Fast Inference with Min-Sum Matrix Product

Pedro F. Felzenszwalb, *Member*,
IEEE Computer Society, and
Julian J. McAuley, *Student Member*, IEEE

Abstract—The MAP inference problem in many graphical models can be solved efficiently using a fast algorithm for computing min-sum products of $n \times n$ matrices. The class of models in question includes cyclic and skip-chain models that arise in many applications. Although the worst-case complexity of the min-sum product operation is not known to be much better than $O(n^3)$, an $O(n^{2.5})$ *expected time* algorithm was recently given, subject to some constraints on the input matrices. In this paper, we give an algorithm that runs in $O(n^2 \log n)$ expected time, assuming that the entries in the input matrices are independent samples from a uniform distribution. We also show that two variants of our algorithm are quite fast for inputs that arise in several applications. This leads to significant performance gains over previous methods in applications within computer vision and natural language processing.

Index Terms—Graphical models, MAP inference, min-sum matrix product.

◆

1 INTRODUCTION

MIN-SUM matrix product (MSP) (a.k.a. distance matrix product) is an important operation with applications in a variety of areas, including in inference algorithms for graphical models [15], parsing with context-free grammars [23], and shortest paths algorithms [1].

Our main interest in MSP involves its application to MAP inference in graphical models and optimization problems that have similar form. It is well known that inference in discrete graphical models with low tree width can be done using dynamic programming and belief propagation. McAuley and Caetano [15] showed that faster inference can be done in a large class of models if we have a fast method for MSP. This class includes cyclic and skip-chain models that arise in many applications including in natural language understanding and computer vision. See Fig. 1 for some examples.

Let A and B be two $n \times n$ matrices. The MSP of A and B is the $n \times n$ matrix $C = A \otimes B$ defined by

$$C_{ik} = \min_j A_{ij} + B_{jk}. \qquad (1)$$

Note that this is exactly matrix multiplication in the min-plus (tropical) semiring.

Standard algorithms for inference with a tree-width 2 model take $O(mn^3)$ time, where m is the number of variables in the model and n is the number of possible values for each variable. For models that contain only pairwise factors, inference can be done in $O(mf(n))$ time if we have an algorithm for computing MSP of $n \times n$ matrices in $O(f(n))$ time (see Section 2).

The brute-force approach for computing MSP of $n \times n$ matrices takes $O(n^3)$ time. Unfortunately, there is no known

method that improves this bound by a significant amount in the worst case. An important difference from the standard matrix product is that the minimum operation does not have an inverse. This means that fast matrix multiplication methods that rely on a ring structure, such as Strassen's algorithm [21], cannot be directly applied to compute MSP.

Our main theoretical result is an algorithm for MSP that runs in $O(n^2 \log n)$ expected time, assuming that the entries of each matrix are independent samples from a uniform distribution. Our experimental results show that the method also performs well under realistic inputs that arise in several applications.

Our basic algorithm uses a Fibonacci heap (or similar structure) and is mainly of theoretical interest. The algorithm can be implemented with an integer queue to obtain a practical solution. We also describe an alternative algorithm that computes exact values using a scaling technique and avoids any complex data structure. Our experimental results show that the methods perform well in three different applications: interactive image segmentation with active contours models ("snakes"), point pattern matching with belief propagation, and text denoising with skip-chain models.

1.1 Related Work

Our work is motivated by McAuley and Caetano [15], who noted the application of MSP for MAP inference with graphical models and gave an $O(n^{2.5})$ expected time algorithm for MSP. The method in [15] assumes every permutation of values in the inputs occurs with equal probability. This is a weaker assumption than the one we make in our analysis and could lead to a faster method in some applications. Section 4 compares the two methods and shows that our algorithms perform better in several applications.

The worst-case complexity of the MSP operation has been heavily studied in the theoretical computer science community because of its relation to the all-pairs-shortest-paths (APSP) problem. The worst-case asymptotic complexity of computing MSP of $n \times n$ matrices is the same as solving APSP on dense graphs with n nodes [1]. To our knowledge, the best-known algorithm for the APSP problem takes $O(n^3 / \log n)$ time in the worst case [8]. The search for a truly subcubic algorithm ($O(n^{3-\epsilon})$) is a significant open problem in the area.

There are several known algorithms for the APSP problem which have good expected runtime, assuming that the input graph comes from a simple distribution (e.g., [11], [17], [12]). However, the usual reduction of MSP to APSP (see [1]) leads to graphs that have deterministic structure, violating the assumptions made by the APSP algorithms designed for random graphs.

Our basic algorithm can be seen as an application of Knuth's lightest derivation method (KLD) [14], [10] with a special stopping criterion. Felzenszwalb and McAllester [10] suggested using KLD and an A* version of it for inference in graphical models. However, there is a difference between the approach we use here and the one suggested in [10]. When doing inference on a large model, we solve several small lightest derivation problems, each defined by a single MSP computation. In contrast, Felzenszwalb and McAllester [10] suggest solving a single large lightest derivation problem. Solving a sequence of small problems leads to better performance and simplifies the implementation.

