Journal of Machine Learning Research 12 (2011) 1349-1388 Submitted 4/10; Revised 1/11; Published 4/11

Faster Algorithms for Max-Product Message-Passing*

Julian J. McAuley† JULIAN.MCAULEY@NICTA.COM.AU
Tibério S. Caetano† TIBERIO.CAETANO@NICTA.COM.AU
Statistical Machine Learning Group
NICTA
Locked Bag 8001
Canberra ACT 2601, Australia

Editor: Tommi Jaakkola

Abstract

Maximum A Posteriori inference in graphical models is often solved via message-passing algorithms, such as the junction-tree algorithm or loopy belief-propagation. The exact solution to this problem is well-known to be exponential in the size of the maximal cliques of the triangulated model, while approximate inference is typically exponential in the size of the model's factors. In this paper, we take advantage of the fact that many models have maximal cliques that are larger than their constituent factors, and also of the fact that many factors consist only of latent variables (i.e., they do not depend on an observation). This is a common case in a wide variety of applications that deal with grid-, tree-, and ring-structured models. In such cases, we are able to decrease the exponent of complexity for message-passing by 0.5 for both exact *and* approximate inference. We demonstrate that message-passing operations in such models are equivalent to some variant of matrix multiplication in the tropical semiring, for which we offer an $O(N^{2.5})$ *expected-case* solution.

Keywords: graphical models, belief-propagation, tropical matrix multiplication

1. Introduction

It is well-known that exact inference in *tree-structured* graphical models can be accomplished efficiently by message-passing operations following a simple protocol making use of the distributive law (Aji and McEliece, 2000; Kschischang et al., 2001). It is also well-known that exact inference in *arbitrary* graphical models can be solved by the junction-tree algorithm; its efficiency is determined by the size of the maximal cliques after triangulation, a quantity related to the tree-width of the graph.

Figure 1 illustrates an attempt to apply the junction-tree algorithm to some graphical models containing cycles. If the graphs are not chordal ((a) and (b)), they need to be triangulated, or made chordal (red edges in (c) and (d)). Their clique-graphs are then guaranteed to be *junction-trees*, and the distributive law can be applied with the same protocol used for trees; see Aji and McEliece (2000) for a beautiful tutorial on exact inference in arbitrary graphs. Although the models in these

∗. Preliminary versions of this work appeared in The 27th International Conference on Machine Learning (ICML 2010), and the 13th International Conference on Artificial Intelligence and Statistics (AISTATS 2010), The NIPS 2009 Workshop on Learning with Orderings, The NIPS 2009 Workshop on Discrete Optimization in Machine Learning, and in Learning and Intelligent Optimization (LION 4).

†. Also at Research School of Information Sciences and Engineering, Australian National University, Canberra ACT 0200, Australia.

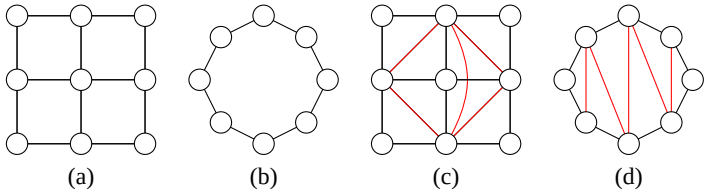


Figure 1: The models at left ((a) and (b)) can be triangulated ((c) and (d)) so that the junction-tree algorithm can be applied. Despite the fact that the new models have larger maximal cliques, the corresponding potentials are still factored over pairs of nodes only. Our algorithms exploit this fact.

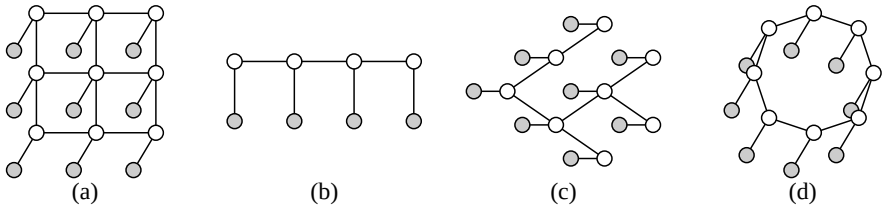


Figure 2: Some graphical models to which our results apply: *factors conditioned upon observations have fewer latent variables than purely latent factors*. White nodes correspond to latent variables, gray nodes to an observation. In other words, factors containing a gray node encode the *data likelihood*, whereas factors containing only white nodes encode *priors*. Expressed more simply, the 'node potentials' depend upon the observation, while the 'edge potentials' do not.

examples contain only pairwise factors, triangulation has increased the size of their maximal cliques, making exact inference substantially more expensive. Hence approximate solutions in the original graph (such as loopy belief-propagation, or inference in a loopy factor-graph) are often preferred over an exact solution via the junction-tree algorithm.

Even when the model's factors are the same size as its maximal cliques, neither exact nor approximate inference algorithms take advantage of the fact that many factors consist only of *latent* variables. In many models, those factors that are conditioned upon the observation contain fewer latent variables than the purely latent factors. Examples are shown in Figure 2. This encompasses a wide variety of models, including grid-structured models for optical flow and stereo disparity as well as chain and tree-structured models for text or speech.

In this paper, we exploit the fact that the maximal cliques (after triangulation) often have potentials that factor over subcliques, as illustrated in Figure 1. We will show that whenever this is the case, the expected computational complexity of message-passing between such cliques *can be improved* (both the asymptotic upper-bound and the actual runtime).

Additionally, we will show that this result can be applied in cliques *whose factors that are conditioned upon an observation* contain fewer latent variables than those factors consisting purely

of latent variables; the 'purely latent' factors can be pre-processed *offline*, allowing us to achieve the same benefits as described in the previous paragraph.

We show that these properties reveal themselves in a wide variety of real applications.

A core operation encountered in the junction-tree algorithm is that of computing the inner-product of two vectors $\mathbf{v}_a$ and $\mathbf{v}_b$. In the max-product semiring (used for MAP inference), the 'inner-product' becomes

$$\max_{i \in \{1 \ldots N\}} \{\mathbf{v}_a[i] \times \mathbf{v}_b[i]\}. \tag{1}$$

Our results stem from the realization that while (Equation 1) appears to be a *linear* time operation, it can be decreased to $O(\sqrt{N})$ (in the expected case) if we know the permutations that sort $\mathbf{v}_a$ and $\mathbf{v}_b$ (i.e., the order statistics of $\mathbf{v}_a$ and $\mathbf{v}_b$). These permutations can be obtained efficiently when the model factorizes as described above.

Preliminary versions of this work have appeared in McAuley and Caetano (2009), McAuley and Caetano (2010a), and McAuley and Caetano (2010b).

1.1 Summary of Results

A selection of the results to be presented in the remainder of this paper can be summarized as follows:

- Our speedups apply to the operation of *passing a single message*. As a result, our method can be used regardless of the message-passing protocol.
- We are able to lower the asymptotic expected running time of max-product message-passing for *any* discrete graphical model whose cliques factorize into lower-order terms.
- The results obtained are exactly those that would be obtained by the traditional version of the algorithm, that is, no approximations are used.
- Our algorithm also applies whenever factors that are conditioned upon an observation contain fewer latent variables than those factors that are not conditioned upon an observation, as in Figure 2 (in which case certain computations can be taken offline).
- For pairwise models satisfying the above properties, we obtain an expected speed-up of *at least* $\Omega(\sqrt{N})$ (assuming N states per node; Ω denotes an *asymptotic lower-bound*). For example, in models with third-order cliques containing pairwise terms, message-passing is reduced from $\Theta(N^3)$ to $O(N^2\sqrt{N})$, as in Figure 1(d). For pairwise models whose edge potential is not conditioned upon an observation, message-passing is reduced from $\Theta(N^2)$ to $O(N\sqrt{N})$, as in Figure 2.
- For cliques composed of K-ary factors, the expected speed-up generalizes to at least $\Omega(\frac{1}{K}N^{\frac{1}{K}})$, though it is *never asymptotically slower* than the original solution.
- The expected-case improvement is derived under the assumption that the order statistics of different factors are *independent*.
- If the different factors have 'similar' order statistics, the performance will be better than the expected case.

- If the different factors have 'opposite' order statistics, the performance will be worse than the expected case, but is never asymptotically more expensive than the traditional version of the algorithm.

Our results do not apply for every semiring $(\oplus, \otimes)$, but only to those whose 'addition' operation defines an order (for example, min or max); we also assume that under this ordering, our 'multiplication' operator $\otimes$ satisfies

$$a < b \wedge c < d \;\; \Rightarrow \;\; a \otimes c < b \otimes d. \tag{2}$$

Thus our results certainly apply to the *max-sum* and *min-sum* ('tropical') semirings (as well as *max-product* and *min-product*, assuming non-negative potentials), but not for *sum-product* (for example). Consequently, our approach is useful for computing MAP-states, but cannot be used to compute marginal distributions. We also assume that the domain of each node is *discrete*.

We shall initially present our algorithm in terms of *pairwise* graphical models such as those shown in Figure 2. In such models message-passing is precisely equivalent to matrix-vector multiplication over our chosen semiring. Later we shall apply our results to models such as those in Figure 1, wherein message-passing becomes some variant of matrix multiplication. Finally we shall explore other applications besides message-passing that make use of tropical matrix multiplication as a subroutine, such all-pairs shortest-path problems.

1.2 Related Work

There has been previous work on speeding-up message-passing algorithms by exploiting different types of structure in certain graphical models. For example, Kersting et al. (2009) study the case where different cliques share the same potential function. In Felzenszwalb and Huttenlocher (2006), fast message-passing algorithms are provided for cases in which the potential of a 2-clique is only dependent on the *difference* of the latent variables (which is common in some computer vision applications); they also show how the algorithm can be made faster if the graphical model is a bipartite graph. In Kumar and Torr (2006), the authors provide faster algorithms for the case in which the potentials are *truncated*, whereas in Petersen et al. (2008) the authors offer speed-ups for models that are specifically grid-like.

The latter work is perhaps the most similar in spirit to ours, as it exploits the fact that certain factors can be *sorted* in order to reduce the search space of a certain maximization problem.

Another course of research aims at speeding-up message-passing algorithms by using 'informed' scheduling routines, which may result in faster convergence than the random schedules typically used in loopy belief-propagation and inference in factor graphs (Elidan et al., 2006). This branch of research is orthogonal to our own in the sense that our methods can be applied independently of the choice of message passing protocol.

Another closely related paper is that of Park and Darwiche (2003). This work can be seen to compliment ours in the sense that it exploits essentially the same type of factorization that we study, though it applies to *sum-product* versions of the algorithm, rather than the *max-product* version that we shall study. Kjærulff (1998) also exploits factorization within cliques of junction-trees, albeit a different type of factorization than that studied here.

In Section 4, we shall see that our algorithm is closely related to a well-studied problem known as 'tropical matrix multiplication' (Kerr, 1970). The worst-case complexity of this problem has been studied in relation to the all-pairs shortest-path problem (Alon et al., 1997; Karger et al., 1993).

Example	description
$A;B$	capital letters refer to sets of nodes (or similarly, cliques);
$A\cup B;A\cap B;A\setminus B$	standard set operators are used ($A\setminus B$ denotes set difference);
$\mathrm{dom}(A)$	the domain of a set; this is just the Cartesian product of the domains of each element in the set;
$\mathbf{P}$	bold capital letters refer to arrays;
$\mathbf{x}$	bold lower-case letters refer to vectors;
$\mathbf{x}[a]$	vectors are indexed using square brackets;
$\mathbf{P}[n]$	similarly, square brackets are used to index a *row* of a 2-d array,
$\mathbf{P}[\mathbf{n}]$	or a row of an $(\lvert\mathbf{n}\rvert+1)$-dimensional array;
$\mathbf{P}^X;\mathbf{v}^a$	superscripts are just labels, that is, $\mathbf{P}^X$ is an array, $\mathbf{v}^a$ is a vector;
$\mathbf{v}_a$	*constant* subscripts are also labels, that is, if a is a constant, then $\mathbf{v}_a$ is a constant vector;
$x_i;\mathbf{x}_A$	*variable* subscripts define variables; the subscript defines the domain of the variable;
$\mathbf{n}\vert_X$	if $\mathbf{n}$ is a constant vector, then $\mathbf{n}\vert_X$ is the *restriction* of that vector to those indices corresponding to variables in X (assuming that X is an ordered set);
$\Phi_A;\Phi_A(\mathbf{x}_A)$	a function over the variables in a set A; the argument $\mathbf{x}_A$ will be suppressed if clear, given that 'functions' are essentially arrays for our purposes;
$\Phi_{i,j}(x_i,x_j)$	a function over a pair of variables (x_i,x_j);
$\Phi_A(\mathbf{n}\vert_B;\mathbf{x}_{A\setminus B})$	if one argument to a function is constant (here $\mathbf{n}\vert_B$), then it becomes a function over fewer variables (in this case, only $\mathbf{x}_{A\setminus B}$ is free);

Table 1: Notation

2. Background

The notation we shall use is briefly defined in Table 1. We shall assume throughout that the *max-product* semiring is being used, though our analysis is almost identical for any suitable choice.

MAP-inference in a graphical model $\mathcal{G}$ consists of solving an optimization problem of the form

$$\hat{\mathbf{x}} = \operatorname*{argmax}_{\mathbf{x}} \prod_{C\in\mathcal{C}} \Phi_C(\mathbf{x}_C),$$

where $\mathcal{C}$ is the set of maximal cliques in $\mathcal{G}$. This problem is often solved via *message-passing* algorithms such as the junction-tree algorithm, loopy belief-propagation, or inference in a factor-graph (Aji and McEliece, 2000; Weiss, 2000; Kschischang et al., 2001).

Often, the clique-potentials $\Phi_C(\mathbf{x}_C)$ shall be decomposable into several smaller factors, that is,

$$\Phi_C(\mathbf{x}_C) = \prod_{F\subseteq C} \Phi_F(\mathbf{x}_F).$$

Some simple motivating examples are shown in Figure 3: a model for pose estimation from Sigal and Black (2006), a 'skip-chain CRF' from Galley (2006), and a model for shape-matching from Coughlan and Ferreira (2002). In each case, the triangulated model has third-order cliques, but the potentials are only pairwise. Other examples have already been shown in Figure 1; analogous cases are ubiquitous in many real applications.

It will often be more convenient to express our objective function as being conditioned upon some *observation*, $\mathbf{y}$. Thus our optimization problem becomes

$$\hat{\mathbf{x}}(\mathbf{y}) = \operatorname*{argmax}_{\mathbf{x}} \prod_{C \in \mathcal{C}} \Phi_C(\mathbf{x}_C|\mathbf{y}) \tag{3}$$

(for simplicity when we discuss 'cliques' we are referring to sets of *latent* variables).

Further factorization may be possible if we express (Equation 3) in terms of those factors that depend upon the observation $\mathbf{y}$, and those that do not:

$$\hat{\mathbf{x}}(\mathbf{y}) = \operatorname*{argmax}_{\mathbf{x}} \prod_{C \in \mathcal{C}} \Big\{ \underbrace{\prod_{F \subseteq C} \Phi_F(\mathbf{x}_F)}_{\text{data-independent}} \times \underbrace{\prod_{Q \subseteq C} \Phi_Q(\mathbf{x}_Q|\mathbf{y})}_{\text{data-dependent}} \Big\},$$

We shall say that those factors that are not conditioned on the observation are 'data-independent'.

Our results shall apply to message-passing equations in those cliques C where for each data-independent factor F we have $F \subset C$, *or* for each data-dependent factor Q we have $Q \subset C$, that is, when all F or all Q in C are *proper* subsets of C. In such cases we say that the clique C is *factorizable*.

