Journal of Machine Learning Research 22 (2021) 1-24 Submitted 8/17; Revised 4/21; Published 5/21

On Solving Probabilistic Linear Diophantine Equations

Patrick Kreitzberg PATRICK.KREITZBERG@UMONTANA.EDU
Department of Mathematics
University of Montana
Missoula, MT 59812-0003, USA

Oliver Serang OLIVER.SERANG@UMONTANA.EDU
Department of Computer Science
University of Montana
Missoula, MT 59812-0003, USA

Editor: Amos Storkey

Abstract

Multiple methods exist for computing marginals involving a linear Diophantine constraint on random variables. Each of these extant methods has some limitation on the dimension and support or on the type of marginal computed (*e.g.*, sum-product inference, max-product inference, *maximum a posteriori*, *etc.*). Here, we introduce the "trimmed p-convolution tree" an approach that generalizes the applicability of the existing methods and achieves a runtime within a log-factor or better compared to the best existing methods. A second form of trimming we call underflow/overflow trimming is introduced which aggregates events which land outside the supports for a random variable into the nearest support. Trimmed p-convolution trees with and without underflow/overflow trimming are used in different protein inference models. Then two different methods of approximating max-convolution using Cartesian product trees are introduced.

Keywords: Bayesian inference, convolution, L_p space, noisy-or, max-convolution, loopy belief propagation, sum-product inference, max-product inference, Cartesian product

1. Introduction

Probabilistic linear Diophantine equations are useful for efficient message passing in Bayesian graphical models: Pearl's belief propagation (BP) (Pearl, 1982) computes exact posteriors when the graph is a tree by passing a number of messages linear in the number of nodes. When G is not a tree, "loopy belief propagation" (LBP) heuristically estimates posteriors by performing BP on subtrees of G.

Belief propagation is a message-passing algorithm which is able to find the exact marginals when the graph formed is a tree (sum-product inference solves for the marginal whereas max-product inference solves for the max-marginal). Message passing is how nodes in a graphical model share information with each other. LBP has been shown to perform well in practice (Murphy et al., 1999) and to have desirable theoretical properties. For instance, when a graph contains exactly one loop, LBP has been shown to converge in a number of messages linear in the number of nodes. "Turbo codes," (McEliece et al., 1998) the first forward error correction scheme to approach the Shannon coding limit (Berrou et al., 1993), were shown to be a form of LBP.

©2021 Patrick Kreitzberg and Oliver Serang.

License: CC-BY 4.0, see https://creativecommons.org/licenses/by/4.0/. Attribution requirements are provided at http://jmlr.org/papers/v22/17-474.html.

However, both BP and LBP do not guarantee the efficient computation of a single message. For example, the constraint $Y = X_1 + X_2 + \cdots + X_m$ on random variables $Y, X_1, X_2, \ldots, X_m$ can be encoded with an $m+1$-dimensional table. Adaptive Belief Propagation performs efficient message passing in trees and Gaussian loopy graphs (Papachristoudis and Fisher, 2015). When a marginal is updated, the message sent between the updated and the query node goes only through their lowest common ancestor.

Linear Diophantine equations are of the form $y = a_1 \cdot x_1 + a_2 \cdot x_2 + \cdots + a_{m-1} \cdot x_{m-1} + a_m \cdot x_m$, where all $a_i \in \mathbb{Z}$ and x_i are unknown integers for which we solve. In the probabilistic generalization of the linear Diophantine equation, $y, x_1, x_2, \ldots, x_m$ are no longer integers, but are random variables with discrete probability mass functions (PMFs): $\text{pmf}_Y, \text{pmf}_{X_1}, \text{pmf}_{X_2}, \ldots, \text{pmf}_{X_m}$. From these PMFs and the constraint $Y = X_1 + X_2 + \cdots + X_{m-1} + X_m$, all marginal, or max-marginal, distributions on $Y, X_1, \ldots, X_m$ may be computed. Note that a_i are no longer necessary, since each PMF can take multiple values; for example, a PMF on X_i with support at $\{0, 3, 6, 9, \ldots\}$ would be equivalent to using $a_i = 3$.

There have been several specialized methods developed for efficiently solving certain cases of probabilistic linear Diophantine equations. HOP-MAP (Tarlow et al., 2010) calculates the max-marginals of all random variables when $X_1, X_2, \ldots, X_m \in \{0, 1\}, Y \in \{0, \ldots, m\}$ and noisy-or calculates the marginals under a different operator than standard addition and requires $X_1, X_2, \ldots, X_m \in \{0, 1\}$ as well as $Y \in \{0, 1\}$. When all $X_1, X_2, \ldots, X_m$ are binary inputs, $Y = X_1, X_2, \ldots, X_m$ can be used to calculate the reliability of a k-out-of-n:G system. The system has n operations, each of which may be good or bad and the system as a whole is considered good if at least k operations are good. The G denotes that the system is good if k operations are good. An F denotes that a k-out-of-n:F system fails if k of the n operations fail. Barlow and Heidtmann provide an algorithm which calculates the probabilities that J operations are good; the algorithm trims the output by not calculating probabilities for all J based on the values of k and n (Barlow and Heidtmann, 1984). An extended problem, k-to-ℓ-out-of-n:G, determines the system is good if at least k and no more than ℓ operations are good. Belfore (1995) introduced a convolution tree approach to calculate the reliability of the system which uses only a forward pass to calculate the posterior on Y. Due to the nature of the problem, the leaves are one-dimensional, binary variables.

Similar to Belfore, Tarlow et al. (2012) create a convolution tree which is limited to one-dimensional, binary input variables; however, unlike Belfore's algorithm, Tarlow *et* al. can perform a backward pass to calculate the marginals on $X_1, X_2, \ldots, X_m$. On the backward pass, the convolutions are performed based on the supports generated on the interior nodes during the forward pass. Discovered independently of Belfore and Tarlow *et* al., Serang invented convolution trees with a more general input where the leaves are without limit on their dimension or supports (Serang, 2014).

This paper focuses on methods for efficiently finding the marginals computed by the probabilistic generalization of linear Diophantine equations. Specifically, given PMFs on $X_1, X_2, \ldots, X_m$ with support $\subseteq \{0, 1, 2, \ldots, n-1\}$ and a PMF on Y with support $\subseteq \{0, 1, 2, \ldots, m \cdot n - 1\}$ where $Y = X_1 + X_2 + \cdots + X_{m-1} + X_m$, we present methods which compute all marginals $\text{pmf}_{Y|X_1,X_2,\ldots}$ and $\text{pmf}_{X_i|Y,X_{j\neq i}}$ $\forall X_i$.

1.1 HOP-MAP

HOP-MAP is an efficient method for calculating the max-marginals on all $Y, X_1, \ldots, X_m$ when $X_i \in \{0, 1\}\ \forall i$ and $Y \in \{0, \ldots, m\}$. The method takes advantage of sorting variables in descending order of their probability of being 1. Let $X_{1'}, X_{m'}, \ldots, X_{m'}$ be such a sorting. Then, the most probable way in which $Y = 1$ is to have $X_{1'} = 1, X_{2'} = 0, X_{3'} = 0, \cdots, X_{m'} = 0$ since setting any other X_i to 1 replaces $X_{1'}$ which is the most probable to be 1. Similarly, the most probable way in which $Y = 2$ is to have $X_{1'} = 1, X_{2'} = 1, X_{3'} = 0, X_{4'} = 0, \cdots, X_{m'} = 0$, and so on.