Our algorithms improve dynamic programming and message passing methods for inference in low tree-width graphs when the number of possible values for a variable is large. Another approach for inference in some classes of graphical models involves graph cuts [7], [6]. However, these methods are typically used for models with high tree width, a relatively small number of possible values per variable, and restricted classes of potential functions. None of the applications we consider can be easily addressed with graph cuts.

- *P.F. Felzenszwalb is with the Department of Computer Science, University of Chicago, 1100 E. 58th St., Chicago, IL 60637.
  E-mail: pff@cs.uchicago.edu.*
- *J.J. McAuley is with NICTA and the Australian National University, Locked Bag 8001, Canberra, ACT 2601, Australia.
  E-mail: julian.mcauley@gmail.com.*

d e n o i s e

(a) Skip-chain model

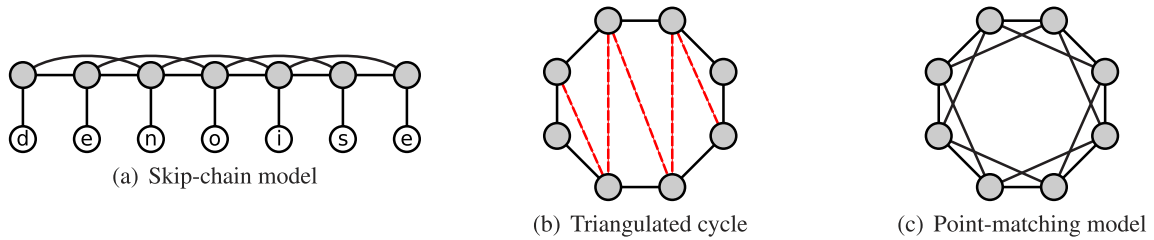(b) Triangulated cycle　　　(c) Point-matching model

Fig. 1. Some typical graphical models with third-order cliques, but only pairwise factors.

2 MAP INFERENCE AND MIN-SUM MATRIX PRODUCT

Let $G = (V, E)$ be a graph with m nodes. Let $\mathbf{x} = (x_1, \ldots, x_m)$ be a set of variables associated with the nodes in V. We are interested in solving optimization problems of the form

$$\mathbf{x}^* = \arg\min_{\mathbf{x}} \sum_{C \in \mathcal{C}} \Phi_C(\mathbf{x}_C), \qquad (2)$$

where $\mathcal{C}$ is the set of maximal cliques in G, $\mathbf{x}_C$ denotes the variables associated with nodes in C, and Φ_C is a *potential function* assigning a cost to each possible configuration of values for these variables.

Optimization problems of this type arise in many situations, including in MAP estimation with graphical models. Exact or approximate solutions are often found using some form of message-passing technique. This includes classical dynamic programming methods [5], [3], [4], loopy belief propagation [24], and the junction-tree algorithm [2]. Sometimes messages are computed between cliques of the original graph and sometimes over a triangulated version.

In general, the message passed from a clique A to a clique B takes the form

$$m_{A \to B}(\mathbf{x}_{A \cap B}) = \min_{\mathbf{x}_{A \setminus B}} \left(\Phi_X(\mathbf{x}_A) + \sum_{C \in \Gamma(A) \setminus B} m_{C \to A}(\mathbf{x}_{A \cap C}) \right), \qquad (3)$$

where $\Gamma(A)$ is the set of cliques neighboring A.

If the model is triangulated and $m_{A \to B}$ is computed after A receives messages from all neighbors except B (i.e., $\Gamma(A) \setminus B$), this leads to the junction-tree algorithm. This is also equivalent to nonserial dynamic programming in a decomposable graph [5], [4]. In loopy belief propagation, messages are updated in parallel or in some arbitrary order until convergence. After messages are computed, a solution $\mathbf{x}^*$ can be obtained by computing beliefs using a similar computation.

As noted in [15], there are many graphical models whose potential functions Φ_C are decomposable into smaller factors, i.e.,

$$\Phi_C(\mathbf{x}_C) = \sum_{F \subset C} \Phi_F(\mathbf{x}_F). \qquad (4)$$

This is a general phenomenon that arises, for example, when one triangulates a model. Triangulation creates new edges, and thus larger cliques, but the potential functions of the triangulated graphs can always be decomposed into the original potential functions.

As in [15], we focus on the case where the potentials take the form

$$\Phi_{ijk}(x_i, x_j, x_k) = \Phi_{ij}(x_i, x_j) + \Phi_{ik}(x_i, x_k) + \Phi_{jk}(x_j, x_k). \qquad (5)$$

That is, we have cliques of size 3 with pairwise factors. Our algorithms can also be generalized to other factorizations discussed in [15], but we concentrate on this particular case because it is the most common in typical applications. For example, if we have a cyclic model, globally optimal solutions $\mathbf{x}^*$ can be obtained by applying the junction-tree algorithm to a triangulated graph (Fig. 1b). We describe experiments with a model of this type for

image segmentation in Section 4.1. Another example is a skip-chain model [22] where we have a sequence of hidden variables and a potential function between pairs of variables that have distance at most 2 from each other (Fig. 1a). Section 4.3 illustrates an application of a model of this type for text denoising.