The fundamental step encountered in message-passing algorithms is defined below. The message from a clique X to an intersecting clique Y (both sets of *latent* variables) is defined by

$$m_{X \to Y}(\mathbf{x}_{X \cap Y}) = \max_{\mathbf{x}_{X \setminus Y}} \left\{ \Phi_X(\mathbf{x}_X) \prod_{Z \in \Gamma(X) \setminus Y} m_{Z \to X}(\mathbf{x}_{X \cap Z}) \right\} \tag{4}$$

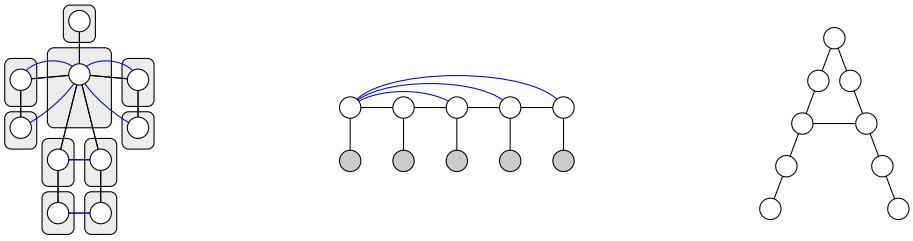
(where $\Gamma(X)$ is the set of neighbors of the clique X, that is, the set of cliques that intersect with X). If such messages are computed after X has received messages from all of its neighbors except Y (i.e., $\Gamma(X) \setminus Y$), then this defines precisely the update scheme used by the junction-tree algorithm. The same update scheme is used for loopy belief-propagation, though it is done iteratively in a randomized fashion.

After all messages have been passed, the MAP-state for a set of latent variables M (assumed to be a subset of a single clique X) is computed using

$$m_M(\mathbf{x}_M) = \max_{\mathbf{x}_{X \setminus M}} \left\{ \Phi_X(\mathbf{x}_X) \prod_{Z \in \Gamma(X)} m_{Z \to X}(\mathbf{x}_{X \cap Z}) \right\}. \tag{5}$$

For cliques that are *factorizable* (according to our previous definition), both (Equation 4) and (Equation 5) take the form

$$m_M(\mathbf{x}_M) = \max_{\mathbf{x}_{X \setminus M}} \left\{ \prod_{F \subseteq X} \Phi_F(\mathbf{x}_F) \prod_{Q \subseteq X} \Phi_Q(\mathbf{x}_Q|\mathbf{y}) \right\}. \tag{6}$$



(a) (b) (c)

Figure 3: (a) A model for pose reconstruction from Sigal and Black (2006); (b) A 'skip-chain CRF' from Galley (2006); (c) A model for deformable matching from Coughlan and Ferreira (2002). Although the (triangulated) models have cliques of size three, their potentials factorize into pairwise terms.

Note that we always have $Z \cap X \subset X$ for messages $Z \to X$, meaning that the presence of the messages has no effect on the 'factorizability' of (Equation 6).

Algorithm 1 gives the traditional solution to this problem, which does not exploit the factorization of $\Phi_X(\mathbf{x}_X)$. This algorithm runs in $\Theta(N^{|X|})$, where N is the number of states per node, and $|X|$ is the size of the clique X (for a given $\mathbf{x}_X$, we treat computing $\prod_{F \subset X} \Phi_F(\mathbf{x}_F)$ as a constant time operation, as our optimizations shall not modify this cost).

In the following sections, we shall consider the two types of factorizability separately: first, in Section 3, we shall consider cliques X whose messages take the form

$$m_M(\mathbf{x}_M) = \max_{\mathbf{x}_{X \setminus M}} \left\{ \Phi_X(\mathbf{x}_X) \prod_{Q \subset X} \Phi_Q(\mathbf{x}_Q|\mathbf{y}) \right\}.$$

We say that such cliques are *conditionally factorizable* (since all conditional terms factorize); examples are shown in Figure 2. Next, in Section 4, we consider cliques whose messages take the form

$$m_M(\mathbf{x}_M) = \max_{\mathbf{x}_{X \setminus M}} \prod_{F \subset X} \Phi_F(\mathbf{x}_F).$$

We say that such cliques are *latently factorizable* (since terms containing only latent variables factorize); examples are shown in Figure 1.

3. Optimizing Algorithm 1: Conditionally Factorizable Models

In order to specify a more efficient version of Algorithm 1, we begin by considering the simplest nontrivial *conditionally factorizable* model: a pairwise model in which each latent variable depends upon the observation, that is,

$$\hat{\mathbf{x}}(\mathbf{y}) = \operatorname*{argmax}_{\mathbf{x}} \underbrace{\prod_{i \in \mathcal{N}} \Phi_i(x_i|y)}_{\text{node potential}} \times \underbrace{\prod_{(i,j) \in \mathcal{E}} \Phi_{i,j}(x_i, x_j)}_{\text{edge potential}}. \tag{7}$$

This is the type of model depicted in Figure 2 and encompasses a large class of grid- and tree-structured models. Using our previous definitions, we say that the node potentials are 'data-dependent', whereas the edge potentials are 'data-independent'.

Algorithm 1 Brute-force computation of max-marginals

Input: a clique X whose max-marginal $m_M(\mathbf{x}_M)$ (where $M \subset X$) we wish to compute; assume that each node in X has domain $\{1 \dots N\}$

for $\mathbf{m} \in \mathrm{dom}(M)$ {i.e., $\{1 \dots N\}^{|M|}$} **do**
  $max \coloneqq -\infty$
  for $\mathbf{z} \in \mathrm{dom}(X \setminus M)$ **do**
    if $\prod_{F \subset X} \Phi_F(\mathbf{m}|_F; \mathbf{z}|_F) > max$ **then**
      $max \coloneqq \prod_{F \subset X} \Phi_F(\mathbf{m}|_F; \mathbf{z}|_F)$
    end if
  end for {this loop takes $\Theta(N^{|X \setminus M|})$}
  $m_M(\mathbf{m}) \coloneqq max$
end for {this loop takes $\Theta(N^{|X|})$}
Return: m_M

Message-passing in models of the type shown in (Equation 7) takes the form

$$m_{A \to B}(x_i) = \Phi_i(x_i|y) \times \max_{x_j} \Phi_j(x_j|y) \times \Phi_{i,j}(x_i, x_j) \tag{8}$$

(where $A = \{i, j\}$ and $B = \{i, k\}$). Note once again that in (Equation 8) we are not concerned solely with exact inference via the junction-tree algorithm. In many models, such as grids and rings, (Equation 7) shall be solved *approximately* by means of either loopy belief-propagation, or inference in a factor-graph, which consists of solving (Equation 8) according to protocols other than the optimal junction-tree protocol.

It is useful to consider $\Phi_{i,j}$ in (Equation 8) as an $N \times N$ *matrix*, and Φ_j as an N-dimensional *vector*, so that solving (Equation 8) is precisely equivalent to matrix-vector multiplication in the max-product semiring. For a particular value $x_i = q$, (Equation 8) becomes

$$m_{A \to B}(q) = \Phi_i(q|y) \times \max_{x_j} \underbrace{\Phi_j(x_j|y)}_{\mathbf{v}_a} \times \underbrace{\Phi_{i,j}(q, x_j)}_{\mathbf{v}_b}, \tag{9}$$

which is precisely the 'max-product inner-product' operation that we claimed was critical in Section 1.

As we have previously suggested, it will be possible to solve (Equation 9) efficiently if we know the order statistics of $\mathbf{v}_a$ and $\mathbf{v}_b$, that is, if we know the permutations that sort Φ_j and every row of $\Phi_{i,j}$ in (Equation 8). Sorting Φ_j takes $\Theta(N \log N)$, whereas sorting every row of $\Phi_{i,j}$ takes $\Theta(N^2 \log N)$ ($\Theta(N \log N)$ for each of N rows). The critical point to be made is that $\Phi_{i,j}(x_i, x_j)$ *does not depend on the observation*, meaning that its order statistics can be obtained *offline* in several applications.

The following elementary lemma is the key observation required in order to solve (Equation 1), and therefore (Equation 9) efficiently:

Lemma 1 *For any index q, the solution to $p = \mathrm{argmax}_{i \in \{1 \dots N\}} \{\mathbf{v}_a[i] \times \mathbf{v}_b[i]\}$ must have $\mathbf{v}_a[p] \geq \mathbf{v}_a[q]$ or $\mathbf{v}_b[p] \geq \mathbf{v}_b[q]$. Therefore, having computed $\mathbf{v}_a[q] \times \mathbf{v}_b[q]$, we can find 'p' by computing only those products $\mathbf{v}_a[i] \times \mathbf{v}_b[i]$ where either $\mathbf{v}_a[i] > \mathbf{v}_a[q]$ or $\mathbf{v}_b[i] > \mathbf{v}_b[q]$.*

Algorithm 2 Find i such that $\mathbf{v}_a[i] \times \mathbf{v}_b[i]$ is maximized

Input: two vectors $\mathbf{v}_a$ and $\mathbf{v}_b$, and permutation functions p_a and p_b that sort them in decreasing order (so that $\mathbf{v}_a[p_a[1]]$ is the largest element in $\mathbf{v}_a$)

1: **Initialize:** $start \coloneqq 1$, $end_a \coloneqq p_a^{-1}[p_b[1]]$, $end_b \coloneqq p_b^{-1}[p_a[1]]$ {if $end_b = k$, then the largest element in $\mathbf{v}_a$ has the same index as the k^{th} largest element in $\mathbf{v}_b$}
2: $best \coloneqq p_a[1]$, $max \coloneqq \mathbf{v}_a[best] \times \mathbf{v}_b[best]$
3: **if** $\mathbf{v}_a[p_b[1]] \times \mathbf{v}_b[p_b[1]] > max$ **then**
4: $best \coloneqq p_b[1]$, $max \coloneqq \mathbf{v}_a[best] \times \mathbf{v}_b[best]$
5: **end if**
6: **while** $start < end_a$ {in practice, we could also stop if $start < end_b$, but the version given here is the one used for analysis in Appendix A} **do**
7: $start \coloneqq start + 1$
8: **if** $\mathbf{v}_a[p_a[start]] \times \mathbf{v}_b[p_a[start]] > max$ **then**
9: $best \coloneqq p_a[start]$
10: $max \coloneqq \mathbf{v}_a[best] \times \mathbf{v}_b[best]$
11: **end if**
12: **if** $p_b^{-1}[p_a[start]] < end_b$ **then**
13: $end_b \coloneqq p_b^{-1}[p_a[start]]$
14: **end if**
15: {repeat lines 8–14, interchanging a and b}
16: **end while** {this loop takes *expected time* $O(\sqrt{N})$}
17: **Return:** *best*

This observation is used to construct Algorithm 2. Here we iterate through the indices starting from the largest values of $\mathbf{v}_a$ and $\mathbf{v}_b$, stopping once both indices are 'behind' the maximum value found so far (which we then know is the maximum). This algorithm is demonstrated pictorially in Figure 4. Note that Lemma 1 only depends upon the *relative* values of elements in $\mathbf{v}_a$ and $\mathbf{v}_b$, meaning that the number of computations that must be performed is purely a function of their *order statistics* (i.e., it does not depend on the actual values of $\mathbf{v}_a$ or $\mathbf{v}_b$).

If Algorithm 2 can solve (Equation 9) in $O(f(N))$, then we can solve (Equation 8) in $O(Nf(N))$. Determining precisely the running time of Algorithm 2 is not trivial, and will be explored in depth in Appendix A. At this stage we shall state an upper-bound on the true complexity in the following theorem:

Theorem 2 *The* expected *running time of Algorithm 2 is $O(\sqrt{N})$, yielding a speed-up of at least $\Omega(\sqrt{N})$ in cliques containing pairwise factors. This expectation is derived under the assumption that $\mathbf{v}_a$ and $\mathbf{v}_b$ have independent order statistics.*

Algorithm 3 uses Algorithm 2 to solve (Equation 8), where we assume that the order statistics of the rows of $\Phi_{i,j}$ have been obtained offline.

While the offline cost of sorting is not problematic in situations where the model is to be repeatedly reused on several observations, it can be avoided in two situations. Firstly, many models have a 'homogeneous' prior, that is, the same prior is shared amongst every edge (or clique) of the model. In such cases, only a single 'copy' of the prior needs to be sorted, meaning that in any model

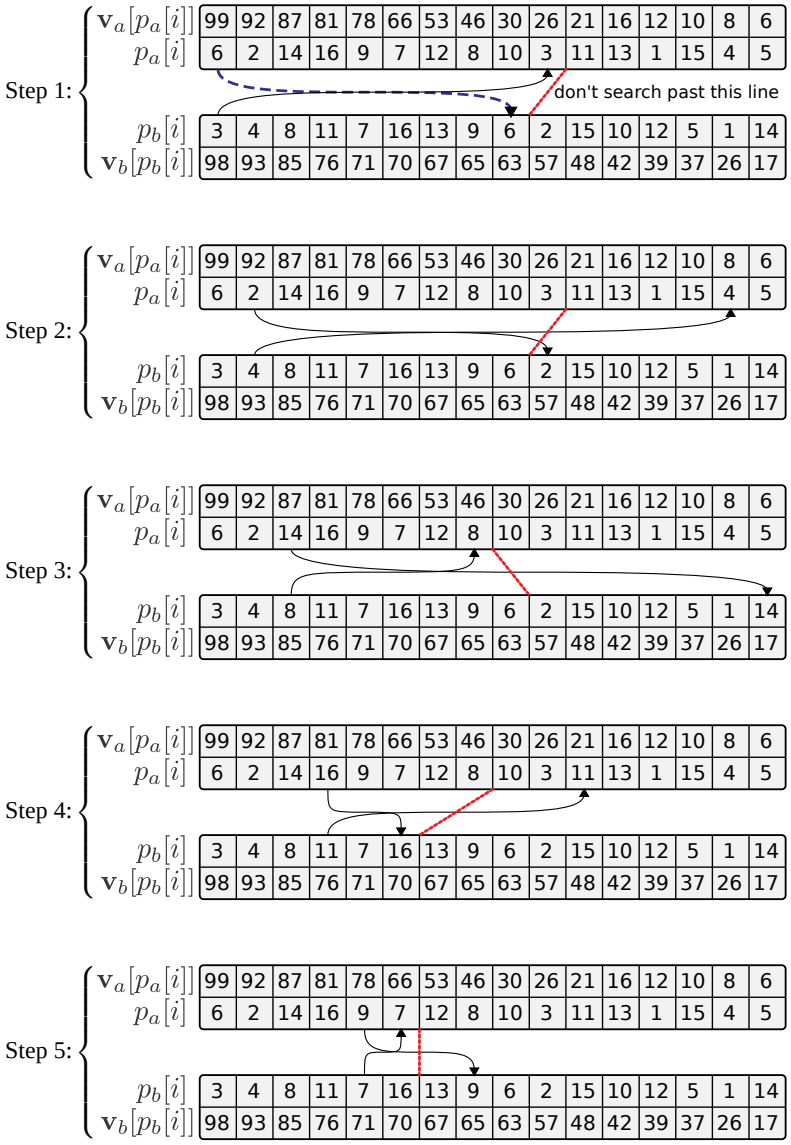


Figure 4: Algorithm 2, explained pictorially. The arrows begin at $p_a[start]$ and $p_b[start]$; the red dashed line connects end_a and end_b, behind which we need not search; a dashed arrow is used when a new maximum is found. Note that in the event that $\mathbf{v}_a$ and $\mathbf{v}_b$ contain repeated elements, they can be sorted arbitrarily.

containing $\Omega(\log N)$ edges, speed improvements can be gained over the naïve implementation. Secondly, where an iterative algorithm (such as loopy belief-propagation) is to be used, the sorting step need only take place prior to the *first* iteration; if $\Omega(\log N)$ iterations of belief-propagation are to be performed (or in a homogeneous model, if the number of edges multiplied by the number of

Algorithm 3 Solve (Equation 8) using Algorithm 2

Input: a potential $\Phi_{i,j}(a,b) \times \Phi_i(a|y_i) \times \Phi_j(b|y_j)$ whose max-marginal $m_i(x_i)$ we wish to compute, and a set of permutation functions $\mathbf{P}$ such that $\mathbf{P}[i]$ sorts the i^{th} row of $\Phi_{i,j}$ (in decreasing order).