In two linear passes, compute and cache the cumulative products $C_1(i) = \prod_{j=1}^{j=i} Pr(X_{i'} = 1)$ and $C_0(i) = \prod_{j=i+1}^{j=m} Pr(X_{i'} = 0)$. Then the max-marginal on Y is $\max_{Y=i, X_1, \ldots, X_m}(Pr(Y = i, X_1 = x_1, \ldots)) = Pr(Y = i) \cdot C_1(i) \cdot C_0(i)$.

To calculate the max-marginals on $X_{1'}, \ldots, X_{m'}$, HOP-MAP relies on the fact that the most probable state of $X_{i'} = 0, Y = y$ and $X_{i'} = 1, Y = y$ is either the same or very close to the max-marginal on $Y = y$. To calculate the max-marginals, in two linear passes compute and cache the cumulative maxes of the cumulative products, $M_\ell(i) = \max(C_1(0) \cdot C_0(0), C_1(1) \cdot C_0(1), \ldots, C_1(i) \cdot C_0(i))$ and $M_r(i) = Pr(Y = i) \cdot \max(C_1(m) \cdot C_0(m), C_1(m-1) \cdot C_0(m-1), \ldots, C_1(i) \cdot C_0(i))$. Then the max-marginal for $X_{i'} = 0$ is $Pr(Y = i) \cdot \max(M_\ell(i), M_r(i+1) \cdot \frac{Pr(X_{i'}=0)}{Pr(X_{i'}=1)})$ and $X_{i'} = 1$ is $Pr(Y = i) \cdot \max(M_\ell(i-1) \cdot \frac{Pr(X_{i'}=1)}{Pr(X_{i'}=0)}, M_r(i))$. The reason for the $\frac{Pr(X_{i'}=0)}{Pr(X_{i'}=1)}$ term is because the maximum configuration for $\{X_{i'} = 0, Y = y\}$ when $y \geq i$ is the maximum configuration for $Y = y$ but with $X_{i'}$ and $X_{(y+1)'}$ switched. Similarly, the reason for the $\frac{Pr(X_{i'}=1)}{Pr(X_{i'}=0)}$ term is because the maximum configuration for $\{X_{i'} = 1, Y = y\}$ when $y < i$ is the maximum configuration for $Y = y$ but with $X_{i'}$ and $X_{y'}$ switched. HOP-MAP sorts the variables and then does a constant number of linear passes so it is $\in O(m \log(m))$.

1.2 Noisy-or

The noisy-or operator is denoted $Y = X_1 \texttt{ or } X_2 \texttt{ or} \cdots \texttt{or } X_m$ and requires $Y, X_1, X_2, \ldots, X_m$ to have support $\in \{0, 1\}$. The noisy-or operator on two variables, $C = A \texttt{ or } B$, aggregates the event $Pr(A = 1, B = 1)$ into the outcome $C = 1$. A noisy-or operator may be used when several different events have an independent chance of "turning on" an outcome (Morris, 2011). This has been used in protein inference where the presence of a protein may be turned on independently by the presence of many different peptides (Serang et al., 2010).

The noisy-or operator is associative ($(X \texttt{ or } Y) \texttt{ or } Z = X \texttt{ or } (Y \texttt{ or } Z) = 0$ if $X = Y = Z = 0, 1$ else) and therefore solving problems of the form $Y = X_1 \texttt{ or } X_2 \texttt{ or } \cdots \texttt{or } X_m$ may be done efficiently by building a tree similar to a convolution tree but instead of $C = A + B$ for each triplet, the relationship is $C = A \texttt{ or } B$.

In a forward pass, the prior on all internal nodes may calculated by solving $\uparrow C = \uparrow A \texttt{ or } \uparrow B$, where $\uparrow C$ is the prior on variable C and $\downarrow C$ is the likelihood. The outcome $\uparrow C = 0$ is the event that both $\uparrow A = 0$ and $\uparrow B = 0$, the outcome $\uparrow C = 1$ is the product of all other events: $Pr(\uparrow C = 0) = Pr(\uparrow A = 0, \uparrow B = 0)$, $Pr(\uparrow C = 1) = Pr(\uparrow A = 1, \uparrow B = 0) \cdot Pr(\uparrow A = 0, \uparrow B = 1) \cdot Pr(\uparrow A = 1, \uparrow B = 1)$.

In the backward pass, the likelihood on all interior nodes may be calculated. The backward pass can be done by negating the PMF of the sibling node then solving for

$\downarrow A = \downarrow C$ `or` $(- \uparrow B)$. The outcome $\downarrow A = 0$ is the product of events that all variables are equal to 0 plus the event that $\downarrow C$ is turned on by $\uparrow B$ and not $\downarrow A$: $Pr(\downarrow A = 0) = Pr(\downarrow C = 0, \uparrow B = 0) \cdot Pr(\downarrow C = 1, \uparrow B = 1)$. The event that $\downarrow A = 1$ is the product of the events when $\downarrow C$ is turned on solely by $\downarrow A$ and when all variables are 1: $Pr(\downarrow A = 1) = Pr(\downarrow C = 1, \uparrow B = 0) \cdot Pr(\downarrow C = 1, \uparrow B = 1)$. After a full forward and backward pass, all priors and likelihoods may be calculated for the root, leaves, and all internal nodes. Since the work done at each node is constant, and there are $\Theta(m)$ nodes, all messages out may be solved $\in \Theta(m)$ time.

1.3 Probabilistic p-convolution Trees

Probabilistic convolution trees solve $Y = X_1 + \cdots + X_m$ where n and the dimension of the variables, d can take any value. They can be used as a node in a graphical model which can calculate the priors and likelihoods of $Y, X_1, \ldots, X_m$. The $X_1, \ldots, X_m$ are given priors and Y is given a likelihood, these are the messages in. The messages out are likelihoods for $X_1, \ldots, X_m$ and the prior on Y. The prior distribution on Y may be considered the marginal on Y, similarly the likelihood distribution on $X_1, \ldots, X_m$ may be considered their marginal.

Probabilistic convolution trees work by turning the addition $Y = X_1 + \cdots + X_m$ into a binary tree where the leaves are $X_1, \ldots, X_m$, the root is Y and the parent of two children is the sum of their random variables. For example if $Y = X_1 + X_2 + X_3 + X_4$, then a probabilistic convolution tree will create the interior nodes $(X_1 + X_2)$ and $(X_3 + X_4)$. Y is solved by the addition of the two interior nodes: $Y = (X_1 + X_2) + (X_3 + X_4)$. The tree does one forward pass (leaves to root) to calculate all priors on the interior nodes and the root and a backward pass to calculate all likelihoods on the interior nodes and the leaves. When asked, the backward pass marginalizes out all variables except the one of interest. The key to its efficiency is that the addition of two random variables corresponds to the convolution of the probability mass functions which can be performed in subquadratic time via the fast Fourier transform (FFT) (Cooley and Tukey, 1965). For $d = 1$, all priors and likelihoods (and therefore, posteriors, which are the products of priors and likelihoods) can be computed simultaneously $\in O(m \cdot n \log(m \cdot n) \log(m))$. If $d > 1$ the runtime is increased to $\Theta(m^d \cdot n^d \cdot \log(m^d \cdot n^d))$.

Various different p-norms may be used to compute or approximate (depending on the p desired) a continuum between sum-product inference ($p = 1$) and max-product inference ($p = \infty$) (Pfeuffer and Serang, 2016). Since a constant number of p-norms are used, the theoretical runtime is not affected by the value of p.

Here the method of "trimmed p-convolution trees" and the lazy variant (efficient for online processing with LBP) are presented. The methods for trimming are then modified to efficiently generalize noisy-or problems.

