are $2+1$ votes for $x_1 = 0$ and $0+1$ votes for $x_1 = 1$.

The behavior of the voting potential contrasts with that of the Potts potential described in Section 3.1, in which each neighbor separately influences the center node without reference to the other neighbors. In line with our intuition, we will see below that networks which use the voting potential can yield more accurate results than the Potts model for structured classification problems.

3.3 The AMN Potential

The associative Markov network (AMN) potential (Taskar et al., 2004) is defined to be

$$\varphi(x_1 \ldots x_k) = 1 + \sum_{y=1}^{n} (\omega_y - 1) I(x_1 = x_2 = \ldots = x_k = y) \tag{7}$$

for parameters $\omega_y > 1$, where $I(\text{predicate})$ is defined to be 1 if the predicate is true and 0 if it is false. So, the AMN potential is constant unless all of the variables $x_1 \ldots x_k$ are assigned to the same class y, in which case it is equal to ω_y.

The AMN potential reduces to the Potts potential when $k = 2$ and $\omega_y = \omega$ for all y. So, in this case it inherits the same problems that the Potts potential has. For any k, the AMN potential has no direct effect on a label vector's likelihood unless *all* of the k argument variables agree on the label y. (It affects all vectors' likelihoods indirectly through the normalizing constant.) As k gets larger, there are comparatively fewer label vectors where all k labels agree, and so the AMN potential may not have much influence on the overall posterior distribution over label vectors unless the cell-level classifiers already were very close to agreement. And in fact, our experiments below demonstrate that factor graphs based on the AMN potential perform best when the neighborhood size k is relatively small (but larger than 2).

4. Decomposable Potentials

While k-way factors can lead to more accurate inference, they can also slow down belief propagation. For a general k-way factor, it takes time exponential in k even to look at all of the entries. So, we cannot expect to find inference algorithms for general k-way potentials that take less than exponential time.

For specific k-way potentials, though, we can hope to take advantage of special structure to design a fast inference algorithm. In particular, for many interesting potential functions, we can write down an algorithm which efficiently performs sums of the form required for message computation:

$$\sum_{x_1} \sum_{x_2} \cdots \sum_{x_{k-1}} \varphi_j^*(x_1, \ldots, x_k), \tag{8}$$

$$\varphi_j^*(x_1, \ldots, x_k) = m_1(x_1) m_2(x_2) \ldots m_k(x_{k-1}) \varphi_j(x_1, \ldots, x_k). \tag{9}$$

Here $m_i(x_i)$ is the message to factor j from variable x_i. (If we removed loops in our factor graph by collapsing groups of variables, then Equation 9 may look instead like

$$\varphi_j^*(x_1, \ldots, x_k) = m_{12}(x_1, x_2) m_{245}(x_2, x_4, x_5) \ldots m_{k-1}(x_{k-1}) \varphi_j(x_1, \ldots, x_k).$$

That is, the argument sets of the messages may overlap with one another. The derivations below apply equally well to either expression for φ_j^*.)

For example, as we will see below, we can compute Equations 8–9 quickly if φ_j is a sum of terms $\sum_l \psi_{jl}$ where each term ψ_{jl} depends only on a small subset of its arguments $x_1 \ldots x_k$. Or, we can compute Equations 8–9 quickly if φ_j is constant except at a small number of input vectors $(x_1, \ldots, x_k)$. In the first case we will say that φ_j is a sum of *low-arity* terms ψ_{jl}, and in the second case we will say that φ_j is *sparse*.

More generally, suppose that φ_j^* in Equation 8 can be written as a sum of products of low-arity functions: writing ψ_{jl} for a generic term in the sum and ξ_{jlm} for a generic factor of ψ_{jl},

$$\varphi_j^*(x_1, \ldots, x_k) = \sum_{l=1}^{L_j} \psi_{jl}(x_1, \ldots, x_k) = \sum_{l=1}^{L_j} \prod_{m=1}^{M_{jl}} \xi_{jlm}(x_{V(j,l,m)}) \tag{10}$$

where the set of indices $V(j,l,m) \subseteq \{1 \ldots k\}$ tells us which variables ξ_{jlm} depends on. Also suppose that, for each term ψ_{jl} in the sum, the sets $V(j,l,m)$ can be arranged into a junction tree. That is, suppose that we can build a cycle-free graph on M_{jl} nodes, with one node labeled with $V(j,l,m)$ for each m, which satisfies the running intersection property.

If φ_j^* satisfies the above properties, and if L_j, M_{jl}, and $|V(j,l,m)|$ are small for all j, l, and m, then we will say that φ_j^* is *decomposable*. And, we will say that φ_j is a *decomposable potential* (in the context of this message computation). Decomposable potentials include the special cases mentioned above, namely sparse potentials and potentials that are sums of low-arity terms, as well as a wide variety of other examples which we will describe in more detail below. We will see below that we can evaluate Equations 8–9 quickly for a decomposable potential by using an algorithm very similar to belief propagation.

5. Belief Propagation with Decomposable Potentials

When we are running BP or loopy BP on a factor graph with decomposable potentials, we can accelerate the computation of the belief messages that would otherwise be slow to compute. There are two types of messages that we need for BP or loopy BP, shown in Equations 1 and 2.

Messages from a variable (or a separator, which we treat as a single large variable) to a factor are fast to compute in any case: we can calculate them from Equation 1 by looping over the incoming edges at node x_i and, for each edge, looping over the n possible classes we can assign to x_i. So, we do not need to accelerate the computation of these messages.

Messages from a factor to a variable (Equation 2) are slow to compute naïvely if the factor connects to many other variables. In this section we will show that we can calculate these messages efficiently for decomposable potentials of the form shown in Equation 10. So, this section shows that we can implement BP and loopy BP efficiently for decomposable potentials. The derivation of this section works with general decomposable potentials; below, in Sections 6.1 and 6.2, we will work out the formulas for specific cases including the voting potential and the associative Markov network potential.

The algorithm that we will use to compute the messages is essentially the same as the overall belief propagation algorithm. So, when we perform inference on a factor graph with decomposable potentials, we will be running two nested copies of belief propagation: the inner copy will act on a single decomposable potential, and will compute the messages which the outer copy needs to send from that potential.

Substituting Equations 8–10 into Equation 2 and rearranging terms tells us that the desired message is

$$\begin{aligned} m_{j\to k}(x_k) &= \sum_{x_1}\sum_{x_2}\cdots\sum_{x_{k-1}} \varphi_j(x_1,\ldots,x_k)\prod_{i=1}^{k-1} m_{i\to j}(x_i) \\ &= \sum_{x_1}\sum_{x_2}\cdots\sum_{x_{k-1}} \left(\sum_{l=1}^{L_j}\prod_{m=1}^{M_{jl}} \xi_{jlm}(x_{V(j,l,m)})\right) \\ &= \sum_{l=1}^{L_j}\sum_{x_1}\sum_{x_2}\cdots\sum_{x_{k-1}}\prod_{m=1}^{M_{jl}} \xi_{jlm}(x_{V(j,l,m)}). \end{aligned} \tag{11}$$

Now let us fix l temporarily, leaving a term of the form

$$\sum_{x_1}\sum_{x_2}\cdots\sum_{x_{k-1}}\prod_{m=1}^{M_{jl}} \xi_{jlm}(x_{V(j,l,m)}). \tag{12}$$

We have assumed that the sets $V(j,l,m)$ for $m \in \{1,\ldots,M_{jl}\}$ can be arranged into a junction tree. Pick a node of this junction tree whose label contains x_k, and call this node the root. Refer to each node by its index m, and write $\mathrm{par}(m)$ for the parent of node m.

Now pick an arbitrary leaf of the junction tree. Without loss of generality, say that this leaf has index $m = 1$. We can partition the variables in the set $V(j,l,1)$ into two subsets,

$$C(1) = V(j,l,1) \cap V(j,l,\mathrm{par}(1)) \quad \text{and} \quad D(1) = V(j,l,1) \setminus V(j,l,\mathrm{par}(1)).$$

$C(1)$ contains the variables that node 1 has in Common with its parent, while $D(1)$ contains the variables from node 1 that are Distinct from its parent's variables. The variables in $D(1)$ can only appear in $V(j,l,1)$: if any of them appeared in $V(j,l,m)$ for $m \neq 1$ it would violate the running intersection property, since any path from 1 to m has to pass through $\mathrm{par}(1)$.

Without loss of generality, suppose that the variables in $D(1)$ are numbered consecutively from 1, say $D(1) = \{1,2\}$. That means that the factor ξ_{jl1} depends on x_1 and x_2, but no other factor ξ_{jlm} for $m \neq 1$ depends on x_1 or x_2. So, by the distributive law, we can rearrange Equation 12 as follows:

$$\sum_{x_3} \cdots \sum_{x_{k-1}} \prod_{m=2}^{M_{jl}} \xi_{jlm}(x_{V(j,l,m)}) \cdot \left(\sum_{x_1} \sum_{x_2} \xi_{jl1}(x_{V(j,l,1)}) \right) \tag{13}$$

To get Equation 13 we have moved the x_1 and x_2 summations inward as far as possible.