1: compute the permutation function p_a by sorting Ψ_j {takes $\Theta(N \log N)$}
2: **for** $q \in \{1 \ldots N\}$ **do**
3: $(\mathbf{v}_a, \mathbf{v}_b) \coloneqq (\Psi_j, \Phi_{i,j}(q, x_j | y_i, y_j))$
4: $best \coloneqq Algorithm2\,(\mathbf{v}_a, \mathbf{v}_b, p_a, \mathbf{P}[q])$ $\{O(\sqrt{N})\}$
5: $m_{A \to B}(q) \coloneqq \Phi_i(q) \times \Phi_j(best) \times \Phi_{i,j}(q, best | y_i, y_j)$
6: **end for** {this loop takes *expected time* $O(N\sqrt{N})$}
7: **Return:** $m_{A \to B}$

iterations is $\Omega(\log N)$), we shall again gain speed improvements even when the sorting step is done online.

In fact, the second of these conditions obviates the need for 'conditional factorizability' (or 'data-independence') altogether. In other words, in *any* pairwise model in which $\Omega(\log N)$ iterations of belief-propagation are to be performed, *the pairwise terms need to be sorted only during the first iteration*. Thus these improvements apply to those models in Figure 1, so long as the number of iterations of belief-propagation is $\Omega(\log N)$.

4. Latently Factorizable Models

Just as we considered the simplest *conditionally factorizable* model in Section 3, we now consider the simplest nontrivial *latently factorizable* model: a clique of size three containing pairwise factors. In such a case, our aim is to compute

$$m_{i,j}(x_i, x_j) = \max_{x_k} \Phi_{i,j,k}(x_i, x_j, x_k), \tag{10}$$

which we have assumed takes the form

$$m_{i,j}(x_i, x_j) = \max_{x_k} \Phi_{i,j}(x_i, x_j) \times \Phi_{i,k}(x_i, x_k) \times \Phi_{j,k}(x_j, x_k).$$

For a particular value of $(x_i, x_j) = (a, b)$, we must solve

$$m_{i,j}(a,b) = \Phi_{i,j}(a,b) \times \max_{x_k} \underbrace{\Phi_{i,k}(a, x_k)}_{\mathbf{v}_a} \times \underbrace{\Phi_{j,k}(b, x_k)}_{\mathbf{v}_b}, \tag{11}$$

which again is in precisely the form shown in (Equation 1).

Just as (Equation 8) resembled matrix-vector multiplication, there is a close resemblance between (Equation 11) and the problem of matrix-matrix multiplication in the max-product semiring (often referred to as 'tropical matrix multiplication', 'funny matrix multiplication', or simply 'max-product matrix multiplication'). While traditional matrix multiplication is well-known to have a subcubic worst-case solution (see Strassen, 1969), the version in (Equation 11) has no known subcubic solution (the fastest known solution is $O(N^3/\log N)$, but there is no known solution that runs in $O(N^{3-\varepsilon})$ (Chan, 2007); Kerr (1970) shows that no subcubic solution exists under certain models of computation). The worst-case complexity of solving (Equation 11) can also be shown to be

Algorithm 4 Use Algorithm 2 to compute the max-marginal of a 3-clique containing pairwise factors

Input: a potential $\Phi_{i,j,k}(a,b,c) = \Phi_{i,j}(a,b) \times \Phi_{i,k}(a,c) \times \Phi_{j,k}(b,c)$ whose max-marginal $m_{i,j}(x_i,x_j)$ we wish to compute

for $n \in \{1 \dots N\}$ **do**
  compute $\mathbf{P}^i[n]$ by sorting $\Phi_{i,k}(n,x_k)$ {takes $\Theta(N \log N)$}
  compute $\mathbf{P}^j[n]$ by sorting $\Phi_{j,k}(n,x_k)$ {$\mathbf{P}^i$ and $\mathbf{P}^j$ are $N \times N$ arrays, each row of which is a permutation; $\Phi_{i,k}(n,x_k)$ and $\Phi_{j,k}(n,x_k)$ are functions over x_k, since n is constant in this expression}
end for {this loop takes $\Theta(N^2 \log N)$}
for $(a,b) \in \{1 \dots N\}^2$ **do**
  $(\mathbf{v}_a, \mathbf{v}_b) \coloneqq \left(\Phi_{i,k}(a,x_k), \Phi_{j,k}(b,x_k)\right)$
  $(p_a, p_b) \coloneqq \left(\mathbf{P}^i[a], \mathbf{P}^j[b]\right)$
  $best \coloneqq Algorithm2\,(\mathbf{v}_a, \mathbf{v}_b, p_a, p_b)$ {takes $O(\sqrt{N})$}
  $m_{i,j}(a,b) \coloneqq \Phi_{i,j}(a,b) \times \Phi_{i,k}(a,best) \times \Phi_{j,k}(b,best)$
end for {this loop takes $O(N^2\sqrt{N})$}
{the total running time is $O(N^2 \log N + N^2\sqrt{N})$, which is dominated by $O(N^2\sqrt{N})$}
Return: $m_{i,j}$

equivalent to the all-pairs shortest-path problem, which is studied in Alon et al. (1997). Although we shall not improve the worst-case complexity, Algorithm 2 leads to far better *expected-case* performance than existing solutions.

In principle Strassen's algorithm could be used to perform *sum-product* inference in the setting we discuss here, and indeed there has been some work on performing sum-product inference in graphical models that factorize (Park and Darwiche, 2003). Interestingly, there is also a sub-quadratic solution to sum-product matrix-vector multiplication that requires preprocessing (Williams, 2007), that is, the sum-product version of the setting we discussed in Section 3.

A prescription of how Algorithm 2 can be used to solve (Equation 10) is given in Algorithm 4. As we mentioned in Section 3, the expected-case running time of Algorithm 2 is $O(\sqrt{N})$, meaning that the time taken to solve Algorithm 4 is $O(N^2\sqrt{N})$.

5. Extensions

So far we have only considered the case of *pairwise* graphical models, though as mentioned our results can in principle be applied to any conditionally or latently factorizable models, no matter the size of the factors. Essentially our results about matrices become results about tensors. We first treat latently factorizable models, after which the same ideas can be applied to conditionally factorizable models.

5.1 An Extension to Higher-Order Cliques with Three Factors

The simplest extension that we can make to Algorithms 2, 3, and 4 is to note that they can be applied even when there are several overlapping terms in the factors. For instance, Algorithm 4 can

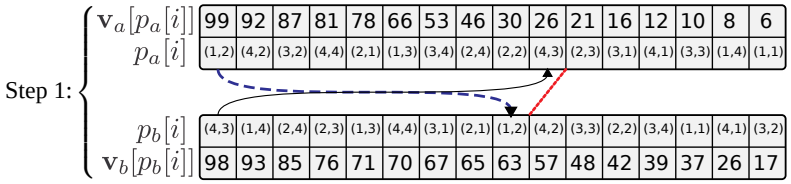


Figure 5: The reasoning applied in Algorithm 2 applies even when the elements of p_a and p_b are multidimensional indices.

be adapted to solve

$$m_{i,j}(x_i,x_j) = \max_{x_k,x_m} \Phi_{i,j}(x_i,x_j) \times \Phi_{i,k,m}(x_i,x_k,x_m) \times \Phi_{j,k,m}(x_j,x_k,x_m), \tag{12}$$

and similar variants containing three factors. Here both x_k and x_m are shared by $\Phi_{i,k,m}$ and $\Phi_{j,k,m}$. We can follow precisely the reasoning of the previous section, except that when we sort $\Phi_{i,k,m}$ (similarly $\Phi_{j,k,m}$) for a fixed value of x_i, we are now sorting an *array* rather than a *vector* (Algorithm 4, lines 2 and 3); in this case, the permutation functions p_a and p_b in Algorithm 2 simply return *pairs* of indices. This is illustrated in Figure 5. Effectively, in this example we are sorting the variable $x_{k,m}$ whose domain is $\mathrm{dom}(x_k) \times \mathrm{dom}(x_m)$, which has state space of size N^2.

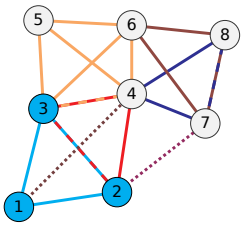
As the number of shared terms increases, so does the improvement to the running time. While (Equation 12) would take $\Theta(N^4)$ to solve using Algorithm 1, it takes only $O(N^3)$ to solve using Algorithm 4 (more precisely, if Algorithm 2 takes $O(f(N))$, then (Equation 12) takes $O(N^2 f(N^2))$, which we have mentioned is $O(N^2\sqrt{N^2}) = O(N^3)$). In general, if we have S shared terms, then the running time is $O(N^2\sqrt{N^S})$, yielding a speed-up of $\Omega(\sqrt{N^S})$ over the naïve solution of Algorithm 1.

5.2 An Extension to Higher-Order Cliques with Decompositions Into Three Groups

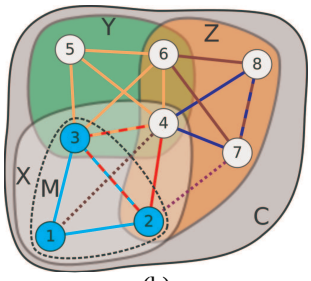
By similar reasoning, we can apply our algorithm to cases where there are more than three factors, in which the factors can be separated into three *groups*. For example, consider the clique in Figure 6(a), which we shall call G (the entire graph is a clique, but for clarity we only draw an edge when the corresponding nodes belong to a common factor). Each of the factors in this graph have been labeled using either differently colored edges (for factors of size larger than two) or dotted edges (for factors of size two), and the max-marginal we wish to compute has been labeled using colored nodes. We assume that it is possible to split this graph into three groups such that every factor is contained within a single group, along with the max-marginal we wish to compute (Figure 6, (b)). If such a decomposition is not possible, we will have to resort to further extensions to be described in Section 5.3.

Ideally, we would like these groups to have size $\simeq |G|/3$, though in the worst case they will have size no larger than $|G|-1$. We call these groups X, Y, Z, where X is the group containing the max-marginal M that we wish to compute. In order to simplify the analysis of this algorithm, we shall express the running time in terms of the size of the largest group, $S = \max(|X|,|Y|,|Z|)$, and the largest difference, $S_\setminus = \max(|Y \setminus X|,|Z \setminus X|)$. The max-marginal can be computed using Algorithm 5.

The running times shown in Algorithm 5 are loose upper-bounds, given for the sake of expressing the running time in simple terms. More precise running times are given in Table 2; any of the

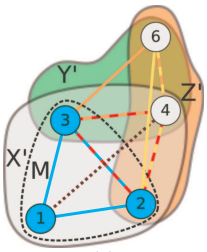


(a)

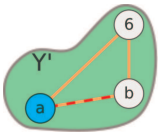


(b)

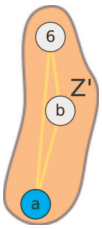
(a) We begin with a set of factors (indicated using colored lines), which are assumed to belong to some clique in our model; we wish to compute the max-marginal with respect to one of these factors (indicated using colored nodes); (b) The factors are split into three groups, such that every factor is entirely contained within one of them (Algorithm 5, line 1).



(c)

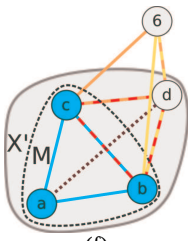


(d)

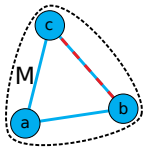


(e)

(c) Any nodes contained in only one of the groups are marginalized (Algorithm 5, lines 2, 3, and 4); the problem is now very similar to that described in Algorithm 4, except that *nodes* have been replaced by *groups*; note that this essentially introduces maximal factors in Y' and Z'; (d) For every value $(a,b) \in \text{dom}(x_3,x_4)$, $\Psi^Y(a,b,x_6)$ is sorted (Algorithm 5, lines 5–7); (e) For every value $(a,b) \in \text{dom}(x_2,x_4)$, $\Psi^Z(a,b,x_6)$ is sorted (Algorithm 5, lines 8–10).



(f)



(g)

(f) For every $\mathbf{n} \in \text{dom}(X')$, we choose the best value of x_6 by Algorithm 2 (Algorithm 5, lines 11–16); (g) The result is marginalized with respect to M (Algorithm 5, line 17).

Figure 6: Algorithm 5, explained pictorially. In this case, the most computationally intensive step is the marginalization of Z (in step (c)), which takes $\Theta(N^5)$. However, the algorithm can actually be applied *recursively* to the group Z, resulting in an overall running time of $O(N^4\sqrt{N})$, for a max-marginal that would have taken $\Theta(N^8)$ to compute using the naïve solution of Algorithm 1.

Algorithm 5 Compute the max-marginal of G with respect to M, where G is split into three groups

Input: potentials $\Phi_G(\mathbf{x}) = \Phi_X(\mathbf{x}_X) \times \Phi_Y(\mathbf{x}_Y) \times \Phi_Z(\mathbf{x}_Z)$; each of the factors should be contained in exactly one of these terms, and we assume that $M \subseteq X$ (see Figure 6)

1: **Define:** $X' := ((Y \cup Z) \cap X) \cup M$; $Y' := (X \cup Z) \cap Y$; $Z' := (X \cup Y) \cap Z$ {X' contains the variables in X that are shared by at least one other group; alternately, the variables in $X \setminus X'$ appear only in X (sim. for Y' and Z')}
2: compute $\Psi^X(\mathbf{x}_{X'}) := \max_{X \setminus X'} \Phi_X(\mathbf{x}_X)$ {we are marginalizing over those variables in X that do not appear in any of the other groups (or in M); this takes $\Theta(N^S)$ if done by brute-force (Algorithm 1), but may also be done by a recursive call to Algorithm 5}
3: compute $\Psi^Y(\mathbf{x}_{Y'}) := \max_{Y \setminus Y'} \Phi_Y(\mathbf{x}_Y)$
4: compute $\Psi^Z(\mathbf{x}_{Z'}) := \max_{Z \setminus Z'} \Phi_Z(\mathbf{x}_Z)$
5: **for** $\mathbf{n} \in \mathrm{dom}(X \cap Y)$ **do**
6: compute $\mathbf{P}^Y[\mathbf{n}]$ by sorting $\Psi^Y(\mathbf{n}; \mathbf{x}_{Y' \setminus X})$ {takes $\Theta(S_\setminus N^{S_\setminus} \log N)$; $\Psi^Y(\mathbf{n}; \mathbf{x}_{Y' \setminus X})$ is free over $\mathbf{x}_{Y' \setminus X}$, and is treated as an array by 'flattening' it; $\mathbf{P}^Y[\mathbf{n}]$ contains the $|Y' \setminus X| = |(Y \cap Z) \setminus X|$-dimensional indices that sort it}
7: **end for** {this loop takes $\Theta(S_\setminus N^S \log N)$}
8: **for** $\mathbf{n} \in \mathrm{dom}(X \cap Z)$ **do**
9: compute $\mathbf{P}^Z[\mathbf{n}]$ by sorting $\Psi^Z(\mathbf{n}; \mathbf{x}_{Z' \setminus X})$
10: **end for** {this loop takes $\Theta(S_\setminus N^S \log N)$}
11: **for** $\mathbf{n} \in \mathrm{dom}(X')$ **do**
12: $(\mathbf{v}_a, \mathbf{v}_b) := \left(\Psi^Y(\mathbf{n}|_{Y'}; \mathbf{x}_{Y' \setminus X'}), \Psi^Z(\mathbf{n}|_{Z'}; \mathbf{x}_{Z' \setminus X'})\right)$ {$\mathbf{n}|_{Y'}$ is the 'restriction' of the vector $\mathbf{n}$ to those indices in Y' (meaning that $\mathbf{n}|_{Y'} \in \mathrm{dom}(X' \cap Y')$); hence $\Psi^Y(\mathbf{n}|_{Y'}; \mathbf{x}_{Y' \setminus X'})$ is free in $\mathbf{x}_{Y' \setminus X'}$, while $\mathbf{n}|_{Y'}$ is fixed}
13: $(p_a, p_b) := \left(\mathbf{P}^Y[\mathbf{n}|_{Y'}], \mathbf{P}^Z[\mathbf{n}|_{Z'}]\right)$
14: $best := \mathit{Algorithm2}(\mathbf{v}_a, \mathbf{v}_b, p_a, p_b)$ {takes $O(\sqrt{S_\setminus})$}
15: $m_X(\mathbf{n}) := \Psi^X(\mathbf{n}) \times \Psi^Y(best; \mathbf{n}|_{Y'}) \times \Psi^Z(best; \mathbf{n}|_{Z'})$
16: **end for**
17: $m_M(\mathbf{x}_M) := \mathit{Algorithm1}(m_X, M)$ {i.e., we are using Algorithm 1 to marginalize $m_X(\mathbf{x}_X)$ with respect to M; this takes $\Theta(N^S)$}

terms shown in Table 2 may be dominant. Some example graphs, and their resulting running times are shown in Figure 7.