1.4 Cartesian Product Trees

k-selection on $X_1 + X_2 + \cdots + X_m$ returns the top k values of the Cartesian product $X_1 + X_2 + \cdots + X_m$. If $Y = X_1 + X_2 + \cdots + X_m$ and $Y, X_1, X_2, \ldots, X_m$ are discrete random variables, then k-selection on $X_1 + X_2 + \cdots + X_m$ can be used to solve for the max-marginals on $Y, X_1, X_2, \ldots, X_m$; however, since $k \in \{1, 2, \ldots, n^m\}$ (and retrieving the top k values is

at least linear in k) solving max-convolution exactly using a k-selection may be exponential in m. For example, if $X_1, X_2, \ldots, X_m$ are all binary and $Y \in 0, 1, \ldots, m$ then there is only one way for Y to be 0: $X_1 = X_2 = \cdots = X_m = 0$. If Y_0 is the least probable outcome in $X_1 + X_2 + \cdots + X_m$ then to retrieve all values in Y, the selection must have $k = n^m$.

There are several optimal methods which perform optimal selection on $X_1 + X_2$ in $O(n + k)$ time (Frederickson, 1993; Kaplan et al., 2019; Serang, 2021). Serang's method segments the data into layers of exponentially increasing size where all values in a layer are at least as good as all values in the following layers. The method finds which layer products (the Cartesian product of a layer in X_1 with a layer in X_2) may contain values which will be in the resulting k-selection. All values in the candidate layer products are generated and the output is trimmed with a linear time one-dimensional selection.

A Cartesian product tree (CPT) is a balanced, binary tree where each leaf node represents one of the input variables X_i and each interior node solves a pairwise selection problem using Serang's method (Kreitzberg et al., 2021). Similar to the convolution tree, the output of the root is Y, but in this case the root performs pairwise selection on $(X_1 + X_2 + \cdots + X_{\frac{m}{2}}) + (X_{\frac{m}{2}+1} + X_{\frac{m}{2}+2} + \cdots + X_m)$. CPTs have been proven to be fast in practice and are utilized in `NeutronStar`, the world's fastest isotopologue calculator (Kreitzberg et al., 2020b). A similar algorithm to the CPT may be implemented using soft heaps (Chazelle, 2000); however, soft heaps have poor performance in practice (Kreitzberg et al., 2020a). CPTs can generate the top k values of the form $X_1 + X_2 + \cdots + X_m \in O\left(m \cdot \left(n \log\left(\frac{n}{n\cdot(\alpha-1)+1}\right) + \frac{n\cdot\alpha\cdot\log(\alpha)}{\alpha-1}\right) + k \cdot m^{\log(\alpha^2)}\right)$ time for constant $\alpha \in (1, 2)$, *e.g* $\in O(m\cdot n + k\cdot m^{0.141\cdots})$ for $\alpha = 1.05$. CPTs rely on the use of layer-ordered heaps to retrieve the top k values without having to sort the data (thus having runtime $\in o(n \log(n))$). An array may be layer-order heapified $\in \Theta\left(n \log\left(\frac{n}{n\cdot(\alpha-1)+1}\right) + \frac{n\cdot\alpha\cdot\log(\alpha)}{\alpha-1}\right)$ (Pennington et al., 2020). If requested, a CPT may produce the individual indices for each value in the k-selection, in which case there is an added $\Theta(k\cdot m)$ to the runtime to become $\Theta(n\cdot m + k\cdot m)$.

2. Methods

Here, we present the lazily trimmed p-convolution tree. We also introduce another method of trimming that imitates the noisy-or for $Y, X_1, \ldots, X_m \in \{0, 1\}$, which we call underflow/overflow trimming. Then, CPTs are used to perform fast, approximate max-convolution.

2.1 Trimmed p-convolution Trees

Consider $Y = X_1 + X_2 + X_3 + X_4$, where the priors on the X_i variables have support $X_1 \in \{0, 1, 2\}$, $X_2 \in \{0, 1\}$, $X_3 \in \{1, 2\}$, and $X_4 \in \{1, 2, 3\}$, and where the likelihood on Y has support $Y \in \{1, 2, 3\}$. The forward pass of the p-convolution tree algorithm will first compute priors on $X_1 + X_2$ and $X_3 + X_4$, then compute the prior on $Y = (X_1 + X_2) + (X_3 + X_4)$. Then the backward pass will compute the likelihoods on $X_1 + X_2$ and $X_3 + X_4$, and finally the likelihoods on X_1, X_2, X_3, and X_4. After both passes have been performed, all priors and likelihoods will be available, meaning that all posteriors can be computed. An example of the forward and backward passes may be seen in Figure 1.

As the forward pass progresses, the support of the distributions grows, leading to the prior on Y with support $Y \in \{2, 3, \ldots, 8\}$. In a large tree, the cost of this growing support

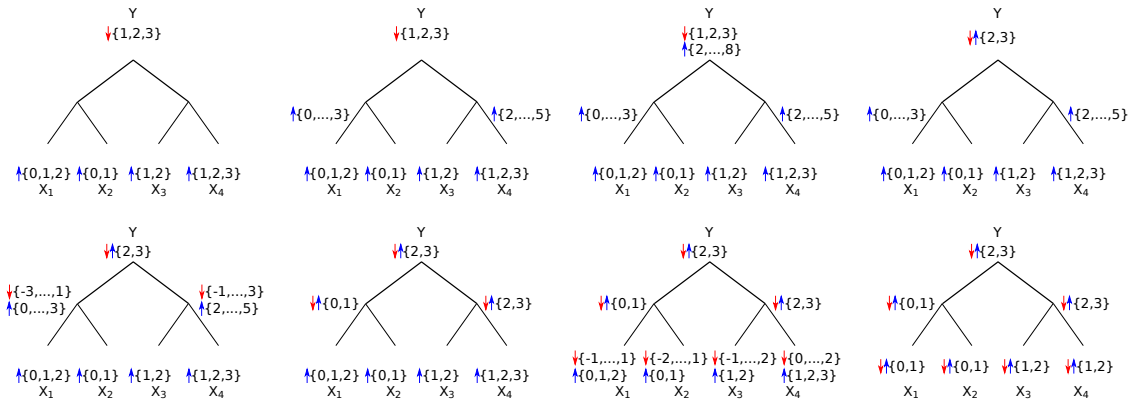

Figure 1: **Trimmed p-convolution tree.** The first forward and backward pass are illustrated, wherein the possible supports at each node are computed. Possible prior supports are labeled using blue up arrows and possible likelihood supports are labeled using red down arrows. When both supports are available, the intersection is labeled with both arrow types. Progressing left to right and then top down: **1:** a p-convolution tree immediately after construction with only supports of leaves and root known and no convolutions propagated. **2:** The forward pass computes the possible prior support of the second layer of the tree. **3:** The forward pass reaches the root node. **4:** The root node both possible supports available; the intersection is stored. **5:** The backward pass begins. **6:** As the backward pass progresses, internal nodes have both prior support and likelihood support known; the intersection is computed before propagating further. **7:** The possible likelihood supports of the inputs are now known. **8:** A bounding box of possible supports for each node in the tree is now known. Now, convolutions are propagated in a forward and backward pass. If the messages passed leads to a smaller intersection with the supports on the node then the new intersection is applied to both PMFs for the node.

is non-trivial because the cost of FFT convolution is super-linear. Moreover, in practice, the cache effects of storing several large distributions (rather than several distributions with trivial support such as $\{0, 1\}$) can be quite pronounced.