We can think of the expression in parentheses in Equation 13 as a message which travels from node 1 to node $\mathrm{par}(1)$ of the junction tree: it depends only on the variables in the set $C(1)$, and it summarizes everything that node $\mathrm{par}(1)$ needs to know about node 1 in order to compute the term of Equation 12. We will write $\mu_{1 \to \mathrm{par}(1)}$ for this message, that is,

$$\mu_{1 \to \mathrm{par}(1)}(x_{C(1)}) = \sum_{x_1} \sum_{x_2} \xi_{jl1}(x_{V(j,l,1)}). \tag{14}$$

We can continue computing messages in this fashion from leaf nodes of the junction tree to their parents. After several steps our expression will look something like this:

$$\sum_{x_7} \cdots \sum_{x_{k-1}} \prod_{m=4}^{M_{jl}} \xi_{jlm}(x_{V(j,l,m)}) \prod_{m=1}^{3} \mu_{m \to \mathrm{par}(m)}(x_{C(m)}). \tag{15}$$

Here we have eliminated the variables x_1 through x_6 and computed the messages from nodes $m = 1,2,3$ to their parents. At this point suppose that node $m = 4$ is an internal node of the junction tree, and that it has as its only child node $m = 1$. Because we have already computed the message from node 1 to node 4, we can now compute the message from 4 to $\mathrm{par}(4)$. Suppose that $D(4) = \{7\}$; then we can rearrange Equation 15 as follows.

$$\begin{aligned} &\sum_{x_8} \cdots \sum_{x_{k-1}} \prod_{m=5}^{M_{jl}} \xi_{jlm}(x_{V(j,l,m)}) \times \\ &\qquad \prod_{m=2}^{3} \mu_{m \to \mathrm{par}(m)}(x_{C(m)}) \left(\sum_{x_7} \mu_{1 \to 4}(x_{C(1)}) \xi_{jl4}(x_{V(j,l,4)}) \right). \end{aligned} \tag{16}$$

Here we have used the distributive law to move the x_7 summation inward as far as possible. As above, none of the functions ξ_{jlm} for $m > 4$ can depend on x_7: if they did, then by the running

intersection property, 7 would have to be an element of $V(\text{par}(4))$ and couldn't be in $D(4)$. The message $\mu_{1\to 4}$ could depend on x_7, since 7 could be in $C(1)$. So, to be safe, we have left $\mu_{1\to 4}$ inside the x_7 summation. On the other hand, the messages $\mu_{2\to\text{par}(2)}$ and $\mu_{3\to\text{par}(3)}$ cannot depend on x_7: if one of them did, 7 would have to be in $V(j,l,\text{par}(2))$ or $V(j,l,\text{par}(3))$, which would again violate the running intersection property.

In a natural generalization of Equation 14, we will write $\mu_{4\to\text{par}(4)}(x_{C(4)})$ for the expression in parentheses in Equation 16. It should be clear at this point that we can compute a message from each node in the junction tree to its parent, so long as we work from the leaves upward; the message from a node m to its parent $\text{par}(m)$ will depend on the messages from m's children to m. Once we process all of the children of the root r, we will be left with an expression that contains summations only over variables in $V(j,l,r)$. (In fact, it will contain summations over exactly the variables in $V(j,l,r)\setminus\{k\}$.) We can perform these summations to find the desired term (Equation 12), which is a function only of x_k. We can then repeat the process for each l to get all of the terms in Equation 11.

The above algorithm computes the message from a factor φ_j to a single neighboring variable x_k. If we want the messages from φ_j to all of its variables we can run the above algorithm multiple times; however, the multiple runs will redundantly recompute many of the messages $\mu_{s\to t}$. Instead, as is usual for the belief propagation algorithm, we can combine all of the runs into a single computation which passes one message in each direction over each edge of the junction tree.

If we have merged groups of variables in constructing our outer junction tree, then there are two modifications needed for the above analysis. First, if the argument sets of the incoming messages at a given factor node overlap, we may need to build different inner junction trees to compute different outgoing messages. So, we may not be able to share computation as described in the previous paragraph. This loss of sharing may increase our runtime by a small factor. Second, and more importantly, if the desired outer message depends on several original variables, then there may be no node of our inner junction tree that contains all the needed variables. In this case we may condition on the possible values of some of the needed variables, and use several runs of inner message passing, each of which computes a slice of the desired outer message. In general, there may be several ways to decompose a potential, and several possible choices of which variables to condition on for a given decomposition. Each of these setups may lead to a different runtime for message computation. See Section 6.3 below and Kjærulff (1998) for more details.

6. Instances of Decomposable Potentials

Decomposable potentials are common and useful. In this section, we discuss the details and derivations of message passing with the voting potential, the associative Markov network potential, and the nested junction tree.

6.1 Decomposing the Voting Potential

The general BP-style algorithm of Section 5 is more complicated than we need when we are computing messages for the voting potential: since Equation 6 is a sum of low-arity functions rather than a sum of products of low-arity functions, the computation for each term in the sum is particularly simple. So, in this section we will derive efficient expressions for the necessary messages.

There are two types of messages we need to think about: those from a factor φ_j to the node x_k that φ_j is centered on, and those from a factor φ_j to some non-centered variable node x_i with $i\neq k$. To simplify notation, assume that $V(j)=\{1,\ldots,k\}$; also assume that we have normalized the

messages $m_{i\to j}(x_i)$ so that $\sum_{x_i} m_{i\to j}(x_i) = 1$. With these assumptions, the message from a factor φ_j to its center variable node x_k can be computed as follows:

$$\begin{aligned}
&(\lambda + |N(j)|)\, m_{j\to k}(x_k) \\
&= \sum_{x_1}\sum_{x_2}\cdots\sum_{x_{k-1}} \left(\lambda/n + \sum_{i=1}^{k-1} I(x_i, x_k)\right) \prod_{i'=1}^{k-1} m_{i'\to j}(x_{i'}) \\
&= \frac{\lambda}{n}\sum_{x_1}\sum_{x_2}\cdots\sum_{x_{k-1}}\prod_{i'=1}^{k-1} m_{i'\to j}(x_{i'}) + \\
&\quad \sum_{x_1}\sum_{x_2}\cdots\sum_{x_{k-1}}\sum_{i=1}^{k-1} I(x_i, x_k)\prod_{i'=1}^{k-1} m_{i'\to j}(x_{i'}) \\
&= \frac{\lambda}{n} + \sum_{i=1}^{k-1}\sum_{x_1}\sum_{x_2}\cdots\sum_{x_{k-1}} I(x_i, x_k)\prod_{i'=1}^{k-1} m_{i'\to j}(x_{i'}) \\
&= \frac{\lambda}{n} + \sum_{i=1}^{k-1}\sum_{x_1}\sum_{x_2}\cdots\sum_{x_{k-1}} I(x_i, x_k) m_{i\to j}(x_i) \prod_{i'=1, i'\neq i}^{k-1} m_{i'\to j}(x_{i'}) \\
&= \frac{\lambda}{n} + \sum_{i=1}^{k-1}\sum_{x_i} I(x_i, x_k) m_{i\to j}(x_i) \\
&= \frac{\lambda}{n} + \sum_{i=1}^{k-1} m_{i\to j}(x_k).
\end{aligned}$$

The first equation above is the definition of the desired message. The second equation distributes multiplication over addition. The third equation uses the fact that all terms in the product $\prod_{i'=1}^{k-1} m_{i'\to j}(x_{i'})$ are independent, along with our assumption $\sum_{x_{i'}} m_{i'\to j}(x_{i'}) = 1$, to compute the first summation. The fourth equation factors $m_{i\to j}$ out of the product. The fifth equation uses again the facts that all terms in the product are independent and $\sum_{x_{i'}} m_{i'\to j}(x_{i'}) = 1$. The last line uses the fact that $I(x_i, x_k)$ is nonzero iff $x_i = x_k$.

The message from a factor j to a non-centered variable can be computed similarly. Under the same assumptions as above, we will calculate the message $m_{j\to 1}$:

$$\begin{aligned}
&(\lambda + |N(j)|)\, m_{j\to 1}(x_1) \\
&= \sum_{x_2}\cdots\sum_{x_k} \left(\lambda/n + \sum_{i=1}^{k-1} I(x_i, x_k)\right) \prod_{i'=2}^{k} m_{i'\to j}(x_{i'}) \\
&= \frac{\lambda}{n} + \sum_{x_2}\cdots\sum_{x_k}\sum_{i=1}^{k-1} I(x_i, x_k)\prod_{i'=2}^{k} m_{i'\to j}(x_{i'}) \\
&= \frac{\lambda}{n} + \sum_{x_2}\cdots\sum_{x_k} I(x_1, x_k)\prod_{i'=2}^{k} m_{i'\to j}(x_{i'}) + \sum_{x_2}\cdots\sum_{x_k}\sum_{i=2}^{k-1} I(x_i, x_k)\prod_{i'=2}^{k} m_{i'\to j}(x_{i'}) \\
&= \frac{\lambda}{n} + \sum_{x_k} I(x_1, x_k) m_{k\to j}(x_k) + \sum_{x_2}\cdots\sum_{x_k}\sum_{i=2}^{k-1} I(x_i, x_k)\prod_{i'=2}^{k} m_{i'\to j}(x_{i'}) \\
&= \frac{\lambda}{n} + m_{k\to j}(x_1) + \sum_{x_2}\cdots\sum_{x_k}\sum_{i=2}^{k-1} I(x_i, x_k)\prod_{i'=2}^{k} m_{i'\to j}(x_{i'})
\end{aligned}$$

$$
\begin{aligned}
&= \frac{\lambda}{n} + m_{k\to j}(x_1) + \sum_{i=2}^{k-1} \sum_{x_2} \ldots \sum_{x_k} I(x_i, x_k) m_{i\to j}(x_i) m_{k\to j}(x_k) \prod_{i'=2, i'\neq i}^{k-1} m_{i'\to j}(x_{i'}) \\
&= \frac{\lambda}{n} + m_{k\to j}(x_1) + \sum_{i=2}^{k-1} \sum_{x_i} \sum_{x_k} I(x_i, x_k) m_{i\to j}(x_i) m_{k\to j}(x_k) \\
&= \frac{\lambda}{n} + m_{k\to j}(x_1) + \sum_{i=2}^{k-1} \sum_{x_k} m_{i\to j}(x_k) m_{k\to j}(x_k).
\end{aligned}
$$

The first equality above is the definition of the desired message. The second pulls the term λ/n out of the sums, using the facts that the terms in the product $\prod_{i'=2}^{k} m_{i'\to j}(x_{i'})$ are independent and $\sum_{x_{i'}} m_{i'\to j}(x_{i'}) = 1$. The third equality splits the sum over i into two pieces, one for $i = 1$ and the other for $i \geq 2$. The fourth equality uses the independence of the terms in the left-hand product to simplify away the summations over x_2 through x_{k-1}. The fifth uses the fact that $I(x_1, x_k) = 0$ when $x_1 \neq x_k$. The sixth equality pulls the terms $m_{i\to j}$ and $m_{k\to j}$ out of the remaining product. The seventh uses the independence of terms in the product to simplify away the summations over variables other than x_i and x_k. And the last equality uses the fact that $I(x_i, x_k) = 0$ when $x_i \neq x_k$.