5.2.1 APPLYING ALGORITHM 5 RECURSIVELY

The marginalization steps of Algorithm 5 (lines 2, 3, and 4) may further decompose into smaller groups, in which case Algorithm 5 can be applied recursively. For instance, the graph in Figure 7(a) represents the marginalization step that is to be performed in Figure 6(c) (Algorithm 5, line 4). Since this marginalization step is the asymptotically dominant step in the algorithm, applying Algorithm 5 recursively lowers the asymptotic complexity.

Another straightforward example of applying recursion in Algorithm 5 is shown in Figure 8, in which a ring-structured model is marginalized with respect to two of its nodes. Doing so takes $O(MN^2\sqrt{N})$; in contrast, solving the same problem using the junction-tree algorithm (by triangulating the graph) would take $\Theta(MN^3)$. Loopy belief-propagation takes $\Theta(MN^2)$ per iteration, meaning

Description	lines	time
Marginalization of Φ_X, without recursion	2	$\Theta(N^{\lvert X\rvert})$
Marginalization of Φ_Y	3	$\Theta(N^{\lvert Y\rvert})$
Marginalization of Φ_Z	4	$\Theta(N^{\lvert Z\rvert})$
Sorting Φ_Y	5–7	$\Theta(\lvert Y'\setminus X\rvert N^{\lvert Y'\rvert}\log N)$
Sorting Φ_Z	8–10	$\Theta(\lvert Z'\setminus X\rvert N^{\lvert Z'\rvert}\log N)$
Running Algorithm 2 on the sorted values	11–16	$O(N^{\lvert X'\rvert}\sqrt{N^{\lvert(Y'\cap Z')\setminus X'\rvert}})$

Table 2: Detailed running time analysis of Algorithm 5; any of these terms may be asymptotically dominant

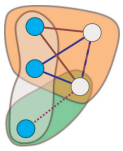
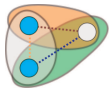
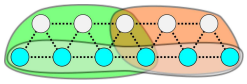
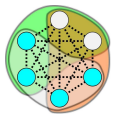
Graph:	 (a)	 (b)	 (c)	 (d)	{A complete graph K_M, with pairwise terms} (e)
Algorithm 1:	$\Theta(N^5)$	$\Theta(N^3)$	$\Theta(N^{11})$	$\Theta(N^6)$	$\Theta(N^M)$
Algorithm 5:	$O(N^3\sqrt{N})$	$O(N^2\sqrt{N})$	$O(N^6\sqrt{N})$	$O(N^5)$	$O(N^{5M/6})$
Speed-up:	$\Omega(N\sqrt{N})$	$\Omega(\sqrt{N})$	$\Omega(N^4\sqrt{N})$	$\Omega(N)$	$\Omega(N^{M/6})$

Figure 7: Some example graphs whose max-marginals are to be computed with respect to the colored nodes, using the three regions shown. Factors are indicated using differently colored edges, while dotted edges always indicate pairwise factors. (a) is the region Z from Figure 6 (recursion is applied *again* to achieve this result); (b) is the graph used to motivate Algorithm 4; (c) shows a query in a graph with regular structure; (d) shows a complete graph with six nodes; (e) generalizes this to a clique with M nodes.

that our algorithm will be faster if the number of iterations is $\Omega(\sqrt{N})$. Naturally, Algorithm 4 could be applied directly to the triangulated graph, which would again take $O(MN^2\sqrt{N})$.

5.3 A General Extension to Higher-Order Cliques

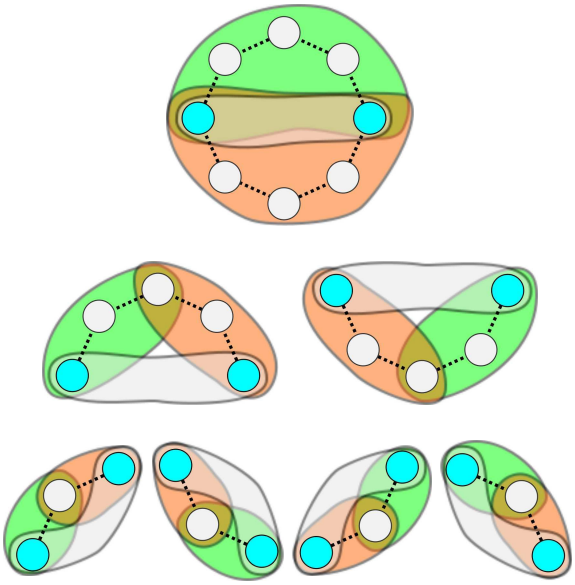
Naturally, there are cases for which a decomposition into three terms is not possible, such as

$$m_{i,j,k}(x_i,x_j,x_k) = \max_{x_m} \Phi_{i,j,k}(x_i,x_j,x_k) \times \Phi_{i,j,m}(x_i,x_j,x_m) \times \Phi_{i,k,m}(x_i,x_k,x_m) \times \Phi_{j,k,m}(x_j,x_k,x_m) \quad (13)$$

(i.e., a clique of size four with all possible third-order factors). However, if the model contains factors of size K, it must always be possible to split it into $K+1$ groups (e.g., four in the case of Equation 13).

Our optimizations can easily be applied in these cases simply by adapting Algorithm 2 to solve problems of the form

$$\max_{i\in\{1\ldots N\}} \{\mathbf{v}_1[i] \times \mathbf{v}_2[i] \times \cdots \times \mathbf{v}_K[i]\}. \quad (14)$$



$O(N^2)$

$+$

$O(2N^2\sqrt{N})$

$+$

$O(4N^2\sqrt{N})$ (by Algorithm 4)

Figure 8: In the above example, lines 2–4 of Algorithm 5 are applied recursively, achieving a total running time of $O(MN^2\sqrt{N})$ for a loop with M nodes (our algorithm achieves the same running time in the triangulated graph).

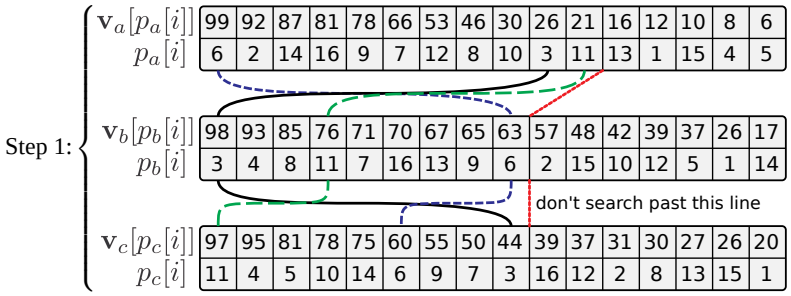


Figure 9: Algorithm 2 can easily be extended to cases including more than two sequences.

Pseudocode for this extension is presented in Algorithm 6. Note carefully the use of the variable *read*: we are storing which indices have been read to avoid re-reading them; this guarantees that our Algorithm is never asymptotically worse than the naïve solution. Figure 9 demonstrates how such an algorithm behaves in practice. Again, we shall discuss the running time of this extension in Appendix A. For the moment, we state the following theorem:

Theorem 3 *Algorithm 6 generalizes Algorithm 2 to K lists with an expected running time of* $O(KN^{\frac{K-1}{K}})$, *yielding a speed-up of at least* $\Omega(\frac{1}{K}N^{\frac{1}{K}})$ *in cliques containing K-ary factors. It is never worse than the naïve solution, meaning that it takes* $O(\min(N, KN^{\frac{K-1}{K}}))$.

Using Algorithm 6, we can similarly extend Algorithm 5 to allow for any number of *groups* (pseudocode is not shown; all statements about the groups Y and Z simply become statements

Algorithm 6 Find i such that $\prod_{k=1}^{K} \mathbf{v}_k[i]$ is maximized

Input: K vectors $\mathbf{v}_1 \dots \mathbf{v}_K$; permutation functions $p_1 \dots p_K$ that sort them in decreasing order; a vector *read* indicating which indices have been read, and a unique value $T \notin read$ {*read* is essentially a boolean array indicating which indices have been read; since *creating* this array is an $O(N)$ operation, we create it externally, and reuse it $O(N)$ times; setting $read[i] = T$ indicates that a particular index has been read; we use a different value of T for each call to this function so that *read* can be reused without having to be reinitialized}

Initialize: $start := 1$,
$max := \max_{p \in \{p_1 \dots p_K\}} \prod_{k=1}^{K} \mathbf{v}_k[p[1]]$,
$best := \text{argmax}_{p \in \{p_1 \dots p_K\}} \prod_{k=1}^{K} \mathbf{v}_k[p[1]]$
for $k \in \{1 \dots K\}$ **do**
  $end_k := \max_{q \in \{p_1 \dots p_K\}} p_k^{-1}[q[1]]$
  $read[p_k[1]] = T$
end for
while $start < \max\{end_1 \dots end_K\}$ **do**
  $start := start + 1$
  for $k \in \{1 \dots K\}$ **do**
    if $read[p_k[start]] := T$ **then**
      continue
    end if
    $read[p_k[start]] := T$
    $m := \prod_{x=1}^{K} \mathbf{v}_x[p_k[start]]$
    if $m > max$ **then**
      $best := p_k[start]$
      $max := m$
    end if
    $e_k := \max_{q \in \{p_1 \dots p_K\}} p_k^{-1}[q[start]]$
    $end_k := \min(e_k, end_k)$
  end for
end while {see Appendix A for running times}
Return: *best*

about K groups $\{G_1 \dots G_K\}$, and calls to Algorithm 2 become calls to Algorithm 6). The one remaining case that has not been considered is when the sequences $\mathbf{v}_1 \cdots \mathbf{v}_K$ are functions of different (but overlapping) variables; naïvely, we can create a new variable whose domain is the product space of all of the overlapping terms, and still achieve the performance improvement guaranteed by Theorem 3; in some cases, better results can again be obtained by applying recursion, as in Figure 7.

As a final comment we note that we have not provided an algorithm for choosing *how* to split the variables of a model into $(K+1)$-groups. We note even if we split the groups in a naïve way, we are guaranteed to get *at least* the performance improvement guaranteed by Theorem 3, though more 'intelligent' splits may further improve the performance.

Furthermore, in all of the applications we have studied, K is sufficiently small that it is inexpensive to consider all possible splits by brute-force.

5.4 Extensions for Conditionally Factorizable Models

Just as in Section 5.2, we can extend Algorithm 3 to factors of any size, so long as the purely latent cliques contain more latent variables than those cliques that depend upon the observation. The analysis for this type of model is almost exactly the same as that presented in Section 5.2, except that any terms consisting of purely latent variables are processed offline.

As we mentioned in 5.2, if a model contains (non-maximal) factors of size K, we will gain a speed-up of $\Omega(\frac{1}{K}N^{\frac{1}{K}})$. If in addition there is a factor (either maximal or non-maximal) consisting of purely latent variables, we can still obtain a speed-up of $\Omega(\frac{1}{K+1}N^{\frac{1}{K+1}})$, since this factor merely contributes an additional term to (Equation 14). Thus when our 'data-dependent' terms contain only a single latent variable (i.e., $K=1$), we gain a speed-up of $\Omega(\sqrt{N})$, as in Algorithm 3.

6. Performance Improvements in Existing Applications

Our results are immediately compatible with several applications that rely on inference in graphical models. As we have mentioned, our results apply to *any model whose cliques decompose into lower-order terms*.

Often, potentials are defined only on *nodes* and *edges* of a model. A D^{th}-order Markov model has a tree-width of D, despite often containing only pairwise relationships. Similarly 'skip-chain CRFs' (Sutton and McCallum, 2006; Galley, 2006), and junction-trees used in SLAM applications (Paskin, 2003) often contain only pairwise terms, and may have low tree-width under reasonable conditions. These are examples of *latently factorizable* models. In each case, if the tree-width is D, Algorithm 5 takes $O(MN^D\sqrt{N})$ (for a model with M nodes and N states per node), yielding a speed-up of $\Omega(\sqrt{N})$.

Models for shape-matching and pose reconstruction often exhibit similar properties (Tresadern et al., 2009; Donner et al., 2007; Sigal and Black, 2006). In each case, third-order cliques factorize into second-order terms; hence we can apply Algorithm 4 to achieve a speed-up of $\Omega(\sqrt{N})$.

Another similar model for shape-matching is that of Felzenszwalb (2005); this model again contains third-order cliques, though it includes a 'geometric' term constraining all three variables. Here, the third-order term is *independent of the input data*, meaning that each of its rows can be sorted *offline*, as described in Section 3. This is an example of a *conditionally factorizable* model. In this case, those factors that depend upon the observation are pairwise, meaning that we achieve a speed-up of $\Omega(N^{\frac{1}{3}})$. Further applications of this type shall be explored in Section 7.4.

In Coughlan and Ferreira (2002), deformable shape-matching is solved approximately using loopy belief-propagation. Their model has only second-order cliques, meaning that inference takes $\Theta(MN^2)$ *per iteration*. Although we cannot improve upon this result, we note that we can typically do *exact* inference in a single iteration in $O(MN^2\sqrt{N})$; thus our model has the same running time as $O(\sqrt{N})$ iterations of the original version. This result applies to all second-order models containing a single loop (Weiss, 2000).

In McAuley et al. (2008), a model is presented for graph-matching using loopy belief-propagation; the maximal cliques for D-dimensional matching have size $(D+1)$, meaning that inference takes $\Theta(MN^{D+1})$ *per iteration* (it is shown to converge to the correct solution); we improve this to $O(MN^D\sqrt{N})$.

Interval graphs can be used to model resource allocation problems (Fulkerson and Gross, 1965); each node encodes a request, and overlapping requests form edges. Maximal cliques grow with the

Reference	description	running time	our method
McAuley et al. (2008)	D-d graph-matching	$\Theta(MN^{D+1})$ (iter.)	$O(MN^D\sqrt{N})$ (iter.)
Sutton and McCallum (2006)	Width-D skip-chain	$O(MN^D)$	$O(MN^{D-1}\sqrt{N})$
Galley (2006)	Width-3 skip-chain	$\Theta(MN^3)$	$O(MN^2\sqrt{N})$
Tresadern et al. (2009)	Deformable matching	$\Theta(MN^3)$	$O(MN^2\sqrt{N})$
Coughlan and Ferreira (2002)	Deformable matching	$\Theta(MN^2)$ (iter.)	$O(MN^2\sqrt{N})$
Sigal and Black (2006)	Pose reconstruction	$\Theta(MN^3)$	$O(MN^2\sqrt{N})$
Felzenszwalb (2005)	Deformable matching	$\Theta(MN^3)$	$\Theta(MN^{\frac{8}{3}})$ (online)
Fulkerson and Gross (1965)	Width-D interval graph	$O(MN^{D+1})$	$O(MN^D\sqrt{N})$

Table 3: Some existing work to which our results can be immediately applied (M is the number of nodes in the model, N is the number of states per node. 'iter.' denotes that the algorithm is iterative).

number of overlapping requests, though the constraints are only pairwise, meaning that we again achieve an $\Omega(\sqrt{N})$ improvement.

Finally, in Section 7.4 we shall explore a variety of applications in which we have pairwise models of the form shown in (Equation 7). In all of these cases, we see an (expected) reduction of a $\Theta(MN^2)$ message-passing algorithm to $O(MN\sqrt{N})$.

Table 3 summarizes these results. Reported running times reflect the *expected case*. Note that we are assuming that *max-product belief-propagation is being used in a discrete model*; some of the referenced articles may use different variants of the algorithm (e.g., Gaussian models, or approximate inference schemes). We believe that our improvements may revive the exact, discrete version as a tractable option in these cases.