However, the likelihood on Y has support $Y \in \{1, 2, 3\}$; therefore, given the observed data, the event $Y = 8$, which is entertained by the prior on Y computed in the forward pass, is impossible. We seek to "trim" the distributions during processing to narrow their support so that only events in the intersection of the prior support and likelihood support are considered, avoiding unnecessarily large convolutions on the interior nodes.

To efficiently perform trimming in an online setting, four passes through the tree are performed instead of two. The first two passes are forward and backward passes that compute only the support of the prior at the given node and the support of the likelihood at the given node (and intersect these whenever either changes). This method only updates supports which are affected by the new information being passed in (the supports may only be trimmed, never expanded). The third pass is a forward pass which computes the priors on the interior nodes and the root and the fourth pass is a backward pass which computes the likelihoods on the interior nodes and the leaves, similar to the untrimmed p-convolution tree. This strategy costs $O(m)$ when updating all supports in a non-lazy manner.

2.2 Lazily Trimmed p-convolution Trees for Online Processing

To most efficiently enable a trimmed p-convolution tree to receive all relevant support information, it is best to not compute any convolutions until necessary (in case further information is received that will narrow the support). For this reason, cached supports and PMFs throughout the tree are recomputed only when a message out is requested (Figure 2).

The first message out will cost $\Omega(m)$ (because it must at least touch each node in the tree). In terms of convolutions, it will require a full forward pass and a partial backward pass along the path from the root to the node of interest. After the first message out, subsequent messages out will be significantly faster, having many nodes in the tree with up-to-date support and PMF information. Likewise, after all nodes in the tree are cached (in both directions), the first message sent into the tree will cost $\Theta(m)$ (because it must mark one direction on all but one node as not cached). This can be done in $\Theta(m)$ rather than $\Omega(m)$ since no convolutions will be performed (because the tree is lazy and only performs convolutions when a message out is requested). After the first message received, subsequent messages received will cost $O(\log(m))$, because they are guaranteed to reach the root in $O(\log(m))$ steps and then reverse direction, and there is at most one path down from the root that has not yet been marked (the path exactly opposite the path used to dirty the first message in).

Using the potential method of amortized time analysis, where ϕ counts the number of nodes cached (including both booleans for whether a prior is cached from below or a likelihood from above), it can be shown that updating the cache for t successive messages in will have runtime $\in O(m)$. This can be amortized $\in \widetilde{O}(1)$ by including it in the cost of constructing the tree (which costs $\Theta(m)$).

Identical reasoning (but where ϕ represents the count of nodes that are not cached rather than the number cached) can be used to show that the cost of updating the cache when t successive messages out are requested will likewise be $\in \widetilde{O}(1)$. Alternately sending and

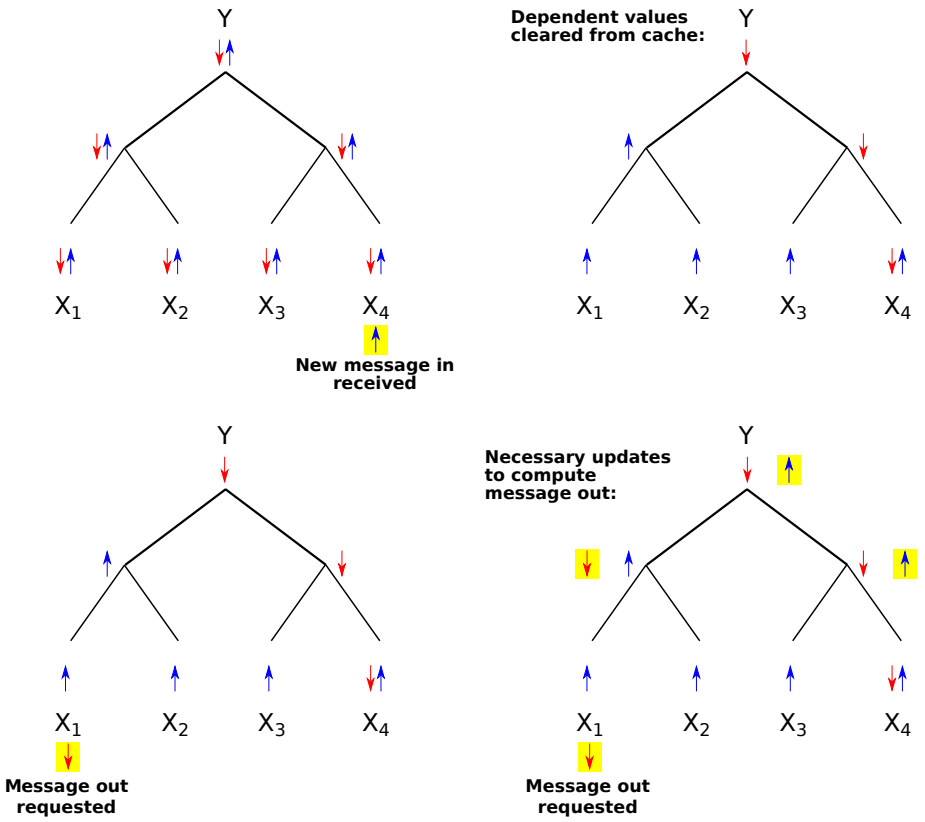

Figure 2: **Lazily, trimmed p-convolution tree.** Progressing left to right and top down: **1:** A p-convolution tree in which all internal nodes have computed their possible prior and likelihood supports as well as their prior and likelihood PMFs receives a new message in (an updated prior for X_4). **2:** Values depending on the prior of X_4 are dirtied in the cache to indicate that they are not current. This costs $O(m)$. But receiving a new prior on X_3 will now take only $O(1)$ steps, because the process of dirtying the cache can be terminated once another node with a dirty prior is reached. **3:** A message out (the likelihood of X_1) is requested. **4:** The nodes where either a prior or likelihood is requested are marked. These requests form a path for repairing the cache. This process does not need to visit every node; instead, in this case, it need only visit $O(\log(m))$ nodes.

receiving messages out is more complicated and would merit further investigation on its own. The balanced construction of probabilistic p-convolution trees means that the longest non-cyclic path between any nodes in the tree will be $\in O(\log(m))$, which would likely benefit the worst-case amortized or average analysis.

In addition to the faster runtimes, trimmed p-convolution trees have the added benefit of greater accuracy. One reason for this is that shorter FFT convolutions (which are used when $p = 1$ and as part of numeric p-convolution when $p > 1$) grow slightly less accurate as the size of the tensors grows (Pfeuffer and Serang, 2016). This is because the implementation of p-convolution relies on the numeric approach (via FFTs) on long tensors, but the naïve approach (especially when implemented in TRIOT, a method of efficiently iterating over multidimensional arrays, created by Heyl and Serang, 2017) is faster on small problems and also achieves the exact result (rather than a numeric approximation).