Despite the fact that the variables $x_1, \ldots, x_k$ have exponentially many possible assignments, the above derivations show that we can compute the messages $m_{j\to k}$ and $m_{j\to 1}$ exactly and almost instantaneously. The message $m_{j\to k}$ is particularly easy to interpret: it is the (Laplace smoothed) average of the messages from $x_1, \ldots, x_{k-1}$.

6.2 Decomposing the AMN Potential

It is even easier to derive an efficient expression for the messages from an AMN potential than it was for the messages from a voting potential. As before, let us assume that $V(j) = \{1, \ldots, k\}$, that we desire the message $m_{j\to k}(x_k)$, and that we have normalized the messages $m_{i\to j}(x_i)$ to sum to 1 over x_i for each $i = 1, \ldots, k-1$. Since the AMN potential is symmetric in its arguments, there is only one type of message to calculate.

We can write Equation 7 in the form of Equation 10 by noting that

$$I(x_1 = x_2 = \ldots = x_k = y) = I(x_1 = y) I(x_2 = y) \ldots I(x_k = y)$$

and that each function $I(x_i = y)$ depends on only one variable x_i. Given this representation, the desired message is:

$$
\begin{aligned}
m_{j\to k}(x_k) &= \sum_{x_1} \sum_{x_2} \ldots \sum_{x_{k-1}} \left(1 + \sum_y (\omega_y - 1) \prod_{i=1}^{k} I(x_i, y) \right) \prod_{i'=1}^{k-1} m_{i'\to j}(x_{i'}) \\
&= 1 + \sum_{x_1} \sum_{x_2} \ldots \sum_{x_{k-1}} \sum_y (\omega_y - 1) \prod_{i=1}^{k} I(x_i, y) \prod_{i'=1}^{k-1} m_{i'\to j}(x_{i'}) \\
&= 1 + \sum_y (\omega_y - 1) I(x_k, y) \sum_{x_1} \sum_{x_2} \ldots \sum_{x_{k-1}} \prod_{i=1}^{k-1} I(x_i, y) m_{i\to j}(x_i) \\
&= 1 + \sum_y (\omega_y - 1) I(x_k, y) \prod_{i=1}^{k-1} m_{i\to j}(y) \\
&= 1 + (\omega_{x_k} - 1) \prod_{i=1}^{k-1} m_{i\to j}(x_k).
\end{aligned}
$$

The first equality above is the definition of the desired message. The second pulls the constant 1 outside of the sums and uses the independence of terms in the product. The third rearranges the order of the sums and products. The fourth uses the fact that the product is zero unless $x_i = y$ for all $i \in \{1,\ldots,k-1\}$. The fifth uses the fact that $I(x_k,y)$ is 0 unless $x_k = y$.

6.3 The Nested Junction Tree

The nested junction tree method (Kjærulff, 1998) can speed up message propagation in the junction trees that arise when we remove loops from a factor graph. In particular, it helps compute messages from the large factors that arise when we merge groups of variables. It works by noticing that each of these messages is computed from the product of many smaller factors, which can sometimes be arranged into a nontrivial "inner" junction tree.

Unlike the previous two examples (the voting and AMN potentials), the nested junction tree method does not attempt to look inside the factors of the original factor graph. Instead, it keeps track of the variable merges during junction tree construction, and sometimes is able to "undo" some of these merges temporarily to provide computational savings.

For example, the factor graph of Fig. 4 can be collapsed into a junction tree by merging x_2 and x_3 as shown. The time and space costs of belief propagation on this junction tree are dominated by its largest potential, φ_{2345}. Consider the message from φ_{2345} to x_{23}:

$$m_{2345\to 23}(x_2,x_3) = \sum_{x_4}\sum_{x_5} \varphi_{2345}(x_2,x_3,x_4,x_5) m_{4\to 2345}(x_4) m_{5\to 2345}(x_5).$$

Standard belief propagation will first compute

$$\begin{aligned} \varphi^*_{2345}(x_2,x_3,x_4,x_5) &= \varphi_{2345}(x_2,x_3,x_4,x_5) m_{4\to 2345}(x_4) m_{5\to 2345}(x_5) \\ &= \varphi_{245}(x_2,x_4,x_5)\varphi_{345}(x_3,x_4,x_5) m_{4\to 2345}(x_4) m_{5\to 2345}(x_5) \end{aligned}$$

for all settings of (x_2,x_3,x_4,x_5), and then marginalize φ^*_{2345} to get $m_{2345\to 23}$.

If there are (for example) ten possible settings of each original variable, the space required for computing $m_{2345\to 23}$ with standard belief propagation is 10100 locations: 10^4 for storing φ^*_{2345}, and 10^2 for storing $m_{2345\to 23}$. And, the time cost is 39900 flops: for each of the 10^4 settings of (x_2,x_3,x_4,x_5) we must multiply together one element each from the tables φ_{245}, φ_{345}, $m_{4\to 2345}$, and $m_{5\to 2345}$, at a cost of 3 flops per iteration. Then, for each of the 10^2 elements of $m_{2345\to 23}$, we must sum 10^2 elements of φ^*_{2345} (using 10^2-1 flops), leading to a total cost of $3\times 10000 + 99\times 100$.

However, if we examine the message computations in more detail, it turns out that we can take advantage of the structure of φ^*_{2345} to save time and space. Since φ^*_{2345} was formed by multiplying together four smaller tables, and since the argument sets of these smaller tables form a junction tree, φ^*_{2345} is decomposable. (The inner junction tree is shown at the far right of Fig. 4.) Using this fact, we can find $m_{2345\to 23}$ by passing messages through the inner junction tree several times. In more detail, suppose we pick the factor φ_{345} as the root of the inner junction tree. This factor contains one of the message variables (x_3) but not the other one (x_2). So, we must condition on each value of x_2 in turn. For each value of x_2, we first compute the intermediate message

$$\varphi^{x_2}(x_4,x_5) = \varphi_{245}(x_2,x_4,x_5) m_{4\to 2345}(x_4) m_{5\to 2345}(x_5).$$

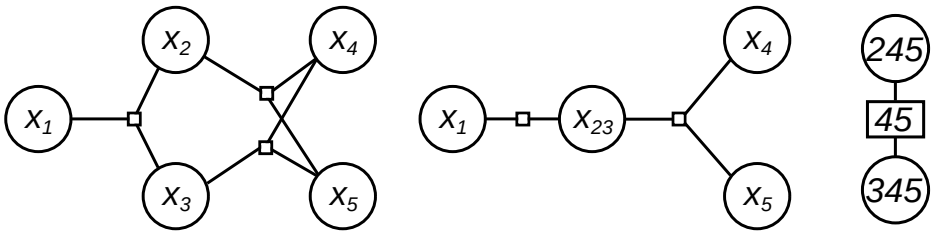


Figure 4: A factor graph for which the nested junction tree method can speed up belief propagation. Left: original factor graph. Middle: factor graph after merging x_2 and x_3. Right: inner junction tree for computing the message $m_{2345\to 23}$.

This message is associated with the separator $\{x_4, x_5\}$ in the inner junction tree. We can then incorporate this inner message into the factor $\{x_3, x_4, x_5\}$, and marginalize to get a slice of our desired (outer) message:

$$m_{2345\to 23}(x_2, x_3) = \sum_{x_4}\sum_{x_5} \varphi_{345}(x_3, x_4, x_5)\varphi^{x_2}(x_4, x_5).$$

Each pass through the inner junction tree keeps x_2 fixed, computing a slice $m_{2345\to 23}(x_2, \cdot)$ of the outer message. (By $m_{2345\to 23}(x_2, \cdot)$, we mean a table of the values of $m_{2345\to 23}$ for a fixed value of the first argument and all possible values of the second argument.)

The nested belief propagation algorithm for computing $m_{2345\to 23}$ saves us both space and time. For time, the cost to compute an element of one of the φ^{x_2} tables is 2 multiplications; there are 10^2 elements of each table, and 10 tables, so the total cost of this step is $2 \times 10 \times 100 = 2000$ flops. To compute $\varphi_{345}(x_3, x_4, x_5)\varphi^{x_2}(x_4, x_5)$ for fixed x_2 takes 1000 flops (one per table element), and to marginalize out (x_4, x_5) takes 99 flops for each of the 10 resulting elements, for a total of $1000 + 10 \times 99 = 1990$ per slice. Since there are 10 slices, we need 19900 flops for all of them. The grand total is therefore $19900 + 2000 = 21900$, a savings of approximately 45% compared to standard belief propagation.

The space savings are even greater: we can reuse a single array of size 10^2 for all of the φ^{x_2} tables, and a single array of size 10^3 for all of the products $\varphi_{345}(x_3, x_4, x_5)\varphi^{x_2}(x_4, x_5)$. Adding in the cost of storing the final message, we have a space cost of $100 + 1000 + 100 = 1200$ locations, a savings of about 88%.

An important limitation of the nested junction tree method is the following lemma, which provides a lower bound on inference time based on the number of variables in the largest node label (or clique):

Lemma 1 *In a junction tree T with binary variables, let C be the largest clique, and say that we wish to compute the posterior marginal for some variable x_q. So long as C is minimal, the nested junction tree method cannot reduce the time for this inference task to less than $\Omega(2^{|C|})$. (The time for non-binary variables is at least as large.)*

Proof Each edge (i, j) in clique C arises either because there is a potential that directly links variables x_i and x_j (this case includes so-called "moral edges"), or because we triangulated a chordless cycle that contains x_i and x_j (in which case some other clique sends C a message that links x_i with x_j). The nested junction tree method achieves its speedup by avoiding consideration of some of

these edges when computing outgoing messages from C: for a message M from C to some other clique D, we can ignore the corresponding incoming message N from D to C. Without N, we may be able to ignore as many as $\binom{|M|}{2}$ of the edges of C while calculating M, since each edge that connects two variables in M may be supported only by N.