7. Experiments

We present experimental results for two types of models: latently factorizable models, whose cliques factorize into smaller terms, as discussed in Section 4, and conditionally factorizable models, whose factors *that depend upon the observation* contain fewer latent variables than their maximal cliques, as discussed in Section 3.

We begin with an asymptotic analysis of the running time of our algorithm on the 'inner product' operations of (Equation 1) and (Equation 14), in order to assess Theorems 2 and 3 experimentally.

7.1 Comparison Between Asymptotic Performance and Upper-Bounds

For our first experiment, we compare the performance of Algorithms 2 and 6 to the naïve solution of Algorithm 1. These are core subroutines of each of the other algorithms, meaning that determining their performance shall give us an accurate indication of the improvements we expect to obtain in real graphical models.

For each experiment, we generate N i.i.d. samples from $[0,1)$ to obtain the lists $v_1 \dots v_K$. N is the domain size; this may refer to a single node, or a *group* of nodes as in Algorithm 6; thus large values of N may appear even for binary-valued models. K is the number of lists in (Equation 14); we can observe this number of lists only if we are working in cliques of size $K+1$, and then only if the factors are of size K (e.g., we will only see $K=5$ if we have cliques of size 6 with factors

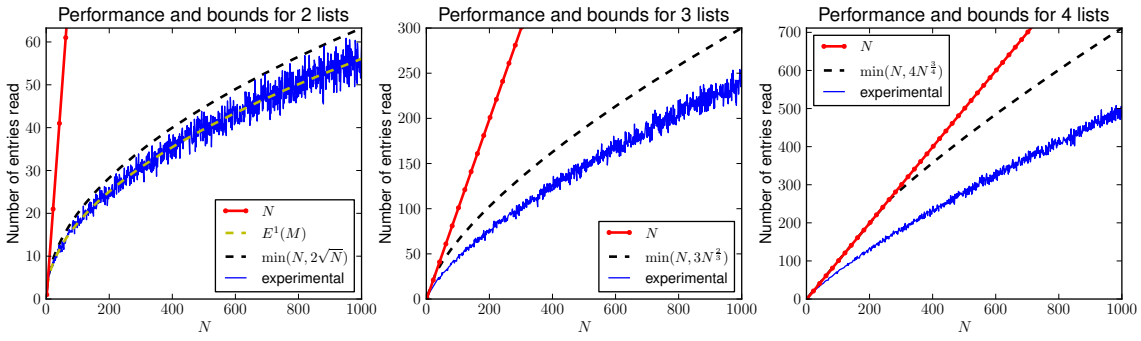


Figure 10: Performance of our algorithm and bounds. For $K = 2$, the exact expectation is shown, which appears to precisely match the average performance (over 100 trials). The dotted lines show the bound of (Equation 23). While the bound is close to the true performance for $K = 2$, it becomes increasingly loose for larger K.

of size 5); therefore smaller values of K are probably more realistic in practice (indeed, all of the applications in Section 6 have $K = 2$).

The performance of our algorithm is shown in Figure 10, for $K = 2$ to 4 (i.e., for 2 to 4 lists). When $K = 2$, we execute Algorithm 2, while Algorithm 6 is executed for $K \geq 3$. The performance reported is simply the number of elements read from the lists (which is at most $K \times start$). This is compared to N itself, which is the number of elements read by the naïve algorithm. The upper-bounds we obtained in (Equation 23) are also reported, while the true expected performance (i.e., Equation 19) is reported for $K = 2$. Note that the variable *read* was introduced into Algorithm 6 in order to guarantee that it can never be asymptotically slower than the naïve algorithm. If this variable is ignored, the performance of our algorithm deteriorates to the point that it closely approaches the upper-bounds shown in Figure 10. Unfortunately, this optimization proved overly complicated to include in our analysis, meaning that our upper-bounds remain highly conservative for large K.

7.2 Performance Improvement for Dependent Variables

The expected-case running time of our algorithm was derived under the assumption that each list has independent order statistics, as was the case for our previous experiment. We suggested that we will obtain worse performance in the case of negatively correlated variables, and better performance in the case of positively correlated variables; we shall assess these claims in this experiment.

Figure 11 shows how the order statistics of $\mathbf{v}_a$ and $\mathbf{v}_b$ can affect the performance of our algorithm. Essentially, the running time of Algorithm 2 is determined by the level of 'diagonalness' of the permutation matrices in Figure 11; highly diagonal matrices result in better performance than the expected case, while highly off-diagonal matrices result in worse performance. The expected case was simply obtained under the assumption that every permutation is equally likely.

We report the performance for two lists (i.e., for Algorithm 2), where each $(\mathbf{v}_a[i], \mathbf{v}_b[i])$ is an independent sample from a 2-dimensional Gaussian with covariance matrix

$$\Sigma = \begin{bmatrix} 1 & c \\ c & 1 \end{bmatrix},$$

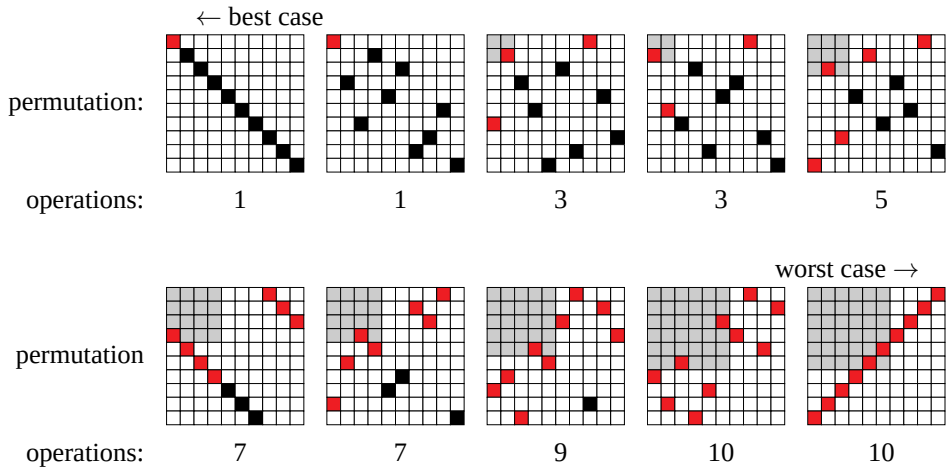


Figure 11: Different permutation matrices and their resulting cost (in terms of entries read/multiplications performed). Each permutation matrix transforms the *sorted* values of one list into the sorted values of the other, that is, it transforms $\mathbf{v}_a$ as sorted by p_a into $\mathbf{v}_b$ as sorted by p_b. The red (lighter) squares show the entries that must be read before the algorithm terminates (each corresponding to one multiplication). See Figure 23 for further explanation.

meaning that the two lists are correlated with correlation coefficient c (here we are working in the max-sum semiring). This dependence between the values of the two lists leads to a dependence in their order statistics, so that in the case of Gaussian random variables, the correlation coefficient precisely captures the 'diagonalness' of the matrices in Figure 11. Performance is shown in Figure 12 for different values of c ($c = 0$, is not shown, as this is the case observed in the previous experiment).

7.3 Message-Passing in Latently Factorizable Models

In this section we present experiments in models whose cliques factorize into smaller terms, as discussed in Section 4.

7.3.1 2-DIMENSIONAL GRAPH-MATCHING

Naturally, Algorithm 5 has additional overhead compared to the naïve solution, meaning that it will not be beneficial for small N. In this experiment, we aim to assess the extent to which our approach is faster in real applications. We reproduce the model from McAuley et al. (2008), which performs 2-dimensional graph-matching, using a loopy graph with cliques of size three, containing only second-order potentials (as described in Section 6); the $\Theta(NM^3)$ performance of McAuley et al. (2008) is reportedly state-of-the-art. We also show the performance on a graphical model with *random* potentials, in order to assess how the results of the previous experiments are reflected in terms of actual running time.

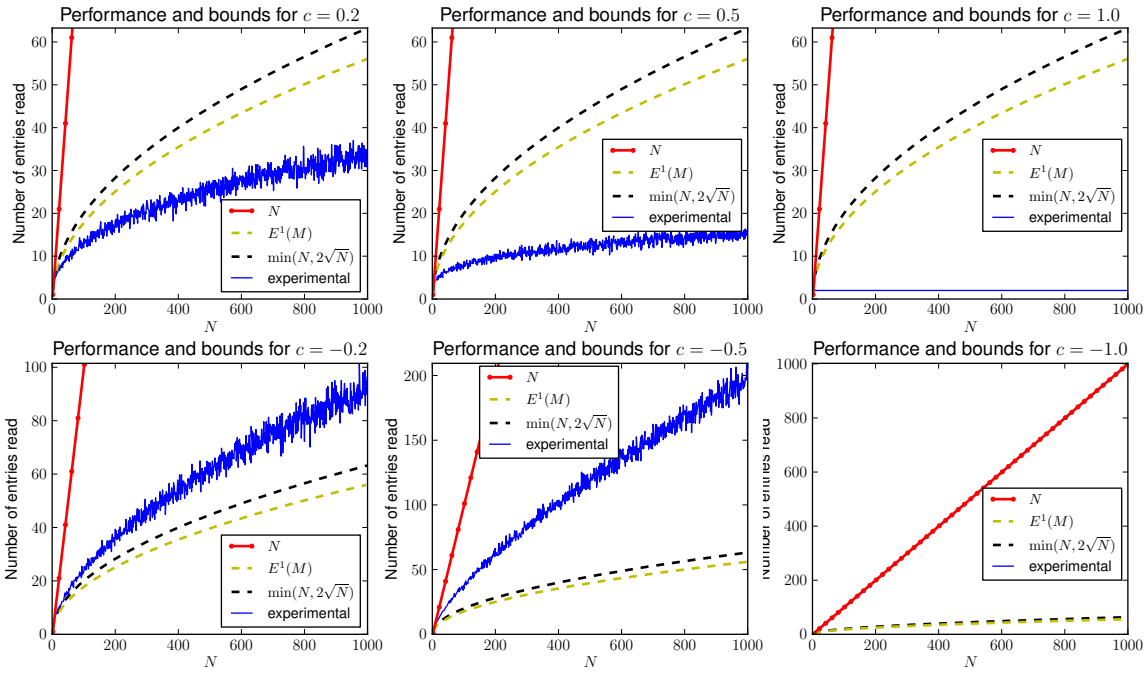


Figure 12: Performance of our algorithm for different correlation coefficients. The top three plots show positive correlation, the bottom three show negative correlation. Correlation coefficients of $c = 1.0$ and $c = -1.0$ capture precisely the best and worst-case performance of our algorithm, resulting in $O(1)$ and $\Theta(N)$ performance, respectively (when $c = -1.0$ the linear curve obscures the experimental curve).

We perform matching between a *template* graph with M nodes, and a *target* graph with N nodes, which requires a graphical model with M nodes and N states per node (see McAuley et al. 2008 for details). We fix $M = 10$ and vary N.

Figure 13 (left) shows the performance on random potentials, that is, the performance we hope to obtain if our model assumptions are satisfied. Figure 13 (right) shows the performance for graph-matching, which closely matches the expected-case behavior. Fitted curves are shown together with the actual running time of our algorithm, confirming its $O(MN^2\sqrt{N})$ performance. The coefficients of the fitted curves demonstrate that our algorithm is useful even for modest values of N.

We also report results for graph-matching using graphs from the MPEG-7 data set (Bai et al., 2009), which consists of 1,400 silhouette images (Figure 14). Again we fix $M = 10$ (i.e., 10 points are extracted in each template graph) and vary N (the number of points in the target graph). This experiment confirms that even when matching real-world graphs, the assumption of independent order statistics appears to be reasonable.

7.3.2 HIGHER-ORDER MARKOV MODELS

In this experiment, we construct a simple Markov model for text denoising. Random noise is applied to a text segment, which we try to correct using a prior extracted from a text corpus. For instance

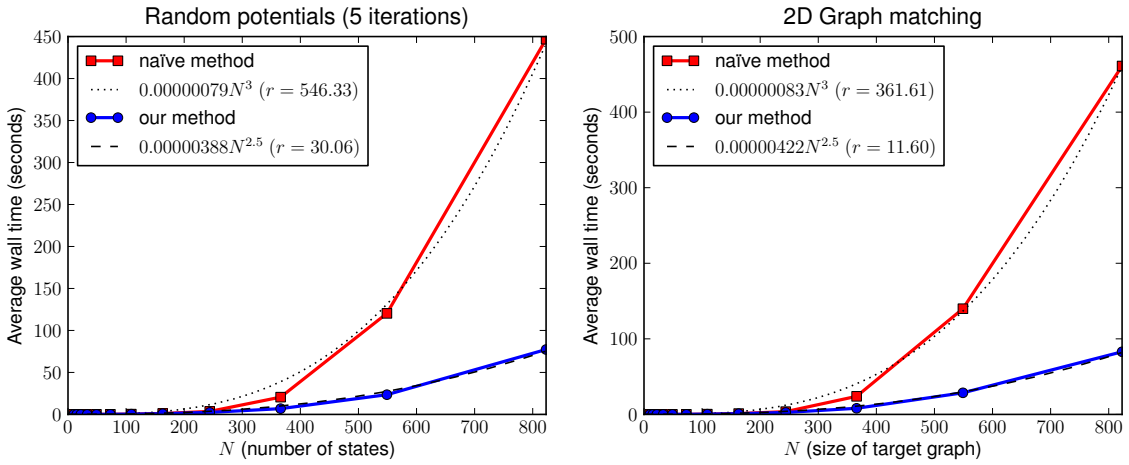


Figure 13: The running time of our method on randomly generated potentials, and on a graph-matching experiment (both graphs have the same topology). Fitted curves are also obtained by performing least-squares regression; the residual error *r* indicates the 'goodness' of the fitted curve.

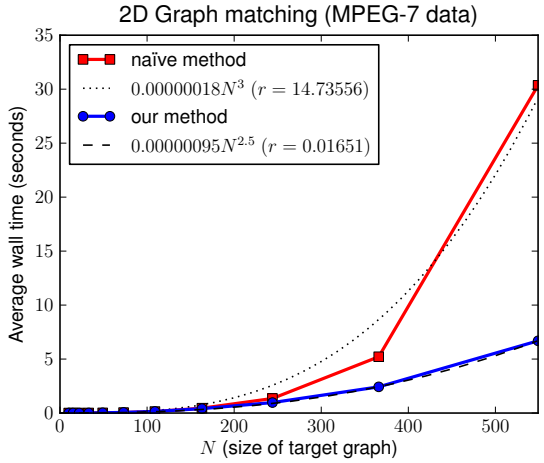


Figure 14: The running time of method our on graphs from the MPEG-7 data set.

`wondrous sight of th4 ivory Pequod` is corrected to `wondrous sight of the ivory Pequod`.

In such a model, we would like to exploit higher-order relationships between characters, though the amount of data required to construct an accurate prior grows exponentially with the size of the maximal cliques. Instead, our prior consists entirely of pairwise relationships between characters (or 'bigrams'); higher-order relationships are encoded by including bigrams of non-adjacent characters.

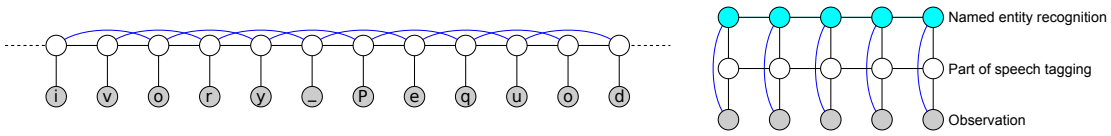


Figure 15: Left: Our model for denoising. Its computational complexity is similar to that of a skip-chain CRF, and models for named-entity recognition (right).

Specifically, our model takes the form

$$\Phi_X(\mathbf{x}_X) = \prod_{i=1}^{|X|-1} \Phi_{i,i+1}(x_i, x_{i+1}) \times \prod_{i=1}^{|X|-2} \Phi_{i,i+2}(x_i, x_{i+2})$$

where

$$\Phi_{i,j}(x_i, x_j) = \psi_{i,j}(x_i, x_j) p(x_i|o_i) p(x_j|o_j).$$

Here ψ is our *prior* (extracted from text statistics), and p is our 'noise model' (given the observation $\mathbf{o}$). The computational complexity of inference in this model is similar to that of the skip-chain CRF shown in Figure 3(b), as well as models for part-of-speech tagging and named-entity recognition, as in Figure 15. Text denoising is useful for the purpose of demonstrating our algorithm, as there are several different corpora available in different languages, allowing us to explore the effect that the domain size (i.e., the size of the language's alphabet) has on running time.