When the potentials are of the form in (5), a message from a clique $A = \{i, j, k\}$ to a clique $B = \{i, l, k\}$ takes the form

$$m_{A \to B}(x_i, x_k) = \Psi_{ik}(x_i, x_k) + \min_{x_j} \Psi_{ij}(x_i, x_j) + \Psi_{jk}(x_j, x_k), \qquad (6)$$

where Ψ_{ij} is the sum of Φ_{ij} and messages from cliques that intersect A at (i, j), Ψ_{jk} is the sum of Φ_{jk} and messages from cliques that intersect A at (j, k), and Ψ_{ik} is the sum of Φ_{ik} and messages from cliques, other than B, that intersect A at (i, k).

Note that (6) is essentially equivalent to the MSP (1) of two matrices (Ψ_{ij} and Ψ_{jk}) of size n, where n is the number of possible values for each variable in the model. The only difference is that (6) requires adding another matrix (Ψ_{ik}) to the result. Suppose we can compute the MSP of two $n \times n$ matrices in $O(f(n))$ time. Then, we can compute messages in $O(f(n))$ time. Consider a problem with m variables in which each variable can take one of n possible values. If the graph has tree-width 2, we can triangulate it and use the junction-tree algorithm to find $\mathbf{x}^*$ in $O(mf(n))$ time.

Just like MSP can be used for MAP inference with graphical models, standard matrix multiplication can be used for computing marginals. Thus, matrix multiplication algorithms such as Strassen's method [21] can be used for marginal computation in the class of models that we consider here. We note, however, that such methods are not very practical due to high constants.

3 THE ALGORITHM

Here, we describe our basic algorithm (Algorithm 1) for computing $C = A \otimes B$. We assume all entries in A and B are nonnegative. Negative (finite) entries can be eliminated by adding a constant to each matrix and subtracting the constants from the resulting C.

The algorithm exploits a priority queue to avoid computing most sums $A_{ij} + B_{jk}$. We initialize the values of C to ∞ and insert all entries of A, B, and C into a queue Q, with priority given by their value. We repeatedly remove items from Q and insert them into a set S. Whenever A_{ij} (respectively, B_{jk}) is removed from Q, we combine it with entries of the form B_{jk} (respectively, A_{ij}) that are already in S and update $C_{ik} = \min(C_{ik}, A_{ij} + B_{jk})$. We stop when all entries of C are in S. Pseudocode for the algorithm is shown in the left column of Fig. 2.

Theorem 1. *If all entries in A and B are nonnegative, then Algorithm 1 correctly computes C.*

Proof. Let $j = \arg\min_j A_{ij} + B_{jk}$. Clearly, we always have $C_{ik} \geq A_{ij} + B_{jk}$. It suffices to show that when C_{ik} is removed from Q, we have $C_{ik} = A_{ij} + B_{jk}$. Since the entries in A and B are nonnegative, $A_{ij}, B_{jk} \leq C_{ik}$ and both A_{ij} and B_{jk} will be removed from Q before C_{ik}. This implies that when C_{ik} is removed from Q, we have $C_{ik} = A_{ij} + B_{jk}$. □

Algorithm 1 Find $C = A \otimes B$

1: $S := \varnothing$
2: $\forall ik\ C_{ik} := \infty$
3: Initialize Q with entries of A, B, C
4: **while** S does not contain all C_{ik} **do**
5:     item := remove-min(Q)
6:     $S := S \cup$ item
7:     **if** item = A_{ij} **then**
8:        **for** $B_{jk} \in S$ **do**
9:           **if** $A_{ij} + B_{jk} < C_{ik}$ **then**
10:             $C_{ik} := A_{ij} + B_{jk}$
11:             decrease-key(Q, C_{ik})
12:           **end if**
13:        **end for**
14:     **end if**
15:     **if** item = B_{jk} **then**
16:        **for** $A_{ij} \in S$ **do**
17:           **if** $A_{ij} + B_{jk} < C_{ik}$ **then**
18:             $C_{ik} := A_{ij} + B_{jk}$
19:             decrease-key(Q, C_{ik})
20:           **end if**
21:        **end for**
22:     **end if**
23: **end while**

Algorithm 2 Find $C = A \otimes B$

1: $\forall ik\ C_{ik} := \infty$
2: $T := $ tmin
3: **while** $\max_{ik} C_{ik} > T$ **do**
4:     $I[j] := \{i \mid A_{ij} \leq T\}$
5:     $K[j] := \{k \mid B_{jk} \leq T\}$
6:     **for** $j \in \{1 \dots n\}$ **do**
7:        **for** $i \in I[j]$ **do**
8:           **for** $k \in K[j]$ **do**
9:             $c := A_{ij} + B_{jk}$
10:             **if** $c < C_{ik}$ **then**
11:                $C_{ik} := c$
12:             **end if**
13:           **end for**
14:        **end for**
15:     **end for**
16:     $T := 2T$
17: **end while**

Fig. 2. Two algorithms for computing MSP. The first (Algorithm 1) keeps track of a Fibonacci heap or similar and data structure Q and a set S. The second (Algorithm 2) avoids the use of "exotic" data structures and is very fast in practice, but may be sensitive to the schedule for T.

Theorem 2. *If all entries in A and B are i.i.d. samples from a uniform distribution, then Algorithm 1 can be implemented to run in $O(n^2 \log n)$ expected time.*

Proof. First, note that we can assume that the entries in A and B come from a uniform distribution over $[0,1]$ by scaling them and then rescaling the resulting C accordingly.