2.3 Time Analysis of Trimmed p-convolution Trees

For a p-convolution tree with m input variables, each with support of size n, there will be $\log(m)$ many layers, indexed by ℓ. The layer above the leaves preform convolutions on inputs of size n, for each layer up the tree, the size of the inputs for the convolution is roughly doubled, if there is no opportunity to trim the supports. The amount of convolutions per layer will be half of the layer below, starting with $m/2$ convolutions in the second layer. Performing a d-dimensional FFT, where each dimension has length n, can be done $\in O(n^d \log(n^d))$ time using an algorithm like Cooley-Tukey. The runtime for a lazily trimmed p-convolution tree on m inputs of size n and dimension d is as follows:

$$
\begin{aligned}
r(m,n,d) =& \sum_{\ell=1}^{\log(m)} \frac{m}{2^{\ell-1}} (2^{\ell-1}\cdot n)^d \cdot \log((2^{\ell-1}\cdot n)^d) \propto \sum_{\ell=1}^{\log(m)} \frac{m}{2^{\ell}} 2^{\ell\cdot d}\cdot n^d \cdot \log(2^{\ell\cdot d}\cdot n^d) \\
=& m\cdot n^d \left(\sum_{\ell=1}^{\log(m)} 2^{\ell(d-1)} \cdot \log(2^{\ell\cdot d}) + \log(n^d)\cdot \sum_{\ell=1}^{\log(m)} 2^{\ell(d-1)} \right) \\
=& \begin{cases}
= m\cdot n \left(\sum_{\ell=1}^{\log(m)} \log(2^\ell) + \log(n)\cdot \sum_{\ell=1}^{\log(m)} 1 \right) & \text{if } d = 1 \\
= m\cdot n\cdot (\log(m)+\log(n))\log(m) & \\
= m\cdot n^d \left(d\cdot \sum_{\ell=1}^{\log(m)} 2^{\ell(d-1)}\cdot \ell + \log(n^d)\cdot \sum_{\ell=1}^{\log(m)} 2^{\ell(d-1)} \right) & \text{if } d > 1 \\
= m\cdot n^d \cdot \left(m^{d-1}\log(m^d) + \log(n^d)\cdot m^{d-1} \right) &
\end{cases}
\end{aligned}
$$

therefore,

$$
r(m,n,d) \in \Theta\left(m^d \cdot n^d \cdot \log(m^d\cdot n^d) \cdot \begin{cases} \log(m), & d=1 \\ 1, & \text{else} \end{cases} \right).
$$

This runtime is derived for the worst case scenario where no trimming of the supports may be performed. In the best case, where the support at all nodes are of constant size,

all FFTs are performed in constant time and so the runtime is $\in \Theta(m)$ since every node in the tree has to be visited but the time spent at each node is constant. Note that $\Theta(m \cdot n \cdot \log(m \cdot n) \cdot \log(m) \subset O((m \cdot n)^{1+\epsilon})$ for any $\epsilon > 0$; in comparison, constructing the m prior PMFs of size n would take runtime $O(m \cdot n)$, meaning the p-convolution tree algorithm is not very much more difficult than loading the data. The runtime of trimmed p-convolution trees and all other methods are listed in Table 1.

2.4 Generalizing the Noisy-or for Use in Trimmed p-convolution Trees

Here, we introduce a method of trimming p-convolution trees which can imitate the noisy-or operator. Then, we introduce underflow/overflow trimming which generalizes this trimming method to random variables that are not necessarily binary.

A noisy-or can be implemented in a p-convolution tree by doing some small post-processing after the convolution on the forward and backward passes. On the forward pass the convolution calculates $\uparrow \hat{C} = \uparrow A \circledast \uparrow B$. The outcome $\uparrow \hat{C} = 0$ contains the event $\uparrow A_0 \cdot \uparrow B_0$ and the outcome $\uparrow \hat{C} = 1$ contains the events $\uparrow A_1 \cdot \uparrow B_0$ and $\uparrow A_0 \cdot \uparrow B_1$. The only difference between this and the noisy-or is that in the noisy-or the outcome $\uparrow C = 1$ also contains the event $\uparrow A_1 \cdot \uparrow B_1$. In order to trim $\uparrow \hat{C}$ to be equivalent to $\uparrow C = \uparrow A$ `or` $\uparrow B$, simply remove the outcome $\uparrow \hat{C} = 2$ and add the event $\uparrow A_1 \cdot \uparrow B_1$ to the outcome $\uparrow \hat{C} = 1$. Since no events where created or destroyed, $\uparrow \hat{C}$ does not need to be renormalized to be equivalent to $\uparrow C$.

On the backward pass, $\downarrow \hat{A} = \downarrow C \circledast (- \uparrow B)$ has three outcomes, $\downarrow \hat{A} = -1, \downarrow \hat{A} = 0$, and $\downarrow \hat{A} = 1$. The outcome $\downarrow \hat{A} = -1$ can be trimmed because the random variables are binary. The outcome $\downarrow \hat{A} = 0$ contains the events $\downarrow C_0 \cdot \uparrow B_0$ and $\downarrow C_1 \cdot \uparrow B_1$, which is the same as the noisy-or. The only difference is that $\downarrow \hat{A} = 1$ contains only the event $\downarrow C_1 \cdot \uparrow B_0$ but it should also contain $\downarrow C_1 \cdot \uparrow B_0$. This extra event, $\downarrow C_1 \cdot \uparrow B_0$, may be calculated directly and added to the outcome $\downarrow \hat{A} = 1$; however, first $\downarrow \hat{A}$ needs to be multiplied by its scaling factor which may be cached at construction. After the extra event is added to the outcome $\downarrow \hat{A} = 1$, normalize $\downarrow \hat{A}$ and it is now equivalent to $\downarrow A = \downarrow C$ `or` $(- \uparrow B)$.

In this scenario, where all of $Y, X_1, \ldots, X_m \in \{0, 1\}$, optimal trimming may occur meaning the supports on all nodes in the tree are $\in \{0, 1\}$. This means the work done at each node in the tree is equal and therefore noisy-or may be implemented $\in \Theta(m)$ time.

2.5 Underflow/overflow Trimming

Underflow/overflow trimming is a generalization of the noisy-or for cases in which $Y, X_1, X_2, \ldots, X_m$ may have any support, not just $\in \{0, 1\}$. Similar to the case of the classic noisy-or, any events which land outside the maximum support allowed are aggregated back into the maximum support; however, this may also include events which land below the minimum support which will than be aggregated into the minimum support. Note that this generalization of the noisy-or is different than generalizing the noisy-or as a max operator on the random variable arguments, which is not easily achieved using convolution.

Underflow/overflow trimming for non-binary supports may be useful when building a cardinal model based on fluctuations in the stock market. If the model is based on the change in the price of a stock, $Pr(Z_{t+1} = b | Z_t = a) = f(b - a)$, then it may be possible that events in the model cause the stock price to fall below \$0. Although it is impossible for a

stock to have a negative value, this event should still be taken into account by aggregating all events in which the stock price falls below \$0 into the event that the stock price is equal to \$0.

Let $\underline{h_\ell}$ and $\overline{h_\ell}$ be the minimum and maximum supports of some PMF X in dimension ℓ, respectively. For each dimension, ℓ, X will have an overflow $(d-1)$-hyperplane formed by outcomes $X_{h_1,h_2,\ldots,\overline{h_\ell},\ldots}$ where h_ℓ is always equal to the maximum support in dimension ℓ and all other indices are at least their minimum and at most their maximum supports in their respective dimensions. Overflow trimming seeks to aggregate any events which land outside the maximum support in dimension ℓ to the nearest outcome in this $(d-1)$-hyperplane. Similarly, for each dimension, ℓ, X will have an underflow $(d-1)$-hyperplane and underflow trimming seeks to aggregate any events which land below the minimum support in dimension ℓ to the nearest outcome in this $(d-1)$-hyperplane

These $(d-1)$-hyperplanes form a fringe around an enclosed space of all supported outcomes, we call the fringe and the enclosed space the interior. The exterior contains all outcomes which land outside the minimum or maximum supports for X.

2.5.1 Forward Pass

During the forward and backward passes, and any subsequent trimming, p is assumed to be constant, and not necessarily equal to 1 as is the case with the classic noisy-or. On the forward pass, $\uparrow C$ is calculated by convolving $\uparrow A$ with $\uparrow B$: $\uparrow \hat{C} = \uparrow A \circledast \uparrow B$, and then performing underflow/overflow trimming on $\uparrow \hat{C}$. The convolution may produce outcomes $\uparrow \hat{C}_{i_1,i_2,\ldots} = \uparrow \hat{C}_{j_1+k_1,j_2+k_2,\ldots} = \uparrow A_{j_1,j_2,\ldots} \cdot \uparrow B_{k_1,k_2,\ldots}$ which land in the exterior of $\uparrow C$, w.l.o.g. let this happen in the first dimension so that $j_1 + k_1 > \overline{i_1}$.