We extracted pairwise statistics based on 10,000 characters of text, and used this to correct a series of 25 character sequences, with 1% random noise introduced to the text. The domain was simply the set of characters observed in each corpus. The Japanese data set was not included, as the $\Theta(MN^2)$ memory requirements of the algorithm made it infeasible with $N \simeq 2000$; this is addressed in Section 7.4.1.

The running time of our method, compared to the naïve solution, is shown in Figure 16. One might expect that texts from different languages would exhibit different dependence structures in their order statistics, and therefore deviate from the expected case in some instances. However, the running times appear to follow the fitted curve closely, that is, we are achieving approximately the expected-case performance in all cases.

Since the prior $\psi_{i,i+1}(x_i, x_{i+1})$ is *data-independent*, we shall further discuss this type of model in reference to Algorithm 3 in Section 7.4.

7.4 Experiments with Conditionally Factorizable Models

In each of the following experiments we perform belief-propagation in models of the form given in (Equation 7). Thus each model is completely specified by defining the node potentials $\Phi_i(x_i|y_i)$, the edge potentials $\Phi_{i,j}(x_i, x_j)$, and the topology $(\mathcal{N}, \mathcal{E})$ of the graph.

Furthermore we assume that the edge potentials are *homogeneous*, that is, that the potential for each edge is the same, or rather that they have the same order statistics (for example, they may differ by a multiplicative constant). This means that sorting can be done *online* without affecting the asymptotic complexity. When subject to heterogeneous potentials we need merely sort them *offline*; the online cost shall be similar to what we report here.

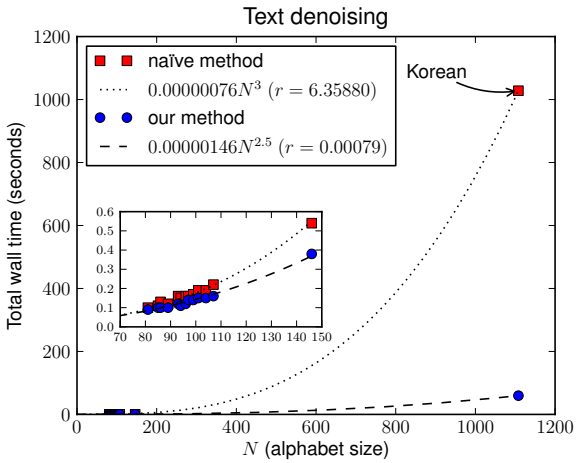


Figure 16: The running time of our method compared to the naïve solution. A fitted curve is also shown, whose coefficient estimates the computational overhead of our model.

7.4.1 CHAIN-STRUCTURED MODELS

In this section, we consider *chain-structured* graphs. Here we have nodes $\mathcal{N} = \{1 \dots Q\}$, and edges $\mathcal{E} = \{(1,2),(2,3) \dots (Q-1,Q)\}$. The max-product algorithm is known to compute the maximum-likelihood solution exactly for tree-structured models.

Figure 17 (left) shows the performance of our method on a model with *random* potentials, that is, $\Phi_i(x_i|y_i) = U[0,1)$, $\Phi_{i,i+1}(x_i,x_{i+1}) = U[0,1)$, where $U[0,1)$ is the uniform distribution. Fitted curves are superimposed onto the running time, confirming that the performance of the standard solution grows quadratically with the number of states, while ours grows at a rate of $N\sqrt{N}$. The residual error r shows how closely the fitted curve approximates the running time; in the case of random potentials, both curves have almost the same constant.

Figure 17 (right) shows the performance of our method on the text denoising experiment. This experiment is essentially identical to that shown in Section 7.3.2, except that the model is a chain (i.e., there is no $\Phi_{i,i+2}$), and we exploit the notion of data-independence (i.e., the fact that $\Phi_{i,i+1}$ does not depend on the observation). Since the same $\Phi_{i,i+1}$ is used for every adjacent pair of nodes, there is no need to perform the 'sorting' step offline—only a single copy of $\Phi_{i,i+1}$ needs to be sorted, and this is included in the total running time shown in Figure 17.

7.4.2 GRID-STRUCTURED MODELS

Similarly, we can apply our method to *grid-structured* models. Here we resort to loopy belief-propagation to approximate the MAP solution, though indeed the same analysis applies in the case of factor-graphs (Kschischang et al., 2001). We construct a 50×50 grid model and perform loopy belief-propagation using a random message-passing schedule for five iterations. In these experiments our nodes are $\mathcal{N} = \{1 \dots 50\}^2$, and our edges connect the 4-neighbors, that is, the node (i,j) is connected to both $(i+1,j)$ and $(i,j+1)$ (similar to the grid shown in Figure 2(a)).

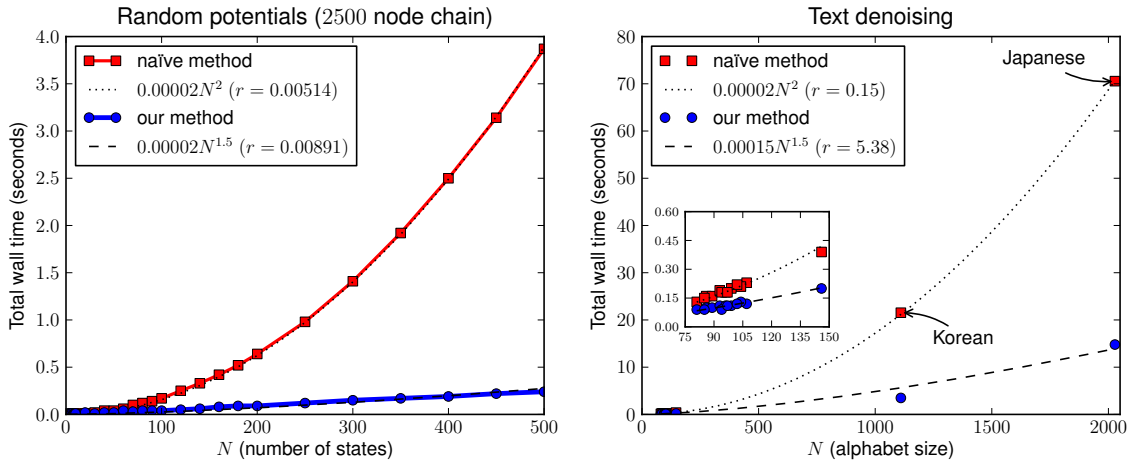


Figure 17: Running time of inference in chain-structured models: random potentials (left), and text denoising (right). Fitted curves confirm that the exponent of 1.5 given theoretically is maintained in practice (r denotes the sum of residuals, that is, the 'goodness' of the fitted curve).

Figure 18 (left) shows the performance of our method on a grid with random potentials (similar to the experiment in Section 7.4.1). Figure 18 (right) shows the performance of our method on an optical flow task (Lucas and Kanade, 1981). Here the states encode *flow vectors*: for a node with N states, the flow vector is assumed to take integer coordinates in the square $[-\sqrt{N}/2, \sqrt{N}/2)^2$ (so that there are N possible flow vectors). For the unary potential we have

$$\Phi_{(i,j)}(x|y) = \left\| Im_1[i,j] - Im_2[(i,j) + f(x)] \right\|,$$

where $Im_1[a,b]$ and $Im_2[a,b]$ return the gray-level of the pixel at (a,b) in the first and second images (respectively), and $f(x)$ returns the flow vector encoded by x. The pairwise potentials simply encode the Euclidean distance between two flow vectors. Note that a variety of low-level computer vision tasks (including optical flow) are studied in Felzenszwalb and Huttenlocher (2006), where the highly structured nature of the potentials in question often allows for efficient solutions.

Our fitted curves in Figure 18 show $O(N\sqrt{N})$ performance for both random data and for optical flow. Clearly the fitted curve for optical flow deviates somewhat from that obtained for random data; naturally the potentials are highly structured in this case, as exploited by Felzenszwalb and Huttenlocher (2006); it appears that some aspect of this structure is slightly harmful to our algorithm, though a more thorough analysis of this type of potential remains as future work. More 'harmful' structures are explored in the following section.

7.4.3 FAILURE CASES

In our previous experiments on graph-matching, text denoising, and optical flow we observed running times similar to those for random potentials, indicating that there is no prevalent dependence structure between the order statistics of the messages and the potentials.

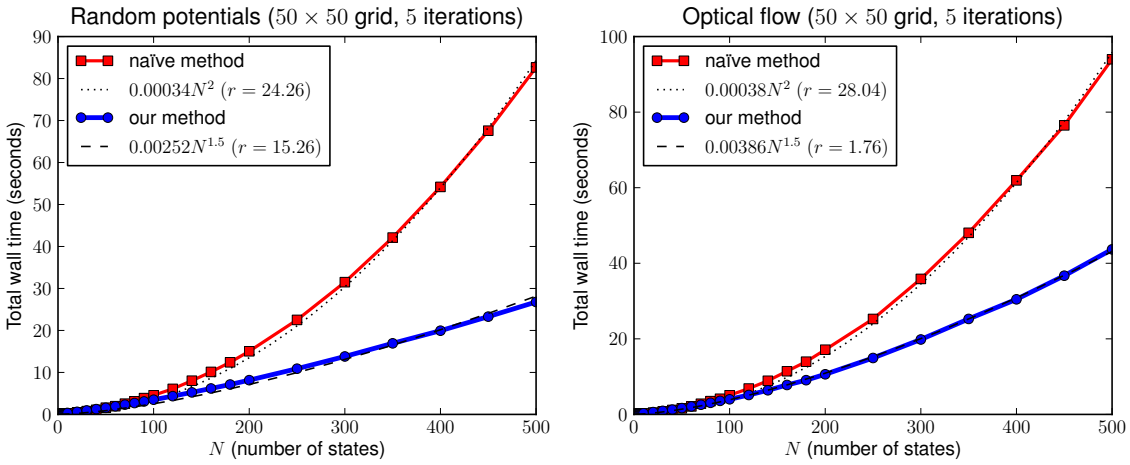


Figure 18: Running time of inference in grid-structured models: random potentials (left), and optical flow (right).

In certain applications the order statistics of these terms are highly dependent in a way that is detrimental to our algorithm. This behavior is observed for certain types of concave potentials (or convex potentials in a min-sum formulation). For instance, in a stereo disparity experiment, the unary potentials encode the fact that the output should be 'close to' a certain value; the pairwise potentials encode the fact that neighboring nodes should take similar values (Scharstein and Szeliski, 2001; Sun et al., 2003).

In these applications, the permutation matrices that transform the sorted values of $\mathbf{v}_a$ to the sorted values of $\mathbf{v}_b$ are block-off-diagonal (see the sixth permutation in Figure 11). In such cases, our algorithm only decreases the number of multiplication operations by a multiplicative constant, and may in fact be slower due to its computational overhead. This is precisely the behavior shown in Figure 19 (left), in the case of stereo disparity.

It should be noted that there exist algorithms specifically designed for this class of potential functions (Kolmogorov and Shioura, 2007; Felzenszwalb and Huttenlocher, 2006), which are preferable in such instances.

We similarly perform an experiment on image denoising, where the unary potentials are again convex functions of the input (see Geman and Geman, 1984; Lan et al., 2006). Instead of using a pairwise potential that merely encodes smoothness, we extract the pairwise statistics from image data (similar to our experiment on text denoising); thus the potentials are no longer concave. We see in Figure 19 (right) that even if a small number of entries exhibit some 'randomness' in their order statistics, we begin to gain a modest speed improvement over the naïve solution (though indeed, the improvements are negligible compared to those shown in previous experiments).

7.5 Other Applications of Tropical Matrix Multiplication

As we have mentioned, our improvements to message-passing in graphical models arise from a fast solution to matrix multiplication in the max-product semiring. In this section we discuss other

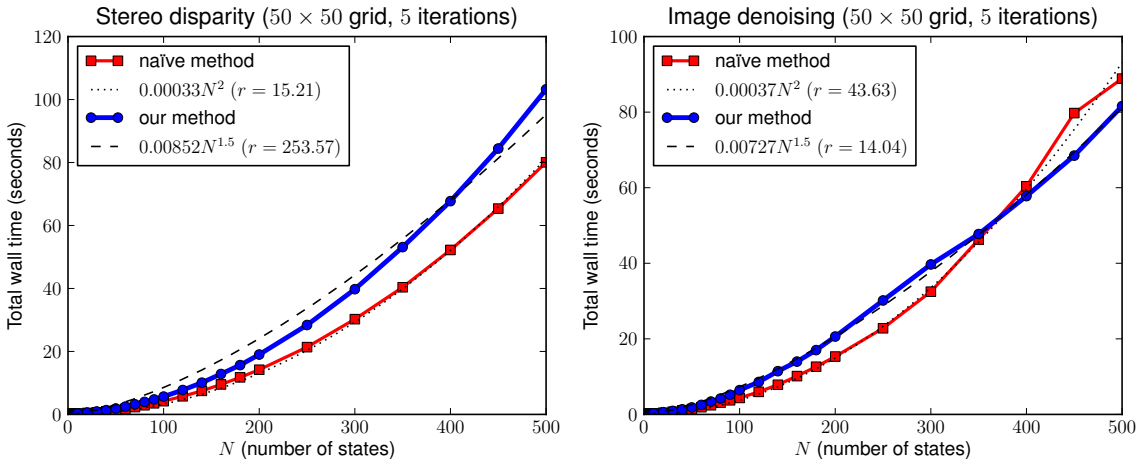


Figure 19: Two experiments whose potentials and messages have highly dependent order statistics: stereo disparity (left), and image denoising (right).

problems which include max-product (or 'tropical') matrix multiplication as a subroutine. Williams and Williams (2010) discusses the relationship between this type of matrix multiplication problem and various other problems.

7.5.1 MAX-PRODUCT LINEAR PROGRAMMING

In Sontag et al. (2008), a method is given for exact MAP-inference in graphical models using LP-relaxations. Where exact solutions cannot be obtained by considering only pairwise factors, 'clusters' of pairwise terms are introduced in order to refine the solution. Message-passing in these clusters turns out to take exactly the form that we consider, as third-order (or larger) clusters are formed from pairwise terms. Although a number of applications are presented in Sontag et al. (2008), we focus on protein design, as this is the application in which we typically observe the largest domain sizes. Other applications with larger domains may yield further benefits.

Without going into detail, we simply copy the two equations from Sontag et al. (2008) to which our algorithm applies. The first of these is concerned with passing messages between clusters, while the second is concerned with choosing new clusters to add. Below are the two equations, reproduced verbatim from Sontag et al. (2008):

$$\lambda_{c\to e}(x_e) \leftarrow -\frac{2}{3}\Big(\lambda_{e\to e}(x_e) + \sum_{c'\neq c, e\in c'} \lambda_{c'\to e}(x_e)\Big) + \frac{1}{3}\max_{x_{c\setminus e}}\Big[\sum_{e'\in c\setminus e}\Big(\lambda_{e'\to e'}(x_{e'}) + \sum_{c'\neq c, e'\in c'} \lambda_{c'\to e'}(x_{e'})\Big)\Big] \tag{15}$$

(see Sontag et al., 2008, Figure 1, bottom), which consists of marginalizing a cluster (c) that decomposes into edges (e), and

$$d(c) = \sum_{e\in c}\max_{x_e} b_e(x_e) - \max_{x_c}\left[\sum_{e\in c} b_e(x_e)\right], \tag{16}$$

(see Sontag et al., 2008, (Equation 4)), which consists of finding the MAP-state in a ring-structured model.

As the code from Sontag et al. (2008) was publicly available, we simply replaced the appropriate functions with our own (in order to provide a fair comparison, we also replaced their implementation of the naïve algorithm, as ours proved to be faster than the highly generic matrix library used in their code).