Because we can ignore some of the edges of C, we can avoid building a single large table of size $2^{|C|}$, and instead run a nested copy of the message passing algorithm to find M. This inner message passing algorithm works on an inner junction tree T' built from the remaining edges of C. To compute M, we pick some clique C' of T' as root; then, for each possible setting of the variables in $M \setminus C'$, we pass messages from leaves to root in T' to find a slice of M. This slice corresponds to the fixed setting of $M \setminus C'$, and covers all possible settings of the variables in $M \cap C'$.

To calculate the total cost of these inner runs of message passing, we need to look at the structure of T'. In particular, we need to figure out the size of the largest clique of T'. For this purpose, we can divide the variables of T' into three sets: those in $M \setminus C'$ (which we hold fixed during each inner iteration), those in $M \cap C'$ (which form the resulting slice of M), and those in $C \setminus M$ (the rest of C). The variables in $M \cap C'$ are fully connected to one another, since they are all members of C'. And, the variables in $C \setminus M$ are fully connected to one another, since none of them are covered by the incoming message N. But, these two sets of variables are also fully connected to each another: no edge between $C \setminus M$ and $M \cap C'$ can be covered by N, since each such edge has one vertex in M and one outside of M. So, $C \setminus M$ and $M \cap C'$ together form a clique of T'.

Recapping, we have $2^{|M \setminus C'|}$ inner runs of message passing, each of which works with a junction tree containing a clique of size $|C \setminus M| + |M \cap C'|$. We will now prove by induction that inference takes time at least $k2^{|C|}$, where k is an implementation-dependent constant.

For the inductive step, our runtime for calculating M is at least

$$2^{|M \setminus C'|} \times k2^{|C \setminus M| + |M \cap C'|}$$

for $2^{|M \setminus C'|}$ runs of message passing, each of which costs $k2^{|C \setminus M| + |M \cap C'|}$ by the inductive hypothesis. Since $|M \setminus C'| + |C \setminus M| + |M \cap C'| = |C|$, our runtime is therefore at least $k2^{|C|}$, as claimed.

There are several possible base cases. The most obvious is when $|C| = 1$; in this case we can choose $k > 0$ so that the runtime is at least $2k$. The induction can also bottom out if the nested junction tree method becomes inapplicable at any step. There are two ways it can become inapplicable: first, if $M \setminus C' = \emptyset$, then the argument above means that the nested junction tree has only a single clique of size $|C|$, and the nested junction tree method therefore offers no speedup. In this case we need to build a table of size $2^{|C|}$ for inference. So, we can again choose $k > 0$ so that our runtime is at least $k2^{|C|}$.

Second, the nested junction tree method is inapplicable if clique C has no outgoing messages. C has no outgoing messages if and only if we select it as the root for message passing. In this case, C gets incoming messages from all of its neighbors, and the nested junction tree method again offers no speedup: to perform any inference task on C's fully-connected graph, we must again build a table of size $2^{|C|}$ and take time at least $k2^{|C|}$ for some $k > 0$.

So, by choosing $k > 0$ small enough to satisfy all of the above base cases, we have completed the induction. The only remaining detail is the question of non-binary variables. But, it should be obvious from the proof above that any non-binary variables can only increase the cost of inference. ■

Lemma 1 means that we cannot expect the nested junction tree method on its own to make it practical

to work with trees with very large cliques. This conclusion is borne out by the experimental results of Kjærulff (1998), Tables 1 and 2: the time savings shown there are never greater than about 60%. In contrast, the examples of the previous two sections show that decomposable potentials in general can lead to far greater savings: in these examples, the message calculation time goes from exponential to polynomial in the size of the clique, and Fig. 10 below shows that this change can easily result in speedups of multiple orders of magnitude.

7. Prior Updating

By calculating messages as described in Section 6.1, we can run loopy belief propagation on a factor graph that includes voting potentials. However, we might expect the messages from a factor φ_j to a non-centered variable x_i (where $i \neq c_j$) to be fairly weak: the overall vote of all of x_{c_j}'s neighbors will not be influenced very much by x_i's single vote, so there will not be a strong penalty if x_i votes the wrong way.

This observation suggests an even simpler algorithm for inference: we can run loopy BP but ignore all of the messages from factors to non-centered variables. (Ignoring a message means considering it to be uniform.) We will call this algorithm Prior Updating, or PU, since it works by using the current classifications of a node's neighbors to update the prior for the node's own classification. Our group proposed a version of Prior Updating for inference on graphs (Chen and Murphy, 2006) before the work described in the current paper, which explores its relationship to potential functions in loopy belief propagation.

We can expect that PU will be noticeably faster than loopy BP, since there will usually be many more non-centered variables than there are centered ones in each factor. We might also hope that PU could be more accurate than loopy BP, since it is less prone to double-count evidence: we have broken any loops which would allow a message that x_{c_j} sends to factor φ_j to return back and influence the classification of x_{c_j}. (As it turns out, we will see below that in our experiments PU is often slightly worse and sometimes slightly better than loopy BP on factor graphs with voting potentials.) We will examine the speed and classification performance of loopy BP, PU, and other inference methods in Section 9. PU can be seen as an approximation of LBP on factor graphs with voting potentials, and like LBP, is not guaranteed to converge. However, in our experiments, we never observed problems with convergence for PU or for LBP with the voting potential.

8. Experimental Materials and Methods

2D HeLa Image Set We applied our methods to the problem of classifying subcellular patterns in an image of many cells, a problem that we have formalized previously (Chen and Murphy, 2006). The starting point is a set of fluorescence microscope images of HeLa cells created by introducing antibodies and molecular probes against proteins in major subcellular organelles (Boland and Murphy, 2001). The data set contains 862 single-cell images from ten classes, with each class having between 73 and 98 images. The true class of each image is known with certainty since the fluorescent probe present in each slide is known.

Classifying protein subcellular patterns is important since it allows us to identify where a protein is located in cell organelles, which is required for it to carry out its specific function (Boland and Murphy, 2001). This knowledge is critical to understanding how that protein works in a cell, and to describing cell behaviors under different conditions.

In the past, the most common way to determine protein location patterns has been by human interpretation of fluorescence microscope images. In recent years, however, automated systems have been developed for consistent and objective interpretation of such images; the current paper's techniques are designed to help the accuracy of these automated systems (Chen et al., 2006c; Glory and Murphy, 2007). Automated systems, when available, can be preferable to human image annotation, since they can help avoid human biases and errors. They also make it possible to analyze images from high-throughput microscopy, which would otherwise be too numerous for humans to handle.

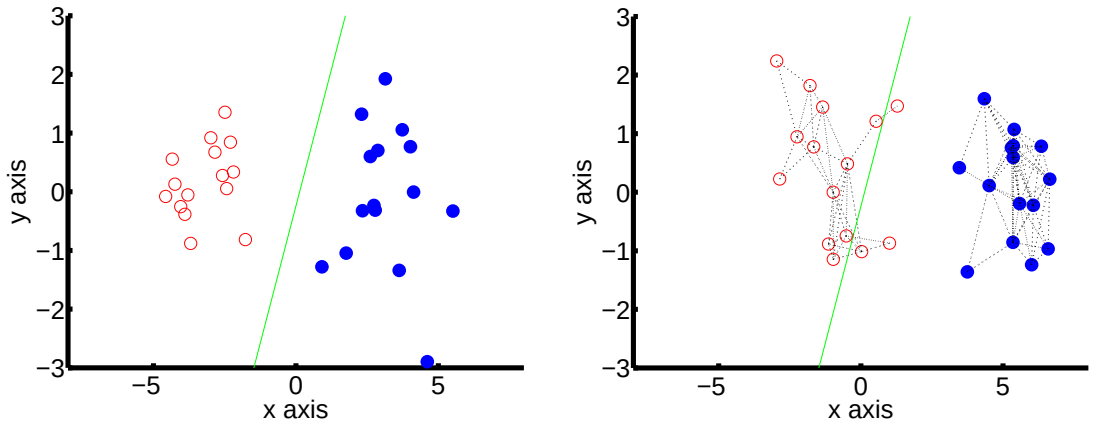
Subcellular Location Features Several sets of informative features have been developed to describe protein subcellular patterns (Boland and Murphy, 2001; Chen et al., 2006c; Glory and Murphy, 2007). These features, termed Subcellular Location Features (SLFs), are of several types, including Zernike moment features, Haralick texture features, morphological features and wavelet features. The details for different versions of SLFs are reviewed elsewhere (Huang and Murphy, 2004). The best single-cell classification results obtained to date with a single feature set for the 2D HeLa data set were with feature set SLF16 (Huang and Murphy, 2004), which we have therefore used in the work described here. In this feature set, each cell is represented by a feature vector f of length $d = 47$.

Support Vector Machine To determine the evidence for each individual cell we used Support Vector Machine classifiers (Cortes and Vapnik, 1995), as implemented in the LIBSVM library version 2.82 (Chang and Lin, 2001). To handle problems with $k > 2$ classes, we learned $k(k-1)/2$ binary SVM classifiers, one for each pair of classes. We then derived probability estimates by applying sigmoid functions to the decision values of these SVMs, in an improved implementation (Lin et al., 2003) of the Platt Scaling method (Platt, 2000).

Simulating Multi-Cell Images We are interested in the simultaneous classification of all of the cells in a multi-cell microscope image. Unfortunately, it is difficult to collect multi-cell images for which we know the ground truth classification of each cell: if we prepare a slide from a mixture of two or more types of cells, we do not have direct control over which type appears where. For this reason, we simulated the multi-cell problem by creating synthetic multi-cell images using multiple real single-cell HeLa images.[1] To generate a structured classification problem, we selected two of the ten classes at random; then we selected N_1 images from the first class and N_2 from the second, with $N_1 + N_2 = 12$. We then treated these images as if they were the output of a segmentation algorithm that was run on a multi-cell image.