We keep two arrays of linked lists I and K such that $I[j]$ stores indices i for which A_{ij} is in S while $K[j]$ stores indices k for which B_{jk} is in S. When an entry is removed from Q, we find the entries in S that combine with it in constant time per entry. For example, when A_{ij} is removed from Q, we iterate over k in $K[j]$. Thus, the running time of the algorithm is dominated by the additions and priority queue operations.

Let N be the number of additions done by the algorithm. We perform $O(n^2)$ insertions and remove-min operations and $O(N)$ decrease-key operations. Lemma 1 shows that $E[N]$ is $O(n^2 \log n)$. Using a Fibonacci heap, we obtain $O(1)$ time insertion and decrease key and $O(\log n)$ time remove-min. This leads to the running time bound of $O(n^2 \log n)$. □

Lemma 1. *Let N be the number of additions performed by Algorithm 1. If the entries in A and B are i.i.d. samples from the uniform distribution over $[0,1]$, then $E[N]$ is $O(n^2 \log n)$.*

Proof. Let $C = A \otimes B$, and let M be the maximum value in C. The algorithm only adds A_{ij} and B_{jk} if both are at most M. Otherwise, at least one of A_{ij} or B_{jk} will not be removed from Q before the algorithm stops. Let $X_{ijk} = 1$ if $A_{ij} \leq M$ and $B_{jk} \leq M$, and 0 otherwise. The number of additions performed by the algorithm is $N = \sum_{ijk} X_{ijk}$. Using linearity of expectation, we have

$$E[N] = \sum_{ijk} E[X_{ijk}] = \sum_{ijk} P(X_{ijk} = 1).$$

First, we show that M is small with high probability because each entry in C is the minimum of n values. Then, we use the fact that A_{ij} and B_{jk} are both small with low probability. This will imply that $X_{ijk} = 1$ with very low probability.

Let ϵ be a value between 0 and 1. $M \geq \epsilon$ if and only if some $C_{ik} \geq \epsilon$. Using the union bound,

$$P(M \geq \epsilon) \leq \sum_{ik} P(C_{ik} \geq \epsilon).$$

$C_{ik} \geq \epsilon$ if and only if for all j we have $A_{ij} + B_{jk} \geq \epsilon$. For fixed (i,k), these are independent events; thus,

$$P(C_{ik} \geq \epsilon) = \prod_j P(A_{ij} + B_{jk} \geq \epsilon).$$

Let $s = A_{ij} + B_{jk}$. Since A_{ij} and B_{jk} are independent samples from the uniform distribution over $[0,1]$, we have $P(s \leq x) = x^2/2$. Thus,

$$P(A_{ij} + B_{jk} \geq \epsilon) = 1 - \epsilon^2/2 \ \text{ and } \ P(C_{ik} \geq \epsilon) = (1 - \epsilon^2/2)^n.$$

Using $1 - x \leq e^{-x}$, we obtain

$$P(C_{ik} \geq \epsilon) \leq e^{-n\epsilon^2/2}.$$

Now, we can see that M is small with high probability or large with low probability:

$$P(M \geq \epsilon) \leq n^2 e^{-n\epsilon^2/2}.$$

Note that $P(A_{ij} \leq \epsilon) = P(B_{jk} \leq \epsilon) = \epsilon$. Since these are independent events, we have

$$P(A_{ij} \leq \epsilon \wedge B_{jk} \leq \epsilon) = \epsilon^2. \qquad (7)$$

Let E_1 denote the event that $M \geq \epsilon$ and E_2 denote the event that $A_{ij} \leq \epsilon \wedge B_{jk} \leq \epsilon$. We have that $X_{ijk} = 1$ requires at least one of E_1 or E_2 to hold. Using the union bound:

$$P(X_{ijk} = 1) \leq P(E_1) + P(E_2) \leq n^2 e^{-n\epsilon^2/2} + \epsilon^2.$$

Now, we can pick ϵ so that both the above terms are small. It is sufficient to pick $\epsilon^2 = \frac{6 \log n}{n}$. Note that as long as n is large enough, we satisfy the requirement that $\epsilon \leq 1$. With this choice:

$$P(X_{ijk} = 1) \leq \frac{1 + 6 \log n}{n}.$$

Finally, we obtain

$$E[N] \leq n^3 \left(\frac{1 + 6 \log n}{n} \right) = n^2(1 + 6 \log n). \qquad (8)$$

So, $E[N]$ is $O(n^2 \log n)$.   □

3.1 Integer Queue

To obtain the desired running time bound for Algorithm 1, we need a complex data structure supporting $O(1)$ time decrease-key operations, such as a Fibonacci heap. We have found that this leads to poor performance since such data structures are relatively slow in practice.

Suppose the entries in A and B are integers in $[0, K]$. Then, we can initialize C_{ik} to $2K$, and the priorities in Q will always be integers in $[0, 2K]$. We can represent such a queue by an array of length $2K + 1$, with each entry $Q[p]$ holding a list of values with priority p.