In order to improve the running time of our algorithm, we made the following two modifications to Algorithm 2:

- We used an *adaptive sorting algorithm* (i.e., a sorting algorithm that runs faster on nearly-sorted data). While Quicksort was used during the first iteration of message-passing, subsequent iterations used insertion sort, as the optimal ordering did not change significantly between iterations.

- We added an additional stopping criterion to the algorithm. Namely, we terminate the algorithm if $\mathbf{v}_a[p_a[start]] \times \mathbf{v}_b[p_b[start]] < max$. In other words, we check how large the maximum *could be* given the best possible permutation of the next elements (i.e., if they have the same index); if this value could not result in a new maximum, the algorithm terminates. This check costs us an additional multiplication, but it means that the algorithm will terminate faster in cases where a large maximum is found early on.

Results for these two problems are shown in Figure 20. Although our algorithm consistently improves upon the running time of Sontag et al. (2008), the domain size of the variables in question is not typically large enough to see a marked improvement. Interestingly, neither method follows the expected running time closely in this experiment. This is partly due to the fact that there is significant variation in the variable size (note that N only shows the *average* variable size), but it may also suggest that there is a complicated structure in the potentials which violates our assumption of independent order statistics.

7.5.2 ALL-PAIRS SHORTEST-PATH

The 'all-pairs shortest-path' problem consists of finding the shortest path between every pair of nodes in a graph. Although the most commonly used solution is probably the well-known Floyd-Warshall algorithm (Floyd, 1962), the state-of-the-art *expected-case* solution to this problem is that of Karger et al. (1993), whose expected-case running time is $O(N^2 \log N)$ when applied to graphs with distances sampled from the uniform distribution.

Unfortunately, the solution of Karger et al. (1993) requires a Fibonacci heap or similar data structure in order to achieve the reported running time (i.e., a heap with $O(1)$ insertion and decrease-key operations); such data structures are known to be inefficient in practice (Fredman and Tarjan, 1987). When their algorithm is implemented using a standard priority queue, it has running time $O(N^2 \log^2 N)$.

In Aho et al. (1983), a transformation is shown between the all-pairs shortest-path problem and min-sum matrix multiplication. Using our algorithm, this gives us an expected-case $O(N^2\sqrt{N})$ solution to the all-pairs shortest-path problem, assuming that the subproblems created by this transformation have i.i.d. order statistics; this assumption is notably different than the assumption of uniformity made in Karger et al. (1993).

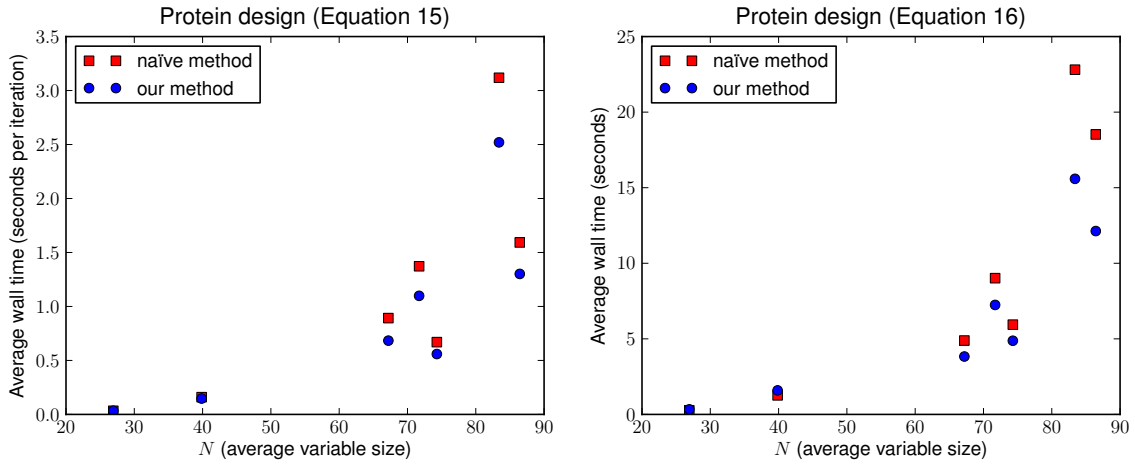


Figure 20: The running time of our method on protein design problems from Sontag et al. (2008). In this figure, N reflects the *average* domain size amongst all variables involved in the problem; fitted curves are not shown due to the highly variable nature of the domain sizes included in each problem instance.

In Figure 21, we show the performance of our method on i.i.d. uniform graphs, compared to the Floyd-Warshall algorithm, and that of Karger et al. (1993). On graph sizes of practical interest, our algorithm is found to give the fastest performance, in spite of its more expensive asymptotic cost. Our solution is comparable to that of Karger et al. (1993) for the largest graph size shown; larger graph sizes could not be shown due to memory constraints. Note that while these algorithms are fast in practice, each has $\Theta(N^3)$ *worst-case* performance; more 'exotic' solutions that improve upon the worst-case bound are discussed in Alon et al. (1997) and Chan (2007), among others, though none are truly subcubic (i.e., $O(N^{3-\varepsilon})$).

It should also be noted that the transformations given in Aho et al. (1983) apply in both directions, that is, solutions to the all-pairs shortest-path problem can be used to solve min-sum matrix multiplication. Thus any subcubic solution to the all-pairs shortest-path problem can be applied to the inference problems in graphical models presented in Section 4. However, the transformation of Aho et al. (1983) introduces a very high computational overhead (namely, solving min-sum matrix multiplication for an $N \times N$ matrix requires solving all-pairs shortest-path in a graph with $3N$ nodes), and moreover it violates the assumptions on the graph distribution required for fast inference given in Karger et al. (1993). In practice, we were unable to produce an implementation of min-sum matrix multiplication based on this transformation that was faster than the naïve solution.

Interestingly, a great deal of attention has been focused on expected-case solutions to all-pairs shortest-path, while to our knowledge ours is the first work to approach the expected-case analysis of min-sum matrix multiplication. Given the strong relationship between the two problems, it remains a promising open problem to assess whether the analysis from these solutions to all-pairs shortest-path can be applied to produce max-product matrix multiplication algorithms with similar asymptotic running times.

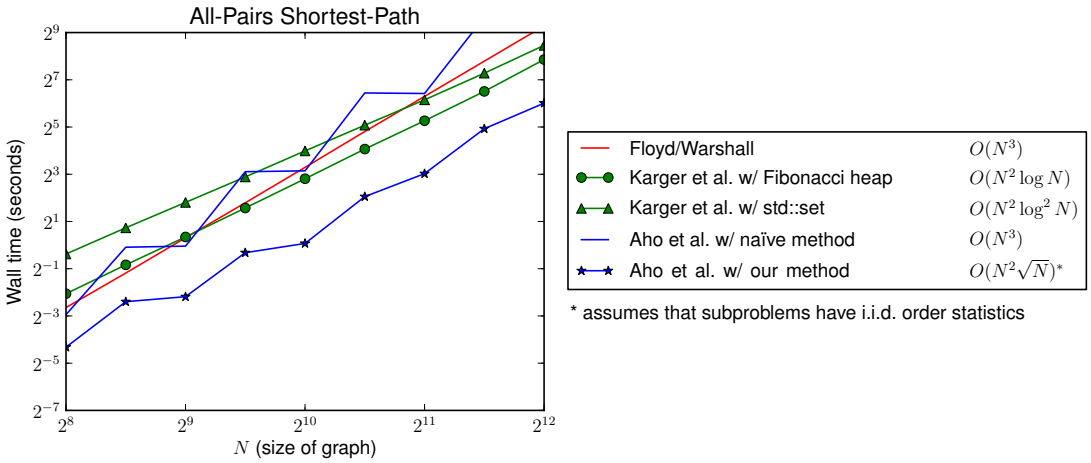


Figure 21: Our algorithm applied to the 'all-pairs shortest-path' problem. The expected-case running times of each algorithm are shown at right.

7.5.3 L^∞ DISTANCES

The problem of computing an inner product in the max-sum semiring is closely related to computing the L^∞ distance between two vectors

$$||\mathbf{v}_a - \mathbf{v}_b||_\infty = \max_{i \in \{1...N\}} \left|\mathbf{v}_a[i] - \mathbf{v}_b[i]\right|. \tag{17}$$

Naïvely, we would like to solve (Equation 17) by applying Algorithm 2 to $\mathbf{v}_a$ and $-\mathbf{v}_b$ with the multiplication operator replaced by $a \times b = |a+b|$, however this violates the condition of (Equation 2), since the optimal solution may arise either when both $\mathbf{v}_a[i]$ and $-\mathbf{v}_b[i]$ are large, or when both $\mathbf{v}_a[i]$ and $-\mathbf{v}_b[i]$ are small (in fact, this operation violates the semiring axiom of associativity).

We address this issue by running Algorithm 2 *twice*, first considering the *largest* values of $\mathbf{v}_a$ and $-\mathbf{v}_b$, before re-running the algorithm starting from the *smallest* values. This ensures that the maximum solution is found in either case.

Pseudocode for this solution is given in Algorithm 7, which adapts Algorithm 4 to the problem of computing an L^∞ distance matrix. Similarly, we can adapt Algorithm 3 to solve L^∞ nearest-neighbor problems, where an array of M points in $\mathbb{R}^N$ is processed offline, allowing us compute the distance of a query point to all M other points $O(M\sqrt{N})$.

Figure 22 shows the running time of our algorithm for computing an L^∞ distance matrix (where $M = N$), and the online cost of performing a nearest-neighbor query. Again the expected speedup over the naïve solution is $\Omega(\sqrt{N})$ for both problems, though naturally our algorithm requires larger values of N than does Algorithm 4 in order to be beneficial, since Algorithm 2 must be executed *twice* in order to solve (Equation 17).

A similar trick can be applied to compute message in the max-product semiring even for potentials that contain negative values, though this may require up to four executions of Algorithm 2, so it is unlikely to be practical.

Algorithm 7 Use Algorithm 2 to compute an L^∞ distance matrix

Input: an $M \times N$ array $\mathbf{A}$ containing M points in $\mathbb{R}^N$

initialize an $M \times M$ distance matrix $\mathbf{D} := \mathbf{0}$
for $x \in \{1 \ldots M\}$ **do**
  compute $\overrightarrow{\mathbf{P}}[x]$ by sorting $\mathbf{A}[x]$ {takes $\Theta N \log N$}
  compute $\overleftarrow{\mathbf{P}}[x]$ by sorting $-\mathbf{A}[x]$ {i.e., $\overrightarrow{\mathbf{P}}[x]$ in reverse order}
end for {this loop takes $\Theta(MN \log N)$}
for $x \in \{1 \ldots M\}$ **do**
  for $y \in \{x+1 \ldots M\}$ **do**
    $best_1 := Algorithm2\left(\mathbf{A}[x], -\mathbf{A}[y], \overrightarrow{\mathbf{P}}[x], \overleftarrow{\mathbf{P}}[y]\right)$
    {takes $O(\sqrt{N})$; Algorithm 2 uses the operator $a \times b = |a+b|$}
    $best_2 := Algorithm2\left(\mathbf{A}[y], -\mathbf{A}[x], \overrightarrow{\mathbf{P}}[y], \overleftarrow{\mathbf{P}}[x]\right)$
    $\mathbf{D}[x,y] := \max\left(\left|\mathbf{A}[x, best_1] - \mathbf{A}[y, best_1]\right|, \left|\mathbf{A}[x, best_2] - \mathbf{A}[y, best_2]\right|\right)$
    $\mathbf{D}[y,x] := \mathbf{D}[x,y]$
  end for
end for {this loop takes expected time $O(M^2\sqrt{N})$}

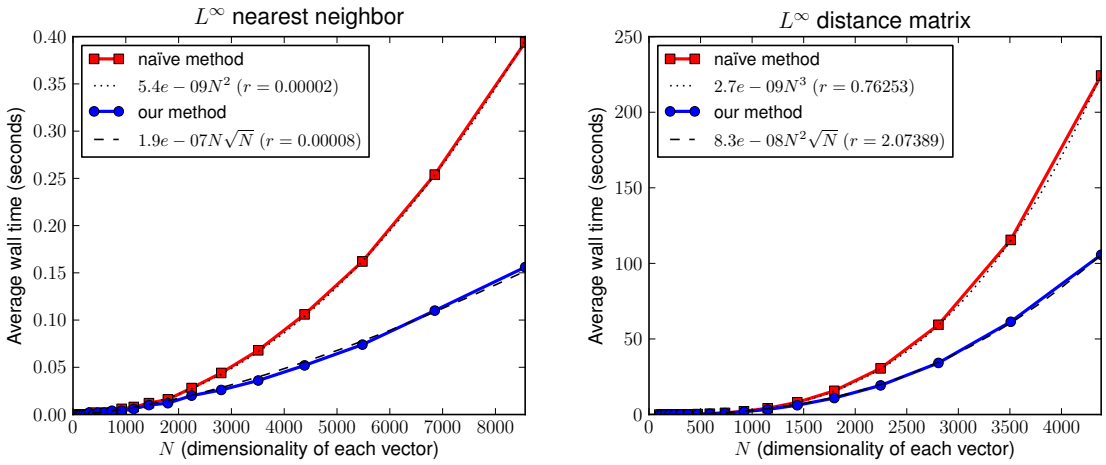


Figure 22: The running time of our method compared to the naïve solution. A fitted curve is also shown, whose coefficient estimates the computational overhead of our model.

8. Discussion and Future Work

We have briefly discussed the application of our algorithm to the all-pairs shortest-path problem, and also mentioned that a variety of other problems are related to max-product matrix multiplication via a series of subcubic transformations (Williams and Williams, 2010). To our knowledge, of all these problems only all-pairs shortest-paths has received significant attention in terms of expected-case analysis. The analysis in question centers around two types of model: the *uniform* model, where edge weights are sampled from a uniform distribution, and the *endpoint-independent model*, which

essentially makes an assumption on the independence of outgoing edge weights from each vertex (Moffat and Takaoka, 1987), which seems very similar to our assumption of independent order statistics. It remains to be seen whether this analysis can lead to better solutions to the problems discussed here, and indeed if the analysis applied to uniform models can be applied in our setting to uniform *matrices*.

It is interesting to consider the fact that our algorithm's running time is purely a function of the input data's *order statistics*, and in fact does not depend on the *data itself*. While it is pleasing that our assumption of independent order statistics appears to be a weak one, and is satisfied in a wide variety of applications, it ignores the fact that stronger assumptions may be reasonable in many cases. In factors with a high dynamic range, or when different factors have different scales, it may be possible to identify the maximum value very quickly, as we attempted to do in Section 7.5.1. Deriving faster algorithms that make stronger assumptions about the input data remains a promising avenue for future work.

Our algorithm may also lead to faster solutions for *approximately* passing a single message. While the stopping criterion of our algorithm *guarantees* that the maximum value is found, it is possible to terminate the algorithm earlier and state that the maximum has *probably* been found. A direction for future work would be to adapt our algorithm to determine the probability that the maximum has been found after a certain number of steps; we could then allow the user to specify an error probability, or a desired running time, and our algorithm could be adapted accordingly.

9. Conclusion

We have presented a series of approaches that allow us to improve the performance of exact and approximate max-product message-passing for models with factors smaller than their maximal cliques, and more generally, for models whose factors *that depend upon the observation* contain fewer latent variables than their maximal cliques. We are *always* able to improve the expected computational complexity in any model that exhibits this type of factorization, no matter the size or number of factors.

Acknowledgments

We would like to thank Pedro Felzenszwalb, Johnicholas Hines, and David Sontag for comments on initial versions of this paper, and James Petterson and Roslyn Lau for helpful discussions. NICTA is funded by the Australian Government's *Backing Australia's Ability* initiative, and the Australian Research Council's *ICT Centre of Excellence* program.

Appendix A. Asymptotic Performance of Algorithm 2 and Extensions

In this section we shall determine the expected-case running times of Algorithm 2 and Algorithm 6. Algorithm 2 traverses $\mathbf{v}_a$ and $\mathbf{v}_b$ until it reaches the smallest value of m for which there is some $j \leq m$ for which $m \geq p_b^{-1}[p_a[j]]$. If M is a random variable representing this smallest value of m, then we wish to find $E(M)$. While $E(M)$ is the number of 'steps' the algorithms take, each step takes $\Theta(K)$ when we have K lists. Thus the expected running time is $\Theta(KE(M))$.