Constructing the Similarity Graph As an intermediate step in the construction of our factor graph, we built a *similarity graph*: the nodes of this graph correspond to cells, and an edge between two cells means that we believe that they are likely to share the same label. The simplest approach to building the similarity graph is to include all possible edges. With a fully-connected similarity graph, all cells in the test image are considered equally similar to one another; such a graph expresses a prior belief that images containing a few groups of same-class cells are more likely than images containing cells of many different classes. Our experiments below show that even this modest amount of prior information can improve the accuracy of our classifier compared to the no-edges (independent classification) case; for example, in Figs. 7–9, the performance of LBVP is higher

1. We are in the process of analyzing true multi-cell images with known ground truth, which we collect by tagging one of the cell types with a fluorescent marker at a wavelength different from the one used to label the target protein.



(a) Training Stage

(b) Testing Stage

Figure 5: Classification of multiple examples using proximity in feature space. (a) At the training stage, a linear classifier separates two classes in a 2D feature space. (b) At the testing stage, we have added feature bias, causing 3 examples of one class to be misclassified. But, these examples can be classified correctly by constructing a similarity graph and running belief propagation on the corresponding factor graph.

when 100% of edges are present than it is when 0% of edges are present. But of course, if more-specific information is available, we will get better performance by using it to construct a more informative similarity graph with an intermediate number of edges.

One possible source of additional information is physical proximity between cells. Physical proximity is informative if we believe that nearby cells are likely to share an ancestor. However, we wanted to avoid having to simulate cell positions in our synthetic images, so in the current work we did not include edges based on physical proximity. In previous work (Chen and Murphy, 2006), we did evaluate graphs built using physical proximity, and they improved classification accuracy when applicable. So long as edges tend to connect cells that share the same class, the exact source of edges does not matter for our inference algorithms; so, the experimental results below should apply equally well to graphs that contain edges from physical proximity.

Instead, for our experiments below, we built the similarity graph according to feature-space proximity: we added edges between cells whose feature vectors were close to one another according to z-scored Euclidean distance. Using feature-space proximity in this way makes sense because minor experimental variations can perturb the features of a whole group of test cells in similar ways.

In more detail, minor differences in how cells were prepared (such as variations in plating time, exposure time, concentration of reagents, ambient temperature, etc.) can cause a noticeable bias in the computed features of a test group of cells compared to our training set. We do not generally have enough cells of enough different classes in a test group to estimate this bias accurately before classifying the cells. However, if there is a margin in feature space between classes, then building a similarity graph based on feature-space proximity is likely to connect each cell mostly to other cells

of the same class. So, as long as the feature bias is not so large that it causes most cells of a given class to be misclassified, we can hope to "rescue" cells that have been moved just across the class boundary, since they will be connected to other cells of the same class that are correctly classified. Fig. 5 justifies this intuition with a simple synthetic example.

In addition to the above reasons, feature-space proximity among test images could be informative if the training set wasn't large enough or the classifier wasn't flexible enough to learn the location classes well. In our experiments, we believe that the influence of this last effect is minimal.

To produce a variety of graphs with different edge densities, we introduced a parameter d_{cutoff}: we connected two nodes whenever their z-scored Euclidean distance in feature space was less than d_{cutoff}.[2] Large values of d_{cutoff} correspond to graphs with many edges, while small values correspond to graphs with few edges. We varied d_{cutoff} to produce graphs with edge densities ranging from 0% to 100% of the possible edges.

Constructing the Factor Graphs From each similarity graph we built several different factor graphs using different kinds of potentials. Each different factor graph corresponds to a different way to turn our qualitative similarity judgements into a precise probability distribution over label vectors.

The simplest factor graph was the Potts model. In the Potts model, we used one Potts potential for each edge in the similarity graph; each potential had parameter $\omega = 1.7$. The next type of factor graph was an associative Markov network. In this model we used one AMN potential for each node; this potential covered the node and all of its similarity-graph neighbors, and had $\omega_y = 2.9$ for all y. The last type of factor graph used the voting potential. In this graph there was one voting potential centered on each node; this potential covered the node and all of its similarity-graph neighbors, and had parameter $\lambda = 1.7$. For all three types of potential, we determined the above parameter values ahead of time by a coarse search.[3]

Synthetic Graphs Since the size of the 2D HeLa data set is limited, we performed additional experiments on automatically-generated inference problems. These experiments investigated the sensitivity of our method to the accuracy of the base classifier. We generated a synthetic graph by picking two of the ten classes at random and two numbers N_1 and N_2. We generated N_1 cells of the first class and N_2 cells of the second. For each cell we selected feature vectors of length $d = 2$ from a standard normal distribution; for one of the groups of cells we then displaced the feature vectors by a distance s. We chose $s = 3$ as a value which yielded a reasonable degree of overlap between classes. Finally, as above, we connected pairs of nodes whose feature-space distances were less than d_{cutoff}.

Synthetic Evidence To generate the evidence for a node in one of our synthetic graphs, we picked random scores for each class. The score for the true class was generated from a normal distribution

2. A reviewer of a previous version of this paper suggested that all edges should be present in the graph, and that we should adjust the weight of each edge based on the distance between the nodes. This is a reasonable suggestion; however, some of the potential functions we are evaluating do not have an obvious way to take edge weights into account. So, for the sake of an easier comparison, in this paper we work only with 0-1 weights.
3. As we examined the parameter space of the AMN potential, we discovered that results appear to be very sensitive to the strength parameter ω and the edge density of the graph. So, while we fixed a single compromise parameter for the AMN potentials in our experiments (so that our AMN results are comparable to our results for other potentials), we strongly recommend parameter learning for practical use of the AMN potential, for example, by the method described in (Taskar et al., 2004).

with mean $\mu > 0$ and unit variance, while the scores for other classes were generated from normal distributions with mean 0 and unit variance. Each score was then transformed by a sigmoid to produce the evidence for its corresponding class. Large values of μ result in a highly-accurate simulated base classifier, while small values yield a classifier that performs only a little better than chance. We show results for a range of values of μ that result in base accuracies from 50% to 90%.

9. Experimental Results

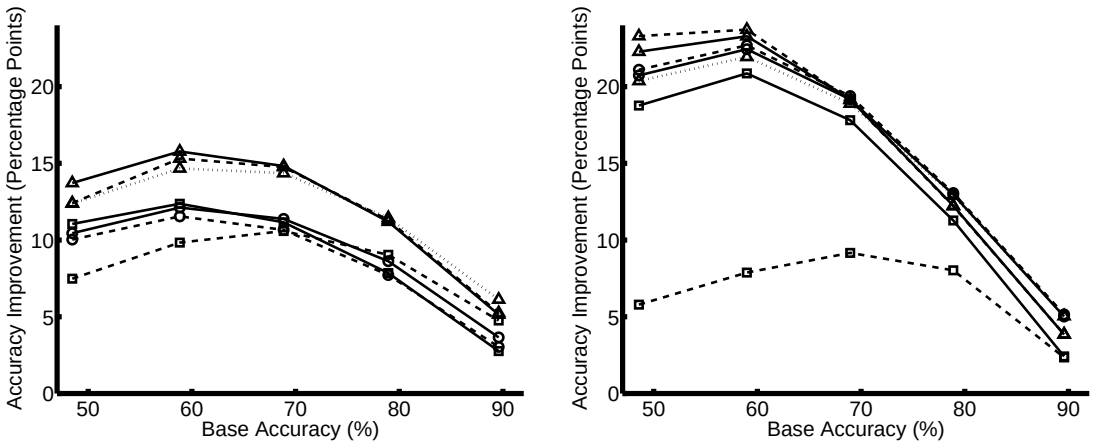
We conducted experiments to determine the effect of various potential functions and inference algorithms on overall classification accuracy in structured classification problems. In particular, we designed our experiments to compare the two-way Potts potential with the k-way AMN and voting potentials, and to compare our approximate inference algorithms to their exact counterparts. Our results support the following conclusions:

- We can achieve better classification accuracy by moving from the Potts model, with its two-way potentials, to models that contain k-way potentials for $k > 2$.
- Of the k-way potentials that we tested, the voting potential is the best for a range of problem types.
- For small networks where exact inference is feasible, our approximate inference algorithms yield results similar to exact inference at a fraction of the computational cost.
- For larger networks where exact inference becomes intractable, our approximate inference algorithms are still feasible, and structured classification with approximate inference lets us take advantage of the similarity graph to improve classification accuracy compared to the base (unstructured) classifier.

9.1 Synthetic Graphs

Our first experiment used small, synthetic graphs to compare the Potts, AMN, and voting potentials, and to compare exact and approximate inference. (Since we wanted a large number of test samples, and since we wanted to vary the accuracy of the base classifier over a wide range, it would not have been practical to conduct this experiment with the real HeLa data.) In this experiment, for each trial, we randomly generated a synthetic similarity graph with 10 nodes split between 2 classes (out of a list of 4 total classes), as described above. We used 50% graph edge density for the Potts and voting potentials and 40% graph edge density for the AMN potential, since these densities were approximately optimal according to our preliminary experiments. We also generated simulated classifier likelihoods at each node using values of μ that corresponded to base classifier accuracies ranging from 50% to 90%.