Underflow/overflow trimming seeks to aggregate the outcome $\uparrow \hat{C}_{j_1+k_1>\overline{i_1},j_2+k_2,\ldots}$ into $\uparrow C_{\overline{i_1},j_2+k_2,\ldots}$. This can be done through the use of a function, $f(X, h_1, h_2, \ldots)$, which accepts a PMF X and an index $h_1, h_2, \ldots$ and returns the nearest support in the interior of X. Then, if $\uparrow A_{j_1,j_2,\ldots} \cdot \uparrow B_{k_1,k_2,\ldots}$ lands in the exterior of $\uparrow \hat{C}$, simply aggregate the event to outcome $\uparrow \hat{C}_{f(C,j_1+k_1,j_2+k_2,\ldots)} = \uparrow \hat{C}_{\overline{i_1},j_2+k_2,\ldots}$ during trimming. All underflow/overflow trimming on the forward pass may be done by a single iteration through all outcomes in $\uparrow \hat{C}$ and aggregating $\uparrow \hat{C}_{i_1,i_2,\ldots}$ into $\uparrow \hat{C}_{f(\uparrow C,i_1,i_2,\ldots)}$. After underflow/overflow trimming is complete, normalize $\uparrow \hat{C}$ to become $\uparrow C$.

2.5.2 Backward Pass

In the backward pass, the convolution calculates $\downarrow \hat{A} = \downarrow C \circledast (- \uparrow B)$. Ideally, for any event $\uparrow A_{j_1,j_2,\ldots} \cdot \uparrow B_{k_1,k_2,\ldots}$ calculated in the forward pass, the event $\downarrow C_{j_1+k_1,j_2+k_2,\ldots} \cdot \uparrow B_{-k_1,-k_2,\ldots}$ will be calculated in the backward pass. However, if the event $\uparrow C_{j_1+k_1,j_2+k_2,\ldots}$ is in the exterior of $\uparrow C$, then the event would have been aggregated to $\uparrow C_{f(\uparrow C,j_1+k_1,j_2+k_2,\ldots)}$ with $f(\uparrow C, j_1 + k_1, j_2 + k_2, \ldots) \neq j_1 + k_1, j_2 + k_2, \ldots$ due to underflow/overflow trimming. If $f(\uparrow C, j_1 + k_1, j_2 + k_2, \ldots) = \overline{i_1} < j_1 + k_1, j_2 + k_2, \ldots$ (*i.e.* there was overflow in the first dimension), then $\downarrow C \circledast (- \uparrow B)$ will not properly recover the event $\downarrow C_{f(\uparrow C,j_1+k_1,j_2+k_2,\ldots)} \cdot \uparrow B_{-k_1,-k_2,\ldots}$ in outcome $\downarrow A_{j_1,j_2,\ldots}$. The key to underflow/overflow trimming is to efficiently recover this information which is lost during the forward pass.

This information may be recovered efficiently using FFT; however, the convolution is not between $\downarrow C$ and $(- \uparrow B)$. Instead, create a new PMF C' with minimum supports

$\underline{i'_\ell} = \underline{j_\ell} + \underline{k_\ell}\ \forall \ell \in \{1, 2, \ldots, d\}$ and maximum supports $\overline{i'_\ell} = \overline{j_\ell} + \overline{k_\ell}\ \forall \ell \in \{1, 2, \ldots, d\}$ (these are the same supports as $\uparrow \hat{C}$ before trimming). For all indices $i'_1, i'_2, \ldots$ which are in the interior of $\uparrow C$, copy $\downarrow C_{i'_1, i'_2, \ldots}$ into $C'_{i'_1, i'_2, \ldots}$. For all indices $i'_1, i'_2, \ldots$ which are in the exterior of $\uparrow C$, copy $\downarrow C_{f(\uparrow C, i'_1, i'_2, \ldots)}$ into $C'_{i'_1, i'_2, \ldots}$. Now, convolve $\downarrow \hat{A} = C' \circledast (- \uparrow B)$.

For $\downarrow \hat{A} = C' \circledast (- \uparrow B)$ to correctly perform underflow/overflow trimming it has to do two things: calculate the same outcomes as $\downarrow \hat{A} =\downarrow C \circledast (- \uparrow B)$ and recover all events which caused either underflow or overflow in the forward pass. Because the supports for C' are a superset of the supports for $\downarrow C$ and any value in $\downarrow C$ was copied into C', all outcomes calculated by $\downarrow \hat{A} =\downarrow C \circledast (- \uparrow B)$ will be calculated by $\downarrow \hat{A} = C' \circledast (- \uparrow B)$ with the same resulting value.

Let $\uparrow A_{j_1, j_2, \ldots} \cdot \uparrow B_{k_1, k_2, \ldots}$ be an event which landed in an outcome in the exterior of $\uparrow C$, $\uparrow C_{j_1+k_1, j_2+k_2, \ldots}$, on the forward pass. Then, in the backward pass $\downarrow \hat{A}_{j_1, j_2, \ldots}$ will contain the event $C'_{j_1+k_1, j_2+k_2, \ldots} \cdot \uparrow B_{-k_1, -k_2, \ldots} =\downarrow C_{f(\uparrow C, j_1+k_1, j_2+k_2, \ldots)} \cdot \uparrow B_{-k_1, -k_2, \ldots}$ as long as $C'_{j_1+k_1, j_2+k_2, \ldots} \cdot \uparrow B_{-k_1, -k_2, \ldots}$ occurs in the convolution $C' \circledast (- \uparrow B)$. Since $\uparrow A_{j_1, j_2, \ldots} \cdot \uparrow B_{k_1, k_2, \ldots}$ occurred in the forward pass, $\underline{j_\ell} \leq j_\ell \leq \overline{j_\ell}$ and $\underline{k_\ell} \leq k_\ell \leq \overline{k_\ell}\ \forall \ell \in \{1, 2, \ldots, d\}$. This means $j_\ell + k_\ell\ \forall \ell \in \{1, 2, \ldots, d\}$ is in bounds for C' and so the event $C'_{j_1+k_1, j_2+k_2, \ldots} \cdot \uparrow B_{-k_1, -k_2, \ldots}$ will occur in the convolution $C' \circledast (- \uparrow B)$. After the convolution $\downarrow \hat{A} = C' \circledast (- \uparrow B)$, $\downarrow \hat{A}$ should be trimmed to its proper supports and normalized to become $\downarrow A$.

The number of events which land in the exterior of $\uparrow C$ is $\in O(n^d)$. During trimming on the forward pass, after the convolution is done, each outcome is touched at most once. The operation to move events from one outcome to another takes constant time, however, calculating the destination uses the function $f(X, h_1, h_2, \ldots)$ which takes $\Theta(d)$ time. Therefore, trimming on the forward pass is $\in O(d \cdot n^d)$.

During the backward pass, when constructing C', all outcomes are touched exactly once. The function $f(X, h_1, h_2, \ldots)$ is used to calculate the nearest support in the interior of X, once this has been calculated there are a constant number of constant-time operations done. The function $f(X, h_1, h_2, \ldots)$ takes time $\in \Theta(d)$ and so constructing C' takes time $\in O(d \cdot n^d)$. The convolutions on either pass takes time $\in O(n^d \log(n^d)) = O(d \cdot n^d \log(n))$ so underflow/overflow trimming is free compared to the cost of the FFT.