To aid understanding our algorithm, we show the elements being read for specific examples of $\mathbf{v}_a$ and $\mathbf{v}_b$ in Figure 23. This figure reveals that the actual *values* in $\mathbf{v}_a$ and $\mathbf{v}_b$ are unimportant, and

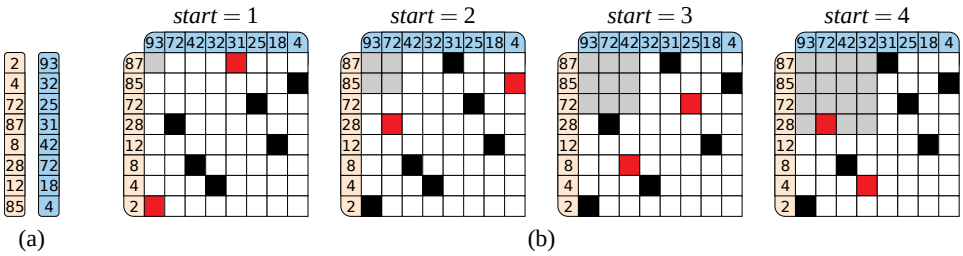


Figure 23: (a) The lists $\mathbf{v}_a$ and $\mathbf{v}_b$ before sorting; (b) Black squares show corresponding elements in the sorted lists ($\mathbf{v}_a[p_a[i]]$ and $\mathbf{v}_b[p_b[i]]$); red squares indicate the elements read during each step of the algorithm ($\mathbf{v}_a[p_a[start]]$ and $\mathbf{v}_b[p_b[start]]$). We can imagine expanding a gray box of size $start \times start$ until it contains an entry; note that the maximum is found during the first step.

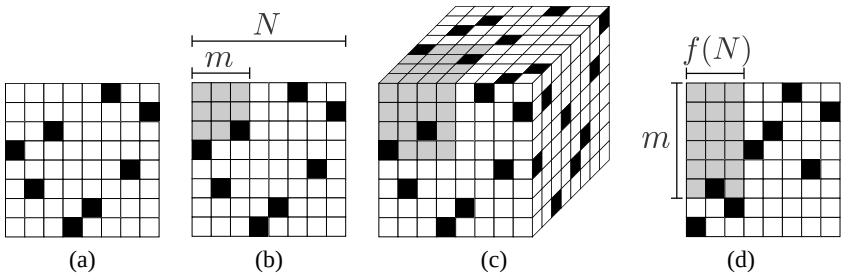


Figure 24: (a) As noted in Figure 23, a permutation can be represented as an array, where there is exactly one non-zero entry in each row and column; (b) We want to find the smallest value of m such that the gray box includes a non-zero entry; (c) A *pair* of permutations can be thought of as a cube, where every two-dimensional plane contains exactly one non-zero entry; we are now searching for the smallest gray cube that includes a non-zero entry; the faces show the projections of the points onto the exterior of the cube (the third face is determined by the first two); (d) For the sake of establishing an upper-bound, we consider a shaded region of width $f(N)$ and height m.

it is only the order statistics of the two lists that determine the performance of our algorithm. By representing a permutation of the digits 1 to N as shown in Figure 24 ((a), (b), and (d)), we observe that m is simply the width of the smallest square (expanding from the top left) that includes an element of the permutation (i.e., it includes i and $p[i]$).

Simple analysis reveals that the probability of choosing a permutation that does not contain a value inside a square of size m is

$$P(M > m) = \frac{(N-m)!(N-m)!}{(N-2m)!N!}. \tag{18}$$

This is precisely $1 - F(m)$, where $F(m)$ is the cumulative density function of M. It is immediately clear that $1 \leq M \leq \lfloor N/2 \rfloor$, which defines the best and worst-case performance of Algorithm 2.

Using the identity $E(X) = \sum_{x=1}^{\infty} P(X \geq x)$, we can write down a formula for the expected value of M:

$$E(M) = \sum_{m=0}^{\lfloor N/2 \rfloor} \frac{(N-m)!(N-m)!}{(N-2m)!N!}. \tag{19}$$

The case where we are sampling from multiple permutations simultaneously (i.e., Algorithm 6) is analogous. We consider $K-1$ permutations embedded in a K-dimensional hypercube, and we wish to find the width of the smallest shaded hypercube that includes exactly one element of the permutations (i.e., $i, p_1[i], \ldots, p_{K-1}[i]$). This is represented in Figure 24(c) for $K = 3$. Note carefully that K is the number of *lists* in (Equation 14); if we have K lists, we require $K-1$ permutations to define a correspondence between them.

Unfortunately, the probability that there is no non-zero entry in a cube of size m^K is not trivial to compute. It is possible to write down an expression that generalizes (Equation 18), such as

$$P^K(M > m) = \frac{1}{N!^{K-1}} \times \sum_{\sigma_1 \in S_N} \cdots \sum_{\sigma_{K-1} \in S_N} \bigwedge_{i=1}^{m} \left(\max_{k \in \{1 \ldots K-1\}} \sigma_k(i) > m \right) \tag{20}$$

(in which we simply enumerate over all possible permutations and 'count' which of them do not fall within a hypercube of size m^K), and therefore state that

$$E^K(M) = \sum_{m=0}^{\infty} P^K(M > m). \tag{21}$$

However, it is very hard to draw any conclusions from (Equation 20), and in fact it is intractable even to evaluate it for large values of N and K. Hence we shall instead focus our attention on finding an upper-bound on (Equation 21). Finding more computationally convenient expressions for (Equation 20) and (Equation 21) remains as future work.

A.1 An Upper-Bound on $E^K(M)$

Although (Equation 19) and (Equation 21) precisely define the running times of Algorithm 2 and Algorithm 6, it is not easy to ascertain the speed improvements they achieve, as the values to which the summations converge for large N are not obvious. Here, we shall try to obtain an upper-bound on their performance, which we assessed experimentally in Section 7. In doing so we shall prove Theorems 2 and 3.

Proof [Proof of Theorem 2] (see Algorithm 2) Consider the shaded region in Figure 24(d). This region has a width of $f(N)$, and its height m is chosen such that it contains precisely one non-zero entry. Let $\dot{M}$ be a random variable representing the height of the gray region needed in order to include a non-zero entry. We note that

$$E(\dot{M}) \in O(f(N)) \;\Rightarrow\; E(M) \in O(f(N));$$

our aim is to find the smallest $f(N)$ such that $E(\dot{M}) \in O(f(N))$. The probability that none of the first m samples appear in the shaded region is

$$P(\dot{M} > m) = \prod_{i=0}^{m} \left(1 - \frac{f(N)}{N-i}\right).$$

Next we observe that if the entries in our $N \times N$ grid do not define a permutation, but we instead choose a *random* entry in each row, then the probability (now for $\ddot{M}$) becomes

$$P(\ddot{M} > m) = \left(1 - \frac{f(N)}{N}\right)^m \tag{22}$$

(for simplicity we allow m to take arbitrarily large values). We certainly have that $P(\ddot{M} > m) \geq P(\dot{M} > m)$, meaning that $E(\ddot{M})$ is an upper-bound on $E(\dot{M})$, and therefore on $E(M)$. Thus we compute the expected value

$$E(\ddot{M}) = \sum_{m=0}^{\infty} \left(1 - \frac{f(N)}{N}\right)^m.$$

This is just a geometric progression, which sums to $N/f(N)$. Thus we need to find $f(N)$ such that

$$f(N) \in O\left(\frac{N}{f(N)}\right).$$

Clearly $f(N) \in O(\sqrt{N})$ will do. Thus we conclude that

$$E(M) \in O(\sqrt{N}).$$

■

Proof [Proof of Theorem 3] (see Algorithm 6) We would like to apply the same reasoning in the case of multiple permutations in order to compute a bound on $E^K(M)$. That is, we would like to consider $K-1$ *random* samples of the digits from 1 to N, rather than $K-1$ permutations, as random samples are easier to work with in practice.

To do so, we begin with some simple corollaries regarding our previous results. We have shown that in a permutation of length N, we expect to see a value less than or equal to f after N/f steps. There are now $f-1$ other values that are less than or equal to f amongst the remaining $N - N/f$ values; we note that

$$\frac{f-1}{N - \frac{N}{f}} = \frac{f}{N}.$$

Hence we expect to see the *next* value less than or equal to f in the next N/f steps also. A consequence of this fact is that we not only expect to see the *first* value less than or equal to f earlier in a permutation than in a random sample, but that when we sample m elements, we expect *more* of them to be less than or equal to f in a permutation than in a random sample.

Furthermore, when considering the *maximum* of $K-1$ permutations, we expect the first m elements to contain more values less than or equal to f than the maximum of $K-1$ random samples. (Equation 20) is concerned with precisely this problem. Therefore, when working in a K-dimensional hypercube, we can consider $K-1$ random samples rather than $K-1$ permutations in order to obtain an upper-bound on (Equation 21).

Thus we define $\ddot{M}$ as in (Equation 22), and conclude that

$$P(\ddot{M} > m) = \left(1 - \frac{f(N,K)^{K-1}}{N^{K-1}}\right)^m.$$

Thus the expected value of $\ddot{M}$ is again a geometric progression, which this time sums to $(N/f(N,K))^{K-1}$. Thus we need to find $f(N,K)$ such that

$$f(N,K) \in O\left(\left(\frac{N}{f(N,K)}\right)^{K-1}\right).$$

Clearly

$$f(N,K) \in O\left(N^{\frac{K-1}{K}}\right)$$

will do. As mentioned, each step takes $\Theta(K)$, so the final running time is $O(KN^{\frac{K-1}{K}})$. ■

To summarize, for problems decomposable into $K+1$ groups, we will need to find the index that chooses the maximal product amongst K lists; we have shown an upper-bound on the expected number of steps this takes, namely

$$E^K(M) \in O\left(N^{\frac{K-1}{K}}\right). \tag{23}$$

References

Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *Data Structures and Algorithms*. Addison-Wesley, 1983.

Srinivas M. Aji and Robert J. McEliece. The generalized distributive law. *IEEE Transactions on Information Theory*, 46(2):325–343, 2000.

Noga Alon, Zvi Galil, and Oded Margalit. On the exponent of the all pairs shortest path problem. *Journal of Computer and System Sciences*, 54(2):255–262, 1997.

Xiang Bai, Xingwei Yang, Longin Jan Latecki, Wenyu Liu, and Zhuowen Tu. Learning context-sensitive shape similarity by graph transduction. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 32(5):861–874, 2009.

Timothy M. Chan. More algorithms for all-pairs shortest paths in weighted graphs. In *Annual ACM Symposium on Theory of Computing*, pages 590–598, 2007.

James M. Coughlan and Sabino J. Ferreira. Finding deformable shapes using loopy belief propagation. In *ECCV*, 2002.

René Donner, Georg Langs, and Horst Bischof. Sparse MRF appearance models for fast anatomical structure localisation. In *BMVC*, 2007.

Gal Elidan, Ian Mcgraw, and Daphne Koller. Residual belief propagation: informed scheduling for asynchronous message passing. In *UAI*, 2006.

Pedro F. Felzenszwalb. Representation and detection of deformable shapes. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 27(2):208–220, 2005.

Pedro F. Felzenszwalb and Daniel P. Huttenlocher. Efficient belief propagation for early vision. *International Journal of Computer Vision*, 70(1):41–54, 2006.

Robert W. Floyd. Algorithm 97: Shortest path. *Communications of the ACM*, 5(6):345, 1962.

Michael L. Fredman and Robert Endre Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM*, 34(3):596–615, 1987.

Delbert R. Fulkerson and O. A. Gross. Incidence matrices and interval graphs. *Pacific Journal of Mathematics*, (15):835–855, 1965.

Michel Galley. A skip-chain conditional random field for ranking meeting utterances by importance. In *EMNLP*, 2006.

Stuart Geman and Donald Geman. Stochastic relaxation, gibbs distribution and the bayesian restoration of images. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 6(6):721–741, 1984.

David R. Karger, Daphne Koller, and Steven J. Phillips. Finding the hidden path: time bounds for all-pairs shortest paths. *SIAM Journal of Computing*, 22(6):1199–1217, 1993.

Leslie R. Kerr. The effect of algebraic structure on the computational complexity of matrix multiplication. *PhD Thesis*, 1970.

Kristian Kersting, Babak Ahmadi, and Sriraam Natarajan. Counting belief propagation. In *UAI*, 2009.

Uffe Kjærulff. Inference in bayesian networks using nested junction trees. In *Proceedings of the NATO Advanced Study Institute on Learning in Graphical Models*, 1998.

Vladimir Kolmogorov and Akiyoshi Shioura. New algorithms for the dual of the convex cost network flow problem with application to computer vision. Technical report, University College London, 2007.

Frank R. Kschischang, Brendan J. Frey, and Hans-Andrea Loeliger. Factor graphs and the sum-product algorithm. *IEEE Transactions on Information Theory*, 47(2):498–519, 2001.

M. Pawan Kumar and Philip Torr. Fast memory-efficient generalized belief propagation. In *ECCV*, 2006.

Xiang-Yang Lan, Stefan Roth, Daniel P. Huttenlocher, and Michael J. Black. Efficient belief propagation with learned higher-order markov random fields. In *ECCV*, 2006.

Bruce D. Lucas and Takeo Kanade. An iterative image registration technique with an application to stereo vision. In *IJCAI*, 1981.

Julian J. McAuley and Tibério S. Caetano. Exact inference in graphical models: is there more to it? *CoRR*, abs/0910.3301, 2009.

Julian J. McAuley and Tibério S. Caetano. Exploiting within-clique factorizations in junction-tree algorithms. *AISTATS*, 2010a.

Julian J. McAuley and Tibério S. Caetano. Exploiting data-independence for fast belief-propagation. *ICML*, 2010b.

Julian J. McAuley, Tibério S. Caetano, and Marconi S. Barbosa. Graph rigidity, cyclic belief propagation and point pattern matching. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 30(11):2047–2054, 2008.

Alistair Moffat and Tadao Takaoka. An all pairs shortest path algorithm with expected time $O(n^2 \log n)$. *SIAM Journal of Computing*, 16(6):1023–1031, 1987.

James D. Park and Adnan Darwiche. A differential semantics for jointree algorithms. In *NIPS*, 2003.

Mark A. Paskin. Thin junction tree filters for simultaneous localization and mapping. In *IJCAI*, 2003.

Kersten Petersen, Janis Fehr, and Hans Burkhardt. Fast generalized belief propagation for MAP estimation on 2D and 3D grid-like markov random fields. In *DAGM*, 2008.

Daniel Scharstein and Richard S. Szeliski. A taxonomy and evaluation of dense two-frame stereo correspondence algorithms. *International Journal of Computer Vision*, 47(1–3):7–42, 2001.

Leonid Sigal and Michael J. Black. Predicting 3D people from 2D pictures. In *AMDO*, 2006.

David Sontag, Talya Meltzer, Amir Globerson, Tommi Jaakkola, and Yair Weiss. Tightening LP relaxations for MAP using message passing. In *UAI*, 2008.

Volker Strassen. Gaussian elimination is not optimal. *Numerische Mathematik*, 14(3):354–356, 1969.

Jian Sun, Nan-Ning Zheng, and Heung-Yeung Shum. Stereo matching using belief propagation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 25(7):787–800, 2003.

Charles Sutton and Andrew McCallum. *Introduction to Conditional Random Fields for Relational Learning*. MIT Press, 2006.

Philip A. Tresadern, Harish Bhaskar, Steve A. Adeshina, Chris J. Taylor, and Tim F. Cootes. Combining local and global shape models for deformable object matching. In *BMVC*, 2009.

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

Ryan Williams. Matrix-vector multiplication in sub-quadratic time (some preprocessing required). In *SODA*, pages 1–11, 2007.

Virginia Vassilevska Williams and Ryan Williams. Subcubic equivalences between path, matrix, and triangle problems. In *FOCS*, 2010.