For each generated classification problem, we built three different factor graphs according to the methods described above: a Potts model, a model that used AMN potentials, and a model that used voting potentials. Finally, we computed posterior marginal class probabilities using seven different inference algorithms: exact inference on the Potts model (EIPP), loopy belief propagation on the Potts model (LBP), exact inference on the model with AMN potentials (EIAMN), loopy belief propagation on the model with AMN potentials (LBAMN), exact inference on the model with voting potentials (EIVP), loopy belief propagation on the model with voting potentials (LBVP), and



(a) (N_1,N_2)=(5,5) (b) (N_1,N_2)=(2,8)

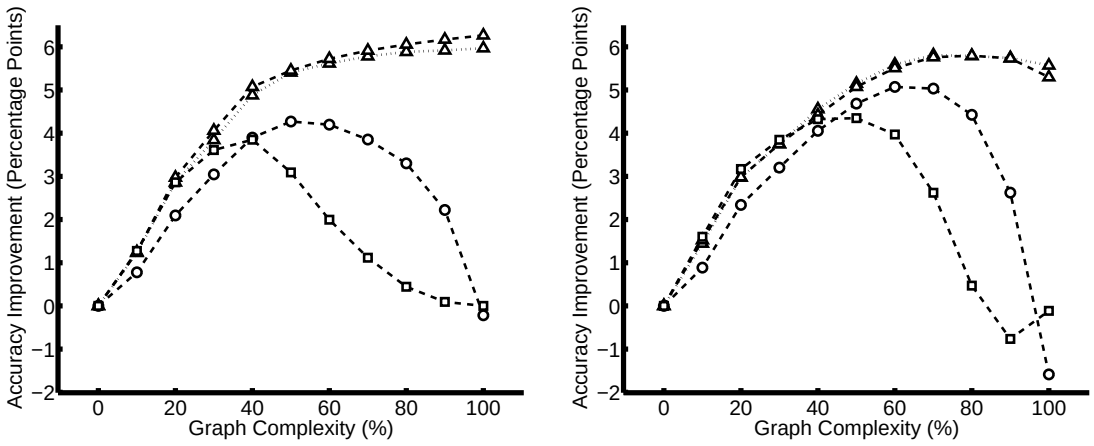
Figure 6: Accuracy Improvement vs. Base Accuracy with different inference methods on graphs with synthetic evidence: exact inference on the model with voting potentials (EIVP, $\triangle-$), loopy belief propagation on the model with voting potentials (LBVP, $\triangle--$), prior updating on the model with voting potentials (PU, $\triangle\cdots$), exact inference on the model with AMN potentials (EIAMN, $\square-$), loopy belief propagation on the model with AMN potentials (LBAMN, $\square--$), exact inference on the Potts model (EIPP, $\bigcirc-$) and loopy belief propagation on the Potts model (LBP, $\bigcirc--$). (a) Graphs with equal numbers of nodes from two classes, $(N_1, N_2) = (5,5)$. (b) Graphs with unequal numbers of nodes from two classes, $(N_1, N_2) = (2,8)$. 95% confidence bars (not shown) are smaller than the plot symbols.

prior updating on the model with voting potentials (PU). We evaluated each method by averaging its classification accuracy over all nodes in the graph, and then over 10,000 different graphs; the results are shown in Fig. 6.

As Fig. 6 illustrates, the methods using the voting potential (EIVP, LBVP, and PU) significantly outperformed the other potentials when the graph has an equal number of nodes from each class. For the unequal case, the voting potential had comparable performance to the Potts potential, and outperformed the AMN potential. In addition, the approximate methods (PU, LBVP, and LBP) did not differ substantially from their corresponding exact algorithms (EIVP, EIVP, and EIPP respectively), with the exception that LBAMN is substantially worse than EIAMN, and PU appears to be slightly worse than EIVP at lower base accuracies, and slightly better at higher base accuracies.

9.2 HeLa Data

Encouraged by the above results, we next considered classification accuracy for the real HeLa cell images. The factor graphs in this case are too large for exact inference, and therefore we cannot compare approximate and exact inference algorithms. Instead, we sought in our second experi-



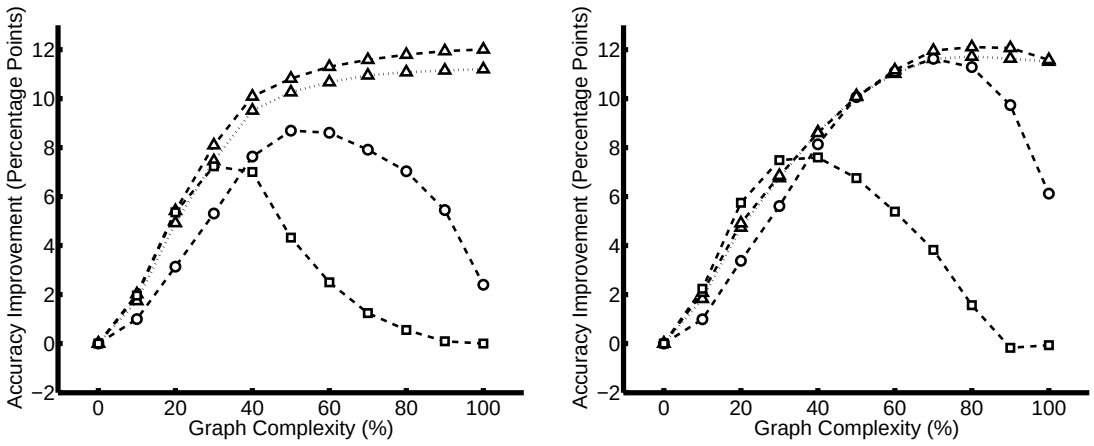
(a) (N_1,N_2)=(6,6) (b) (N_1,N_2)=(2,10)

Figure 7: Accuracy Improvement vs. Graph Complexity with different approximate inference methods on graphs with evidence from real data and base accuracy = 91.6%: loopy belief propagation on the model with voting potentials (LBVP, $\triangle--$), prior updating on the model with voting potentials (PU, $\triangle\cdots$), loopy belief propagation on the model with AMN potentials (LBAMN, $\square--$) and loopy belief propagation on the Potts model (LBP, $\bigcirc--$). (a) Graphs with equal numbers of nodes from two classes, $(N_1,N_2)=(6,6)$. (b) Graphs with unequal numbers of nodes from two classes, $(N_1,N_2)=(2,10)$. Confidence bars are not shown, since paired tests are more powerful for our experimental setup; see text for statistical comparisons.

ment to determine whether our approximate inference algorithms are able to improve classification accuracy substantially compared to our baseline SVM classifier.

For each trial in this experiment, we built a graph containing 12 cells in two classes. (We did not tell the classifier which two classes we used, so there were still 10 possible labels for each cell.) We assigned evidence to each cell using the SVM classifier described above; the base accuracy of this classifier was 91.6%. In order to evalute the inference performance on different base accuracies, we also selected the best 4 and 6 (out of 47) features to achieve the lower base accuracies of 70.6% and 83.1%, respectively. To test the algorithms' performance on a variety of problems, we adjusted d_{cutoff} to achieve levels of connectivity ranging from 0% to 100% of the possible edges; 0% graph complexity corresponds to a completely disconnected graph, in which each node's class is assigned using just the base classifier, while 100% means that the graph is fully connected.

We evaluated each algorithm's accuracy by 6-fold cross-validation: we trained the SVM using 5/6 of the images, used the other 1/6 of the images to construct testing networks, and recorded the improvement in classification accuracy compared to the base classifier in each testing network. We repeated this procedure 6 times so that every image appeared in the test partition once. Finally, we averaged the overall accuracy over 10 different splits into folds. The results in Fig. 7, Fig. 8, and Fig. 9 demonstrate that PU and LBVP can robustly achieve a good improvement in classification



(a) (N_1,N_2)=(6,6) (b) (N_1,N_2)=(2,10)

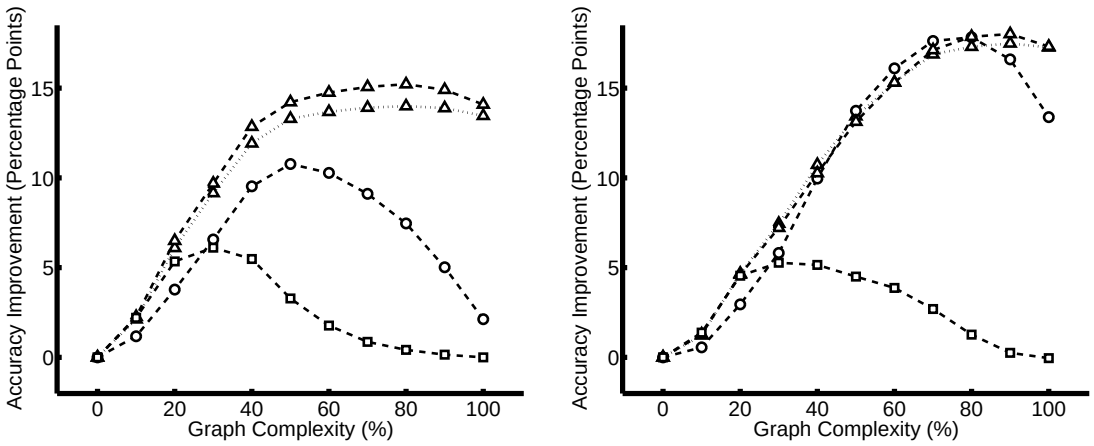
Figure 8: Accuracy Improvement vs. Graph Complexity with different approximate inference methods on graphs with evidence from real data and base accuracy = 83.1%: loopy belief propagation with voting potentials (LBVP, $\triangle --$), prior updating with voting potentials (PU, $\triangle \cdots$), loopy belief propagation with AMN potentials (LBAMN, $\square --$) and loopy belief propagation on the Potts model (LBP, $\bigcirc --$). (a) Graphs with equal numbers of nodes from two classes, $(N_1, N_2) = (6, 6)$. (b) Graphs with unequal numbers of nodes from two classes, $(N_1, N_2) = (2, 10)$. Confidence bars are not shown, since paired tests are more powerful for our experimental setup; see text for statistical comparisons.

accuracy on graphs with equal numbers of nodes from two classes, as compared to methods based on other potentials. (One-tailed paired t-test: $p = 0.0033, p = 0.0018, p = 0.0001$ for graphs with 91.6%, 83.1%, and 70.6% base accuracies, respectively. Tests were conducted at graph complexity 50%; p values are for comparison to closest competitor.)

On graphs with unequal class sizes, voting and Potts potentials achieved statistically comparable results when the graph complexity was tuned optimally. But, the performance of the voting-potential-based methods was more robust: their accuracy remained high for a wider range of graph complexities. (For example, one-tailed paired t-test for LBVP versus LBP: $p = 0.0036$ for graphs with 91.6% base accuracies at graph complexity 80%; $p = 0.0008$ for graphs with 83.1% base accuracies at graph complexity 90%; $p \leq 0.0001$ for graphs with 70.6% base accuracies at graph complexity 100%.)