2.6 Approximating Max-convolution Trees with CPTs

CPTs are used to approximate max-convolution by performing selection on the Cartesian product $Y = X_1 + X_2 + \cdots + X_m$. Each X_i is an array of log probabilities such that $Pr(X_i = j)$ is the exponential of the value in the j^{th} entry in X_i. A $k = 1$ selection on the Cartesian product $Y = X_1 + X_2 + \cdots + X_m$ will produce the most likely outcome in Y. A $k = 2$ selection will produce the two most likely outcomes, etc. Note that multiple events in a k-selection may land at the same support in Y.

2.6.1 A Straight-forward Approach

A standard k-selection may be performed to get approximate values on the max-marginals of all $Y, X_1, X_2, \ldots, X_m$. The value of k will have a great impact on both the runtime and the accuracy of the approximate max-marginals as there may be multiple values in the top k which land at the same outcome in Y. If k is too small, the runtime will be fast

Method	Applicability	Theoretical runtime
Brute force	Universal	$\Theta(m \cdot n^m)$
HOP-MAP	$d = 1, n = 2, p = \infty$	$\Theta(m \cdot \log(m))$
Noisy-or	$d = 1, n = 2, p = 1$	$\Theta(m)$
CPT	$d = 1, p = \infty$ marginal on Y, single X_i	$O\left(m \cdot \left(n \log\left(\frac{n}{n\cdot(\alpha-1)+1}\right) + \frac{n\cdot\alpha\cdot\log(\alpha)}{\alpha-1}\right) + k \cdot m^{\log(\alpha^2)}\right)$
CPT with indices	$d = 1, p = \infty$	$O\left(m \cdot \left(n \log\left(\frac{n}{n\cdot(\alpha-1)+1}\right) + \frac{n\cdot\alpha\cdot\log(\alpha)}{\alpha-1}\right) + k \cdot m\right)$
p-convolution tree	$d = 1$	$\Theta(m \cdot n \cdot \log(m \cdot n) \log(m))$
Trimmed p-convolution tree	$d = 1$	$\in [\Theta(m \cdot n \cdot \log(n)),$ $\Theta(m \cdot n \cdot \log(m \cdot n) \log(m))]$
Trimmed p-convolution tree	$d > 1$	$\in [\Theta(m^d \cdot n^d \cdot \log(n^d)),$ $\Theta(m^d \cdot n^d \cdot \log(m^d \cdot n^d))]$

Table 1: **Theoretical runtimes of all methods considered in this manuscript.** The "Applicability" column describes the requirements the specialized methods put on the data. With these constraints in mind, the trimmed p-convolution tree method is always within a log-factor of the specialized methods. For trimmed p-convolution trees to achieve their fastest runtimes, if X_i has support $X_i \in \{t_1, t_2, \ldots, t_n\}$ for all i, then Y must have support $Y \in \{t_1, t_2, \ldots, t_n\}$. This allows the support on all interior nodes to be constant, thus the convolutions do not grow in size. For CPTs, α is a constant $\in (1, 2)$; the optimal α is not yet known.

but the outcomes in Y may not be saturated. If k is too large, the result may be exact max-convolution but the selection may take $k = n^m$ time. The user may wish to select k based either on a probability mass threshold or desired runtime. If a desired runtime is provided, CPTs may continuously produce output until the runtime limit is reached (either by repeatedly producing the next k values or the next full layer of values). If a probability mass threshold is provided, values may be produced until the threshold is hit after which the remaining outcomes which have not been touched may be assigned a probability of zero or a value based on the threshold and number of empty outcomes.

2.6.2 Withholding X_i from the Selection to Help Saturate Outcomes in Y

To be able to keep a small k (*i.e.* sub-exponential) while producing more outcomes in Y, one can hold out any X_i and perform a k_1-selection on $X_1 + \cdots + X_{i-1} + X_{i+1} + \cdots + X_m$. Then, select the top k_2 values in X_i and perform the Cartesian product on the top k_1 values in $X_1 + \cdots + X_{i-1} + X_{i+1} + \cdots + X_m$ and top the k_2 values in X_i. This will shift all outcomes from the k_1-selection on $X_1 + \cdots + X_{i-1} + X_{i+1} + \cdots + X_m$ by the indices in the k_2-selection on X_i, thus greatly increasing the saturation of outcomes in Y. If $k_1 \cdot k_2 = k$ then this takes no longer than the standard k-selection. The runtime of CPTs, along with all other methods, is listed in Table 1.

3. Results

All programs for benchmarking results are written in `C++` and compiled with `g++` version 7.4.0 using compiler options `-std=c++11 -O3 -march=native -mtune=native`. p-convolution

m	Y Support	Trimmed Time(s)	Untrimmed Time(s)
2	$\{0, 1\}$	0.0001700	0.0001660
4	$\{0, 1, 2, 3\}$	0.0002530	0.0002450
8	$\{0, \ldots, 7\}$	0.0002870	0.0002790
16	$\{0, \ldots, 15\}$	0.0005620	0.0006090
32	$\{0, \ldots, 31\}$	0.0011250	0.001815
64	$\{0, \ldots, 63\}$	0.002299	0.005025
128	$\{0, \ldots, 127\}$	0.004729	0.01333
256	$\{0, \ldots, 255\}$	0.008496	0.04676
512	$\{0, \ldots, 511\}$	0.02227	0.1815
1024	$\{0, \ldots, 1023\}$	0.03608	0.7419
2048	$\{0, \ldots, 2047\}$	0.07648	2.9806
4096	$\{0, \ldots, 4095\}$	0.1636	12.46
8192	$\{0, \ldots, 8191\}$	0.3323	50.77
16384	$\{0, \ldots, 16383\}$	0.6857	214.2
32768	$\{0, \ldots, 32767\}$	1.421	890.6

Table 2: **Table comparing the runtimes of untrimmed vs. trimmed p-convolution trees.** The same problem, $Y = X_1 + X_2 + \cdots + X_m$ with $p = 1$, was used for both trees, where $X_1, \ldots, X_m \in \{0, 1\}$ and $Y \in \{0, 1, \ldots, m\}$.

tree runtimes where calculated using `EvergreenForest` engine. The computer used had dual AMD EPYC 7351 16-core processors with 256gb of RAM.

3.1 Trimmed p-convolution Tree Benchmarks

Trimmed p-convolution trees are benchmarked versus untrimmed p-convolution trees (Table 2). The random variables are under the constraint $Y = X_1 + X_2 + \cdots + X_m$ where $X_1, X_2, \ldots, X_m$ are binary, random variables and $Y \in \{0, m\}$. The times reported are to get posteriors on all variables.