An important property of Algorithm 1 is that the minimum priority of items in Q never decreases. This is because, when the value of C_{ik} is decreased, it does not go below the last value removed from Q. This makes it possible to perform k queue operations in $O(k + K)$ time. Initialization takes $O(K)$ time. Insertions and decrease key each take $O(1)$ time, while k remove-min operations take $O(k + K)$ in total. During remove-min, we may need to search for the minimum p with $Q[p]$ not empty. But, since the minimum never decreases, we never need to search over the same priority twice.

Using an integer queue, Algorithm 1 runs in $O(n^2 \log n + K)$ time. If the entries in A and B are not integers or have high value, we can scale and round them to ensure that K is not too large. If the maximum value in A and B is v, using K priority bins leads to bins of size $b = v/K$. By itself, this approach would introduce an additive error bounded by b. We can avoid this error by making the priority of C_{ik} equal to $C_{ik} + b$. This ensures that A_{ij} and B_{jk} will come off the queue before C_{ik} whenever $j = \text{argmin} A_{ij} + B_{jk}$. Picking $K = \Theta(n^2 \log n)$ ensures that the priorities are accurately represented and we still obtain the expected running time bound of $O(n^2 \log n)$.

3.2 Scaling Method

Here, we describe an alternative algorithm (Algorithm 2) that avoids using a priority queue and computes exact solutions to the MSP problem.

Note that if we knew $M = \max_{ik} C_{ik}$, a very simple algorithm could be developed for computing C. Since the entries in A and B are nonnegative, we have that $C_{ik} = \min_j A_{ij} + B_{jk}$ for those j such that $A_{ij} \leq M \wedge B_{jk} \leq M$. By (8), this would allow us to compute C in $O(n^2 \log n)$ expected time, without any need for a priority queue.

Of course, we do not know M. In practice, we guess a value T and compute $C_{ik} = \min_j A_{ij} + B_{jk}$ for those j such that $A_{ij} \leq T \wedge B_{jk} \leq T$. If $\max_{ik} C_{ik} \leq T$, we have correctly computed C since larger values of A_{ij} or B_{jk} could not lead to smaller values for

C_{ik}. Otherwise, we double T and try again. Pseudocode for the algorithm is shown in the right column of Fig. 2.

If we were able to choose $T := M$, then Algorithm 2 would perform the same additions as Algorithm 1. Note that Algorithm 2 terminates before $T > 2M$. By (7), doubling T increases the expected number of additions by a factor of 4. Thus, the total expected number of additions performed by Algorithm 2 is at most a constant times the number of additions performed by Algorithm 1. In each iteration, Algorithm 2 also takes $O(n^2)$ time to check C and initialize the lists I and K. The total expected running time is $O(n^2 \log n)$ as long as the number of iterations is $O(\log n)$. This holds as long as the initial value for T is not too small.

3.3 Speedups

There are several speedups that improve the running time of our algorithms in practical situations.

1. From each entry in A we subtract the minimum value in its row and from each entry in B we subtract the minimum value in its column. These minima are added back to the resulting C. This makes the values in C closer to the values in A and B and allows the algorithm to stop earlier.
2. In both algorithms, we can remove entries from the set S, or lists I and K, if we already know all of the values in a row or column of C. We keep track of which rows/columns of C are done, and remove the entries the first time we consider them and realize they can no longer affect the result.
3. Let $a(j) = \min_i A_{ij}$ and $b(j) = \min_k B_{jk}$ be column and row minima in A and B, respectively. Entries in C derived from A_{ij} must have value at least $A_{ij} + b(j)$, and entries in C derived from B_{jk} must have value of at least $B_{jk} + a(j)$. This can be used to increase the priority of the items in Q in Algorithm 1, in which case fewer items might be processed before the algorithm stops. We let A_{ij} have priority $A_{ij} + b(j)$ and B_{jk} have priority $B_{jk} + a(j)$. This does not affect the result because A_{ij} and B_{jk} will still come off the queue before C_{ik} whenever $j = \text{argmin} A_{ij} + B_{jk}$. This can be seen as an A* version of the algorithm. The modification is beneficial to handle matricies with non-i.i.d. entries. A similar idea can be used in Algorithm 2 to decide which entries to include in I and K. We only need to include items that could be combined with each other to get values of at most T. Thus, we can take $I[j] := \{i \mid A_{ij} + b(j) \leq T\}$ and $K[j] := \{k \mid a(j) + B_{jk} \leq T\}$.

4 EXPERIMENTS

We implemented our algorithms and tested them in several applications by comparing them to the naive (brute force) method for MSP and the method from [15]. Note that all of these methods are guaranteed to find an exact solution to the MSP problem. Our implementation of Algorithm 1 uses an integer queue, as described in Section 3.1. Both Algorithms 1 and 2 were implemented using all speedups described in Section 3.3. All methods were implemented in C++ using the GNU C compiler on a 2.8 GHz Intel PowerMac running Mac OS X 10.5.

First, we evaluate our algorithms on uniform i.i.d. matrices. Note that such matrices satisfy the random rank statistics assumption made in [15]. Fig. 3 shows the performance of the different methods with running times on the left, and the number of additions done by each method on the right. This confirms that our methods have better asymptotic complexity on random inputs. The experiments below show that results on structured inputs that arise in practical applications are similar to the random case.