9.3 Inference Speed

Our final experiment compares the computational efficiency of the different inference methods. Each point in Fig. 10 shows the average inference time per trial on different sizes of graphs with various algorithms (note the logarithmic time scale). Each graph has $N_1 = N_2$, uses four of the ten classes from the HeLa data set, and has 50% graph complexity. In this figure, we also included the processing time for loopy belief propagation with voting potentials using naive message com-



(a) (N_1,N_2)=(6,6) (b) (N_1,N_2)=(2,10)

Figure 9: Accuracy Improvement vs. Graph Complexity with different approximate inference methods on graphs with evidence from real data and base accuracy = 70.6%: loopy belief propagation on the model with voting potentials (LBVP, $\triangle--$), prior updating on the model with voting potentials (PU, $\triangle\cdots$), loopy belief propagation on the model with AMN potentials (LBAMN, $\square--$) and loopy belief propagation on the Potts model (LBP, $\bigcirc--$). (a) Graphs with equal numbers of nodes from two classes, $(N_1, N_2) = (6, 6)$. (b) Graphs with unequal numbers of nodes from two classes, $(N_1, N_2) = (2, 10)$. Confidence bars are not shown, since paired tests are more powerful for our experimental setup; see text for statistical comparisons.

putation (LBVPNC), which computes messages using ordinary marginalization rather than our new message computation algorithms. LBVPNC is expected to be faster then EIVP when the number of neighbors of one node is only a small fraction of the total number of nodes; otherwise LBVPNC could be much slower than EIVP, as is the case in this experiment.

The exact inference methods take time exponential in the size of the graph and are impractical to run for graphs of more than 12 nodes, while the processing times of PU, LBVP and LBP are much faster, and remain well under a second for the largest graphs tested. (With more than four classes, the exact inference times would rise even more quickly, and the graphs would have to be even smaller to allow exact inference.) The processing times for approximate inference methods are a combination of two factors: the number of iterations to converge and the time for each iteration. Fig. 11 shows the number of iterations needed for each method.

10. Related Work

The problem of how to quickly compute belief messages (or other similar quantities in an inference algorithm) is an important one, and several special cases of our decomposable structure have been previously described in the literature. One recent example is an algorithm for fast inference in

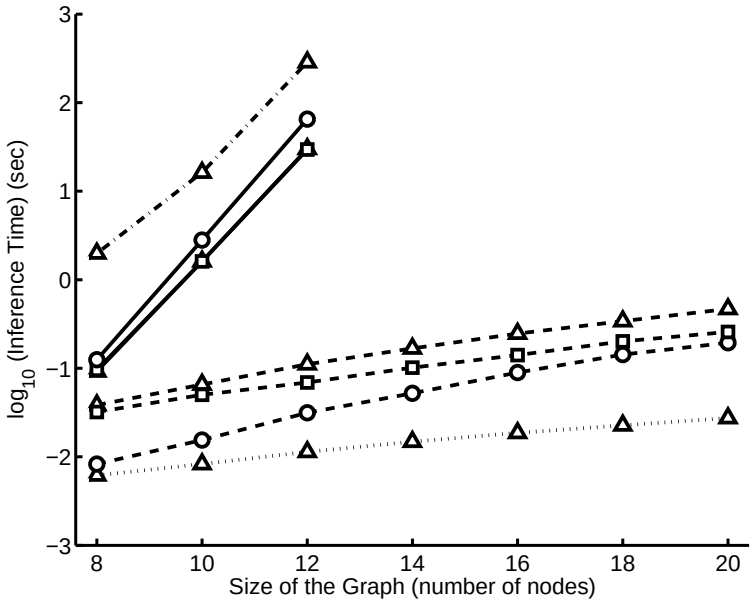


Figure 10: Dependence of inference time on graph size for different inference methods: exact inference on the model with voting potentials (EIVP, $\triangle-$), loopy belief propagation on the model with voting potentials (LBVP, $\triangle--$), loopy belief propagation on the model with voting potentials using naive message computation (LBVPNC, $\triangle\cdot-$), prior updating on the model with voting potentials (PU, $\triangle\cdots$), exact inference on the model with AMN potentials (EIAMN, $\square-$), loopy belief propagation on the model with AMN potentials (LBAMN, $\square--$), exact inference on the Potts model (EIPP, $\bigcirc-$) and loopy belief propagation on the Potts model (LBP, $\bigcirc--$). Values shown are times for one run of inference on the given graph, averaged over 12 trials.

hidden Markov models with structured transition matrices (Siddiqi and Moore, 2005). HMMs are a special case of chain-shaped factor graphs: there is one node representing the state x_t at each time step t, and there is a factor connecting the nodes x_t and x_{t+1} for each t. The potential functions for all factors are the same, and are equal to the transition matrix: $\varphi(x_t, x_{t+1}) = P(x_{t+1} \mid x_t)$.

The special structure for transition matrices proposed by Sidiqqi and Moore is called "Dense Mostly Constant," or DMC. In a DMC transition matrix for n states, each row contains k arbitrary entries in specified positions, and the other $n-k$ entries are all equal to a shared constant. The k arbitrary entries may be in different positions for each row, and the shared constant may also differ from row to row. Siddiqi and Moore show how to run the forward-backward, Viterbi, and Baum-Welsh algorithms quickly for HMMs with DMC transition matrices.

The DMC structure is a special case of our decomposable potential structure: any potential function corresponding to a DMC transition matrix can be written as a rank-one term plus a sparse term. More precisely,

$$\phi(x_t, x_{t+1}) = q(x_t) + \sum_{x,x' \in S} (w_{xx'} - q(x)) I(x_t, x) I(x_{t+1}, x').$$

Here $q(x_t)$ is the shared constant for the row corresponding to x_t, S is the set of transition matrix entries (x,x') which are allowed to differ from the shared constants, and $w_{xx'}$ is the value of the transition matrix entry at position (x,x'). The functions $q(x_t)$, $I(x_t,x)$, and $I(x_{t+1},x')$ each have arity 1; and, the set S is sparse, since it has at most $kn \ll n^2$ elements. Since DMC potentials are decomposable, our message calculation algorithm allows us to run belief propagation quickly; doing so is essentially equivalent to Sidiqqi and Moore's implementation of the forward-backward algorithm.

As we have pointed out above, the associative Markov network potential is also an example of our decomposable potential structure. The original paper on AMNs used an inference algorithm based on linear programming rather than on belief propagation (Taskar et al., 2004). However, the reason that their LP-based inference algorithm is tractable is exactly that they can take advantage of the AMN potential's special structure. There does not appear to be a simple way to extend their LP-based inference algorithm to the more general decomposable potential structure studied here, but this would be an interesting direction for future work. Another interesting direction for future work would be to extend their parameter-learning algorithm to handle more general structures like the decomposable potentials studied here.

In the context of a computer vision application, Felzenszwalb and Huttenlocher (2004) describe how to run loopy belief propagation quickly for a number of different pairwise potential functions. One that they consider is the Potts potential, and their algorithm for handling this potential takes advantage of the fact that it is a constant plus a sparse matrix, $1+(\omega-1)I$. They also consider pairwise potentials based on distance functions, which do not in general appear to be examples of our class of decomposable potential functions. On the other hand, they consider only potentials with up to two arguments.

In addition to the potentials studied in this paper, multiple examples of specific decomposable potentials exist in the literature. One of the most common forms is a potential used in directed graphical models (Bayes nets), in which the child's distribution depends on the sum of the parents' values (e.g., Frey and Kannan, 2000). Another good example is a potential used to represent probability distributions over graphs, in which a structure's probability depends on the degree of its nodes (Morris et al., 2003). All of these examples share a certain similarity to several of the special cases of decomposable potentials mentioned above, in that the speedup comes from accelerating the calculation of a single large sum within the message computation.

Another way to handle large sums is to build a sum tree: for example, the sum $x_1+x_2+x_3+x_4$ can be rewritten $[(x_1+x_2)+(x_3+x_4)]$. So, a potential with 4 arguments, $\varphi(x_1,x_2,x_3,x_4) = f(x_1+x_2+x_3+x_4)$, can be replaced by three smaller potentials and two new variables:

$$I(y_1 = x_1+x_2)\, I(y_2 = x_3+x_4)\, f(y_1+y_2).$$

Similarly, a potential that depends on a sum of n terms can be replaced by $n-1$ three-argument potentials and $n-2$ additional variables. See Liao et al. (2006) for an example of an algorithm based on this intuition. While this method works well for potentials that are functions of sums, it is not straightforward to extend it to more complicated potentials of the form we consider in this paper.

A final, and related, example of a tractable approximation to belief propagation is given by Barber (2001). Barber considers the directed version of the belief propagation algorithm, with conditional probability distribution functions of the form

$$P(x \mid \mathbf{s}) = f(\theta \cdot \mathbf{s}).$$

Here x is a node in the graph, $\mathbf{s}$ is the vector of parents of x, θ is a vector of fixed parameters, and f is a one-dimensional function with a tractable (approximate) Fourier expansion. This class of potentials is different from ours; perhaps an even larger class of potentials could be handled by combining the two techniques, arriving at a class of potentials that looked like

$$\varphi_j(x_1,\ldots,x_k) = \sum_i f_i\left(\sum_{l=1}^{L_j}\prod_{m=1}^{M_{jl}}\xi_{jlm}(x_{V(j,l,m)})\right)$$

where f_i is a basis function (e.g., a multiple of a sine wave if we are using Fourier expansions as above).

It is possible to find the most likely vector of labels for a Potts model using graph cuts when each variable x_i is binary. A related algorithm can be used for approximate inference with non-binary variables, and always finds a label vector within a factor of 2 of the most likely. (See Kleinberg and Tardos, 1999; Taskar et al., 2004, for a discussion.) It does not appear to be easily possible to extend this group of algorithms to k-way potentials such as the voting potential.