4.1 Interactive Image Segmentation

Here, we consider the problem of image segmentation using active contour models ("snakes") [13], [3]. Fig. 4 illustrates an example. In
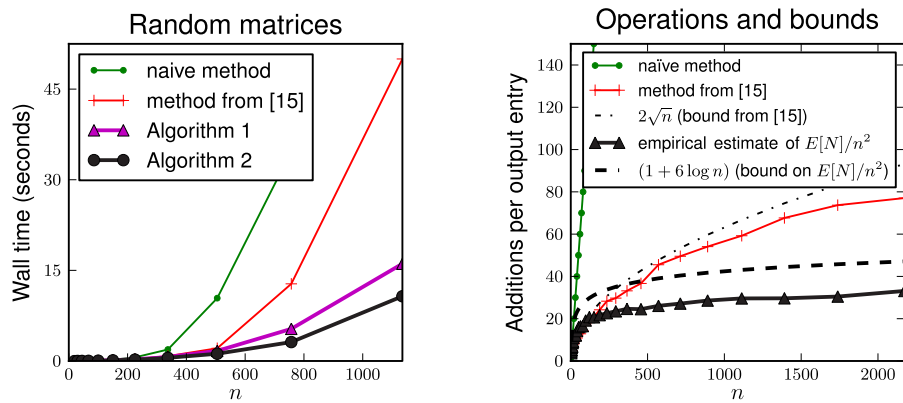
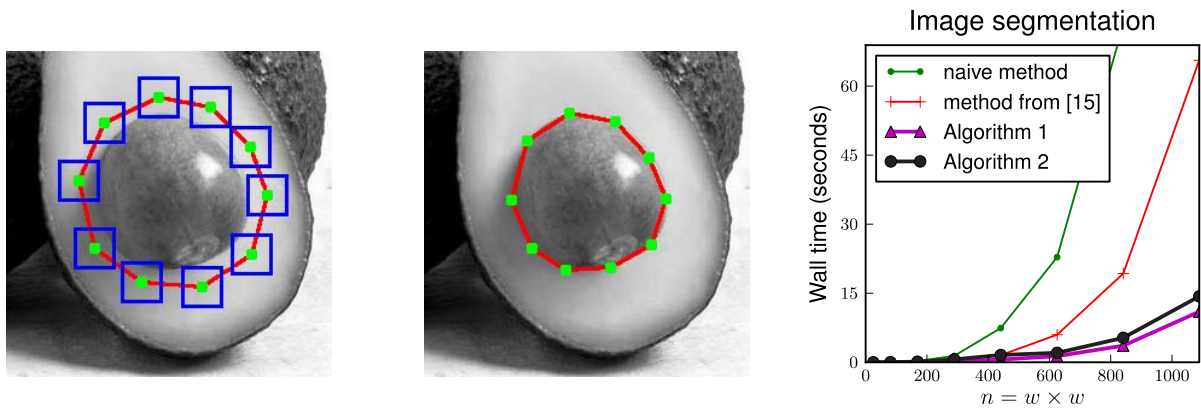Fig. 3. Left: Runtime of different MSP algorithms. Right: Number of additions per output entry.

Fig. 4. Interactive image segmentation with an active contour model. Left: Initial placement of the contour and search neighborhoods for the control points. Center: Final segmentation. Right: Running time as a function of the search space size using different MSP algorithms.

this application, a coarse segmentation of an object is provided by the user, in the form of a polygonal curve with m control points. The goal is to improve the segmentation by moving the control points within a search window around their initial positions.

Let $\mathbf{x} = (x_1, \ldots, x_m)$ denote the position of the control points. In our experiments, the final segmentation was obtained by solving a problem of the form

$$\mathbf{x}^* = \arg\min_{\mathbf{x}} \sum_{i=1}^{m} \frac{1}{\mathrm{grad}(x_i, x_{i+1})} + \|x_i - x_{i+1}\|^2.$$

Here, x_i is constrained to be in a $w \times w$ window around its initial location. The value $\mathrm{grad}(p, q)$ is a measure of the gradient magnitude along the line segment between p and q (we want the object boundary to align with high-gradient regions). The second term in the sum encourages compact boundaries with control points that are approximately uniformly spaced.

Solving for $\mathbf{x}^*$ is equivalent to MAP estimation with a cyclic graphical model and can be done via the junction-tree algorithm in a triangulated graph, such as the one in Fig. 1b. Using the naive MSP algorithm, this takes $O(mn^3)$ time, where $n = w^2$ is the number of possible positions for each control point.[1] Fig. 4 shows the result in one particular image and the total running time obtained on this image using different methods for MSP as a subroutine.

4.2 Point Pattern Matching

Many of the problems suggested in [15] involved finding maps between two point sets. Examples include OCR [9], pose reconstruction [20], SLAM [18], and point pattern matching [16].

Here, we search for a "template" $\mathbf{s}$ containing m points $(s_1, \ldots, s_m)$ within a "target" $\mathbf{t}$ containing n points $(t_1, \ldots, t_n)$. The target consists of a transformed and noisy version of the template, together with outliers. An example is shown in Fig. 5.

A solution to this matching problem is defined by a map from $\mathbf{s}$ to $\mathbf{t}$. Such a map is defined by $\mathbf{x} = (x_1, \ldots, x_m)$ with $x_i \in \{1, \ldots, n\}$. Here, $x_i = j$ indicates that s_i is mapped to t_j. The quality of the solution is defined by how well distances in s are preserved under the map $\mathbf{x}$. We let E be a set of edges over the points in $\mathbf{s}$ specifying which distances should be (explicitly) preserved. The optimal solution is defined as

$$\mathbf{x}^* = \arg\min_{\mathbf{x}} \sum_{(i,j) \in E} g(\|s_i - s_j\|, \|t_{x_i} - t_{x_j}\|),$$
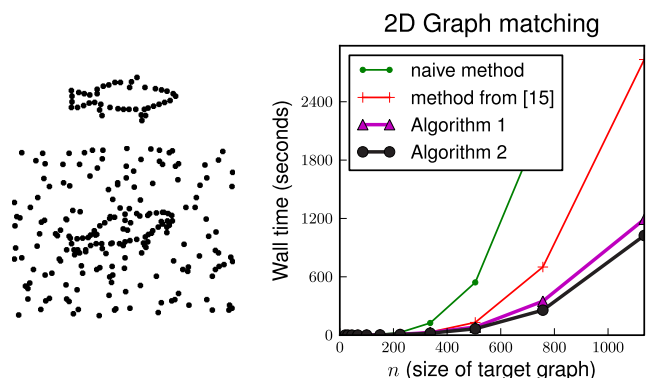

Fig. 5. Point pattern matching. Left: A template (above) and a scene (below) with noise and outliers. Right: Running times for matching using different MSP algorithms as a subroutine.

---

1. Another common approach is to try every possible location for one point and for each choice optimizing the other point locations using dynamic programming on a chain. This also takes $O(mn^3)$ time.
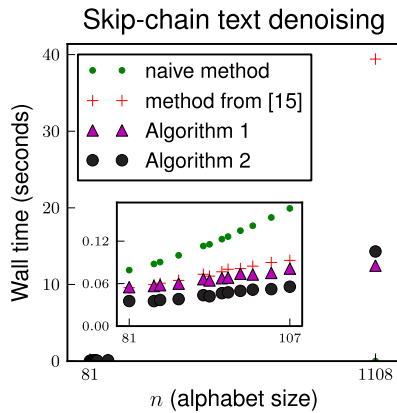
Fig. 6. Text denoising experiment. The box is a closeup of bottom left part of the graph.

where the function $g(a, b)$ defines a robust elasticity constraint enforcing that $a \approx b$.

Solving for $\mathbf{x}^*$ is equivalent to MAP estimation in a graphical model with topology defined by E. It was shown in [15] that in many applications E forms a tractable model. Here, we use the model from [16] with the set of edges E shown in Fig. 1c. For inference, we run loopy belief propagation for 25 iterations in the loop of "width" 2. This takes $O(mn^3)$ time per iteration using the naive MSP method as a subroutine (the iterative nature of this method accounts for the higher total running time compared to the other experiments). The performance on a particular problem instance using different MSP methods is shown in Fig. 5. Note that we could perform pairwise belief propagation for the model from Fig. 1c in $O(mn^2)$ time per iteration. However, McAuley et al. [16] show that passing messages between cliques leads to better theoretical guarantees. Our MSP algorithms allow us to maintain the guarantees while incurring an overhead of only $O(\log n)$ compared to pairwise belief propagation.

4.3 Skip-Chain Models for Text Denoising

In [22], it was observed that powerful inference procedures can be developed by introducing long-range dependencies into pairwise graphical models.

In this experiment, we adapt a simple Markov model for text denoising (typo correction): We model not only the relationship between neighboring characters, but also the relationship between characters at distance 2. This leads to a graphical model of the type shown in Fig. 1a.

Let $\mathbf{t} = (t_1, \ldots, t_m)$ be a sequence of m characters from an alphabet of size n. We assume that each character was corrupted with probability p. The MAP estimate of the hidden (uncorrupted) sequence $\mathbf{x} = (x_1, \ldots, x_m)$ is given by

$$\mathbf{x}^* = \arg\max_{\mathbf{x}} \underbrace{\prod_{i=1}^{m} [p\delta(t_i \neq x_i)) + (1-p)\delta(t_i = x_i)]}_{\text{noise model}}$$

$$\underbrace{\prod_{i=1}^{m-1} q_1(x_i, x_{i+1}) \prod_{i=1}^{m-2} q_2(x_i, x_{i+2})}_{\text{prior}}.$$

Here, $\delta(v)$ is the indicator function that equals 1 if v is true and 0 if v is false. Our priors q_1, q_2 are extracted from the statistics of sentences in the Leipzig corpora [19]. The model has tree-width 2 and inference again requires $O(mn^3)$ operations using the naive MSP method within the junction tree algorithm. The average performance (over 10 sentences each with 200 characters) using different methods for MSP is shown in Fig. 6. The largest alphabet

we consider comes from the Korean data, which contains 1,108 unique characters.