In our own previous work, we applied PU to image segmentation problems to identify cell regions (Chen et al., 2006b). In this case, the voting potential assumed that two sources of information were available: images of nuclear staining and of cell boundary locations, both of which were expected to be noisy. The nuclear staining provided an initial assignment of whether a pixel belongs to the background or to one of the cells, while the cell boundary image provided a probability estimate of whether two neighboring pixels should have the same label. The results indicated that PU provided efficient and accurate inference.

11. Conclusions and Future Work

We have examined the problem of classifying multiple dependent examples in a protein subcellular location pattern recognition task. We have compared several different graphical models and inference algorithms designed to solve this sort of structured classification problem, including one based on the novel voting potential function. The voting potential, like the previously studied Potts and AMN potentials, encodes the intuition that a variable is likely to have the same class as its neighbors. Our experiments show that the voting potential often does a better job of encoding this intuition than the Potts and AMN potentials do.

In addition to the new potential, we have presented new approximate inference algorithms: first, we have shown how to implement loopy belief propagation efficiently for networks with decomposable potentials, including the AMN and voting potentials. And second, we have suggested ignoring certain belief messages during LBP for the voting potential, resulting in an algorithm called Prior Updating. These fast algorithms enable us to use potentials like the voting potential on real data, where they would otherwise be impractical.

We believe that the voting potential function and the class of decomposable potentials will generalize to other graphs and other applications: for example, in image segmentation, it is common to use a potential which encourages nearby pixels to be part of the same object. With a potential similar to the voting potential described here, we could allow many neighboring pixels to vote on which object a particular pixel belongs to; and in fact, we have conducted initial experiments in this direction (Chen et al., 2006b). For another example, in regression and classification it is common to reject outliers to improve the robustness of the learned concept. In a given training set, multiple outliers may result from similar causes, such as being out of focus or in a different phase of the

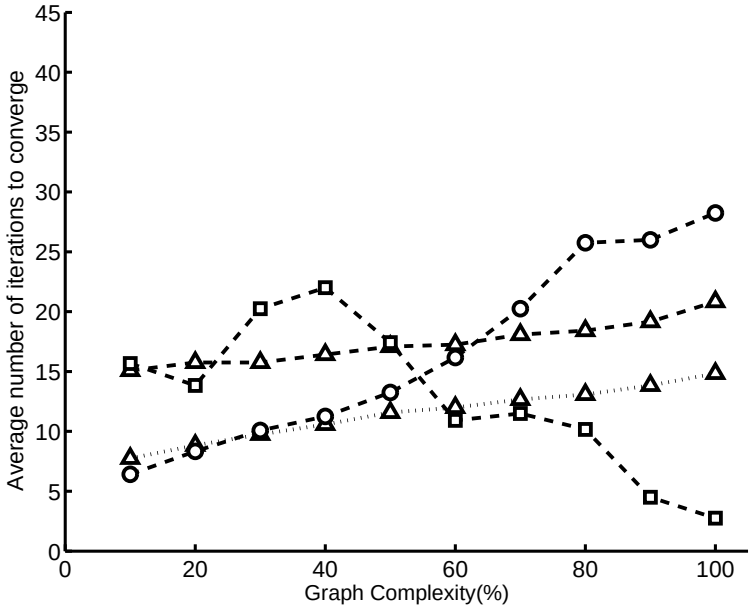


Figure 11: Dependence of number of iterations to converge on graph size for different inference methods: loopy belief propagation on the model with voting potentials (LBVP, $\triangle--$), prior updating on the model with voting potentials (PU, $\triangle\cdots$), loopy belief propagation on the model with AMN potentials (LBAMN, $\square--$) and loopy belief propagation on the Potts model (LBP, $\bigcirc--$). Values shown are numbers of iterations for one run of inference to converge on the given graph, averaged over 12 trials.

cell cycle. By using feature-space similarity to connect training examples in a graph, we can infer which types of outliers are present in the data and use this information to determine more accurately whether each point is an outlier.

In addition to the specific potentials we propose, our algorithms for calculating belief messages efficiently will also generalize to other domains. Whenever a graph contains a factor with a decomposable potential function, we can greatly reduce the time required to calculate belief messages from that factor. Our message calculation algorithm is exact; one might expect that further speed improvements would be possible if we are willing to accept approximate calculations. One promising avenue that we intend to explore is a "loopy" version of our message calculation algorithm. In such an algorithm, an inner loop of message passing would approximate the belief messages needed by the outer loop.

Acknowledgments

This work was supported in part by NSF grant EF-0331657. Facilities and infrastructure support were provided by NIH National Technology Center for Networks and Pathways grant U54 RR022241 and by NIH National Center for Biomedical Computing grant National U54 DA021519.

References

D. Barber. Tractable approximate belief propagation. In *Advanced Mean Field Methods: Theory and Practice*, pages 197–212. MIT Press, 2001.

M. V. Boland and R. F. Murphy. A neural network classifier capable of recognizing the patterns of all major subcellular structures in fluorescence microscope images of HeLa cells. *Bioinformatics*, 17:1213–1223, 2001.

C.-C. Chang and C.-J. Lin. *LIBSVM: A Library for Support Vector Machines*, 2001. Software available at `http://www.csie.ntu.edu.tw/~cjlin/libsvm`.

S.-C. Chen and R. F. Murphy. A graphical model approach to automated classification of protein subcellular location patterns in multi-cell images. *BMC Bioinformatics*, 7:90, 2006.

S.-C. Chen, G. J. Gordon, and R. F. Murphy. A novel approximate inference approach to automated classification of protein subcellular location patterns in multi-cell images. In *Proceedings of the 2006 IEEE International Symposium on Biomedical Imaging (ISBI)*, pages 558–561, 2006a.

S.-C. Chen, T. Zhao, G. J., Gordon, and R. F. Murphy. A novel graphical model approach to segmenting cell images. In *Proceedings of the 2006 IEEE Symposium on Computational Intelligence in Bioinformatics and Computational Biology (CIBCB)*, pages 1–8, 2006b.

X. Chen, M. Velliste, and R. F. Murphy. Automated interpretation of subcellular patterns in fluorescence microscope images for location proteomics. *Cytometry*, 69A:631–640, 2006c.

C. Cortes and V. Vapnik. Support vector networks. *Machine Learning*, 20:273–297, 1995.

P. F. Felzenszwalb and D. P. Huttenlocher. Efficient belief propagation for early vision. In *Proceedings of the 2004 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR)*, volume 1, pages 261–268, 2004.

B. J. Frey and A. Kannan. Accumulator networks: Suitors of local probability propagation. In *Advances in Neural Information Processing Systems (NIPS)*, pages 486–492, 2000.

E. Glory and R. F. Murphy. Automated subcellular location determination and high throughput microscopy. *Developmental Cell*, 12:7–16, 2007.

T. Heskes. On the uniqueness of loopy belief propagation fixed points. *Neural Computation*, 16: 2379–2413, 2004.

K. Huang and R. F. Murphy. Boosting accuracy of automated classification of fluorescence microscope images for location proteomics. *BMC Bioinformatics*, 5:78, 2004.

A. T. Ihler, J. W. Fisher, III, and A. S. Willsky. Loopy belief propagation: Convergence and effects of message errors. *Journal of Machine Learning Research*, 6:905–936, 2005.

U. Kjærulff. Inference in Bayesian networks using nested junction trees. In *Learning in Graphical Models*, pages 51–74. Kluwer Academic Press, 1998.

J. Kleinberg and E. Tardos. Approximation algorithms for classification problems with pairwise relationships: Metric labeling and Markov random fields. In *Proceedings of the 40th Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 14–23, 1999.

D. Koller and N. Friedman. *Bayesian Networks and Beyond*. 2007. Draft manuscript.

F. R. Kschischang, B. J. Frey, and H.-A. Loeliger. Factor graphs and the sum-product algorithm. *IEEE Transactions on Information Theory*, 47(2):498–519, 2001.

L. Liao, D. Fox, and H. Kautz. Location-based activity recognition. In Y. Weiss, B. Schölkopf, and J. Platt, editors, *Advances in Neural Information Processing Systems (NIPS)*, volume 18, pages 787–794, Cambridge, MA, 2006. MIT Press.

H.-T. Lin, C.-J. Lin, and R. C. Weng. *A Note on Platt's Probabilistic Outputs for Support Vector Machines*, 2003. Technical report, Department of Computer Science, National Taiwan University.

R. McEliece, D. MacKay, and J. Cheng. Turbo decoding as an instance of Pearl's belief propagation algorithm. *IEEE Journal on Selected Areas in Communications*, 16:140–152, 1998.

Q. D. Morris, B. J. Frey, and C. J. Paige. Denoising and untangling graphs using degree priors. In *Neural Information Processing Systems Conference (NIPS)*, pages 385–392, 2003.

K. Murphy, Y. Weiss, and M. Jordan. Loopy belief propagation for approximate inference—an empirical study. In *Proceedings of the 15th Annual Conference on Uncertainty in Artificial Intelligence (UAI)*, pages 467–475, 1999.

J. Pearl. *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. Morgan Kaufman, San Mateo, 1988.

J. Platt. Probabilistic outputs for support vector machines and comparisons to regularized likelihood methods. In *Advances in Large Margin Classifiers*, pages 61–74. MIT Press, 2000.

S. Siddiqi and A. Moore. Fast inference and learning in large-state-space HMMs. In *Proceedings of the 22nd International Conference on Machine Learning (ICML)*, pages 800–807, 2005.

B. Taskar, V. Chatalbashev, and D. Koller. Learning associative Markov networks. In *Proceedings of the 21st International Conference on Machine Learning (ICML)*, pages 102–109, 2004.

S. C. Tatikonda and M. I. Jordan. Loopy belief propagation and Gibbs measures. In *Proceedings of the 18th Conference on Uncertainty in Artificial Intelligence (UAI)*, pages 493–500, 2002.

Y. Weiss. Correctness of local probability propagation in graphical models with loops. *Neural Computation*, 12:1–41, 2000.

D.-Q. Zhang and S.-F. Chang. Learning to detect scene text using a higher-order MRF with belief propagation. *IEEE Workshop on Learning in Computer Vision and Pattern Recognition, in conjunction with CVPR (LCVPR)*, 6:101–108, 2004.