5 CONCLUSION

The MSP operation plays an important role for inference in a large class of graphical models. Our basic algorithm runs in $O(n^2 \log n)$ expected time assuming the entries in each input matrix are independent samples from a uniform distribution. Despite this strong assumption, we show that the algorithm can be made very fast for inputs that arise in practical applications, achieving significant performance gains over existing methods. An interesting open question involves showing that the algorithm has good running time bounds for more general inputs once we include the speedups described in Section 3.3. Another direction for future work involves other applications of MSP, such as parsing with context-free grammars.

REFERENCES

[1] A.V. Aho, J.E. Hopcroft, and J.D. Ullman, *The Design and Analysis of Computer Algorithms.* Addison-Wesley, 1974.

[2] S.M. Aji and R.J. McEliece, "The Generalized Distributive Law," *IEEE Trans. Information Theory,* vol. 46, no. 2, pp. 325-343, Mar. 2000.

[3] A. Amini, T. Weymouth, and R. Jain, "Using Dynamic Programming for Solving Variational Problems in Vision," *IEEE Trans. Pattern Analysis and Machine Intelligence,* vol. 12, no. 9, pp. 855-867, Sept. 1990.

[4] Y. Amit and A. Kong, "Graphical Templates for Model Registration," *IEEE Trans. Pattern Analysis and Machine Intelligence,* vol. 18, no. 3, pp. 225-236, Mar. 1996.

[5] U. Bertele and F. Brioschi, *Nonserial Dynamic Programming.* Academic Press, 1972.

[6] Y. Boykov and M.-P. Jolly, "Interactive Graph Cuts for Optimal Boundary and Region Segmentation of Objects in N-D Images," *Proc. IEEE Int'l Conf. Computer Vision,* 2001.

[7] Y. Boykov, O. Veksler, and R. Zabih, "Fast Approximate Energy Minimization via Graph Cuts," *IEEE Trans. Pattern Analysis and Machine Intelligence,* vol. 23, no. 11, pp. 1222-1239, Nov. 2001.

[8] T.M. Chan, "More Algorithms for All-Pairs Shortest Paths in Weighted Graphs," *Proc. Ann. ACM Symp. Theory of Computing,* pp. 590-598, 2007.

[9] J.M. Coughlan and S.J. Ferreira, "Finding Deformable Shapes Using Loopy Belief Propagation," *Proc. European Conf. Computer Vision,* 2002.

[10] P.F. Felzenszwalb and D. McAllester, "The Generalized A* Architecture," *J. Artificial Intelligence Research,* vol. 29, pp. 153-190, 2007.

[11] A.M. Frieze and G.R. Grimmett, "The Shortest-Path Problem for Graphs with Random Arc-Lengths," *Discrete Applied Math.,* vol. 10, no. 1, pp. 57-77, 1985.

[12] D.R. Karger, D. Koller, and S.J. Phillips, "Finding the Hidden Path: Time Bounds for All-Pairs Shortest Paths," *SIAM J. Computing,* vol. 22, no. 6, pp. 1199-1217, 1993.

[13] M. Kass, A. Witkin, and D. Terzopoulos, "Snakes: Active Contour Models," *Int'l J. Computer Vision,* vol. 1, no. 4, pp. 321-331, 1987.

[14] D. Knuth, "A Generalization of Dijkstra's Algorithm," *Information Processing Letters,* vol. 6, no. 1, pp. 1-5, 1977.

[15] J.J. McAuley and T.S. Caetano, "Exploiting within-Clique Factorizations in Junction-Tree Algorithms," *Proc. AI and Statistics,* 2010.

[16] J.J. McAuley, T.S. Caetano, and M.S. Barbosa, "Graph Rigidity, Cyclic Belief Propagation and Point Pattern Matching," *IEEE Trans. Pattern Analysis and Machine Intelligence,* vol. 30, no. 11, pp. 2047-2054, Nov. 2008.

[17] A. Moffat and T. Takaoka, "An All Pairs Shortest Path Algorithm with Expected Time $O(n^2 \log n)$," *SIAM J. Computing,* vol. 16, no. 6, pp. 1023-1031, 1987.

[18] M.A. Paskin, "Thin Junction Tree Filters for Simultaneous Localization and Mapping," *Proc. Int'l Joint Conf. Artificial Intelligence,* 2003.

[19] U. Quasthoff, M. Richter, and C. Biemann, "Corpus Portal for Search in Monolingual Corpora," *Proc. Language Resources and Evaluation,* 2006.

[20] L. Sigal and M.J. Black, "Predicting 3D People from 2D Pictures," *Proc. Conf. Articulated Motion and Deformable Objects,* 2006.

[21] V. Strassen, "Gaussian Elimination Is Not Optimal," *Numerische Mathematik,* vol. 14, no. 3, pp. 354-356, 1969.

[22] C. Sutton and A. McCallum, "An Introduction to Conditional Random Fields for Relational Learning," *Introduction to Statistical Relational Learning,* L. Getoor and B. Taskar, eds., MIT Press, 2006.

[23] L.G. Valiant, "General Context-Free Recognition in Less than Cubic Time," *J. Computer and System Sciences,* vol. 10, pp. 308-315, 1975.

[24] J.S. Yedidia, W.T. Freeman, and Y. Weiss, "Generalized Belief Propagation," *Proc. Neural Information Processing Systems,* 2000.