3.2 Runtime and Error Analysis of Trimmed p-convolution Trees and Specialized Methods

Table 3 shows the runtime and error of trimmed p-convolution trees versus the specialized methods mentioned in the paper. All inputs are binary (*i.e.* $n = 2$) as required by the specialized methods. The specialized methods were written in `C++` and should have very small run-time constants as the only data structures used are vectors and there are very few constant operations done per iteration. "HOP-MAP" is an implementation of the method presented by Tarlow et al. (2010) which calculates both the MAP and max-marginals. Trimmed p-convolution trees use numeric max-convolution for $p = \infty$, this explains the error for larger problems when comparing to HOP-MAP. Since noisy-or uses $p = 1$ numeric error is much less pronounced. Noisy-or was not measured against an untrimmed p-convolution tree because trimming is required to do underflow/overflow trimming which is what enables the imitation of the noisy-or operator. For both the noisy-or and HOP-MAP, n and d are fixed so the trimmed p-convolution solves either case with at worst runtime $\Theta(m \log(m) \log(m))$.

m	p	Method and Runtime(s)		L1 Error	L2 Error	Mean squared error
		Noisy-or	Trimmed p-convolution tree			
2	1.0	8.200×10^{-06}	0.0001639	0	0	0
4	1.0	1.700×10^{-05}	0.0002296	0	0	0
8	1.0	3.680×10^{-05}	0.0004025	0	0	0
16	1.0	5.460×10^{-05}	0.0005449	0	0	0
32	1.0	0.0001161	0.001105	0	0	0
64	1.0	0.0002733	0.00269	0	0	0
128	1.0	0.0005744	0.005009	0	0	0
256	1.0	0.0009627	0.009121	9.728×10^{-111}	9.728×10^{-111}	4.864×10^{-218}
512	1.0	0.001863	0.01453	3.899×10^{-221}	0	0
1024	1.0	0.003494	0.0286	0	0	0
2048	1.0	0.009141	0.05925	0.0003395	0.00024	0.0002361
4096	1.0	0.01293	0.1248	2.574×10^{-05}	1.820×10^{-05}	2.714×10^{-06}
8192	1.0	0.02491	0.2544	6.137×10^{-07}	4.340×10^{-07}	3.086×10^{-09}
16384	1.0	0.05118	0.5097	1.184×10^{-10}	8.374×10^{-11}	2.298×10^{-16}

m	p	Method and Runtime(s)			L1 Error	L2 Error	Mean squared error
		HOP-MAP	Trimmed p-convolution tree	Untrimmed p-convolution tree			
2	∞	6.780×10^{-05}	0.0001011	6.900×10^{-05}	0	0	0
4	∞	5.970×10^{-05}	0.0001555	0.000106	0	0	0
8	∞	8.960×10^{-05}	0.0002902	0.000192	0	0	0
16	∞	0.0001325	0.0006183	0.00371	0	0	0
32	∞	0.000251	0.001356	0.000789	0	0	0
64	∞	0.0002504	0.002414	0.001893	0	0	0
128	∞	0.0007291	0.005067	0.005850	0	0	0
256	∞	0.001393	0.008527	0.01574	0	0	0
512	∞	0.00287	0.01509	0.05524	0	0	0
1024	∞	0.007398	0.02818	0.1861	0	0	0
2048	∞	0.01251	0.08808	44.20	0.0003270	3.067×10^{-05}	5.779×10^{-06}
4096	∞	0.01915	0.2346	184.5	0.0002390	1.304×10^{-05}	2.089×10^{-06}
8192	∞	0.02982	0.5756	791.5	0.0001195	5.575×10^{-06}	7.639×10^{-07}
16384	∞	0.05271	1.372	3218.0	0.0001531	2.387×10^{-06}	2.801×10^{-07}

Table 3: **Tables of error and time analysis for Evergreen versus specialized methods.** All methods used $n = 2$ and varied with the number of the input variables, m. Times are averaged over ten iterations. p-trimmed convolution tree has more numerical instability when using $p = \infty$ due to the use of numeric max-convolution. In both cases all $m+1$ messages out for $Y, X_1, X_2, \ldots, X_m$ were calculated. There are no runtimes for noisy-or implemented in an untrimmed tree because trimming is necessary to imitate the `or` operator.

For the noisy-or case, since $Y \in \{0, 1\}$, the supports, and therefore the convolutions, at all nodes in the tree are constant and so the trimmed p-convolution runs in $\in \Theta(m)$ time.

3.3 Application of Lazily Trimmed p-convolution Trees in Protein Inference

The study of proteins is important in understanding genetics (Pandey and Mann, 2000), drug discovery (Jhanker et al., 2012), and many biological systems. Proteins are comprised of peptides which are chains of amino acids. Protein inference is the problem of deciding which proteins are present in a sample. A protein inference algorithm such as `Fido` (Serang

et al., 2010) or `EPIFANY` (Pfeuffer et al., 2020) may be used to calculate the probability of the proteins being present in the sample given information about which of their constituent peptides are in the sample. While `Fido` and `EPIFANY` both use graphical models, protein inference algorithms in general do not have to use graphical models and may incorporate other information not included here such as which animals the proteins come from.

When a sample is put into a mass spectrometer the proteins in the sample are broken apart into peptides and the spectra produced will give information about the presence of each peptide in the sample. Spectra are processed by performing a peptide search using Comet (Eng et al., 2013) and post-processing using Percolator (Käll et al., 2007). Percolator produces the probability of each peptide being in the sample.

LBP is used to estimate marginal (when $p = 1$) and max-marginal ($p = \infty$) posterior probabilities on the presence or absence of the proteins. In the protein inference models used here (`Fido` and models (A),(B), and (C) in Table 5), the common additive dependency is: $Pr(N_j) = Pr(X_1) + Pr(X_2) + \cdots + Pr(X_m), N_j \in \{0, 1, \ldots, m\}$. N_j is the the number of proteins which emit peptide j in the sample and $Pr(X_i)$ are Bernoulli indicator variables on the presence of protein i in the sample. The creation of peptides by proteins has causal independence (Heckerman and Breese, 1996) and so the additive dependency for peptide j includes the probability of protein i if and only if peptide j is a constituent peptide of protein i. Proteins have identical and independent prior probability γ. A peptide's prior is the probability produced by Percolator. The joint on the presence of a peptide and the number of proteins which emit peptide j is $Pr(Y_j = 1 \mid N_j = n) = 1 - ((1 - \beta) \cdot (1 - \alpha)^n)$. α (which is not the same as the α used for CPTs) is the probability that if the protein is present it will emit a peptide. β is the probability that a truly absent peptide is erroneously observed during the peptide search.

Here, we show the runtimes of `Fido` when implemented using lazily trimmed p-convolution trees and LBP (similar to EPIFANY) on three separate data sets. The 18mix data set is a mixture of eighteen proteins from several different species: bovine, E. coli, B. licheniformis, rabbit, horse, and chicken (Klimek et al., 2008). The IPRG data set contains 5,592 E. coli proteins (Lee et al., 2018). The yeast data set has 3,443 yeast proteins (Ramakrishnan et al., 2009). These three data sets were chosen because they are well curated and are considered to be ground truth data sets.

Table 4 shows runtimes for protein inference when the convolution tree is trimmed versus untrimmed. The additive dependencies in the models are explained above. Trimmed versus untrimmed FFT convolutions will result in different levels of numerical error. This, combined with LBP, caused some models to not fully converge when performing the untrimmed convolutions. For this reason, we set the convergence threshold to zero so that we could measure the time taken for one million iterations, instead of measuring how long until convergence.

Table 5 shows the difference in runtime when standard trimming is applied versus underflow/overflow trimming for a few basic models. In the standard `Fido` model, overflow/underflow trimming would have no noticeable effect due to the supports on the additive dependencies, so the models in Table 5 have to be modified in some way to make overflow/underflow trimming have an effect.

p	Iterations	IPRG Data Set Trimmed	IPRG Data Set Untrimmed	18mix Data Set Trimmed	18mix Data Set Untrimmed	Yeast Data Set Trimmed	Yeast Data Set Untrimmed
1.0	1m	13.90	24.44	2.94	12.48	40.82	43.35
∞	1m	12.23	23.31	2.931	12.71	20.08	29.78

`Fido` Parameters	IPRG Data Set	18mix Data Set	Yeast Data Set
α	0.09017	0.01818	0.02439
β	0.1459	0.2868	0.1622
γ	0.1854	0.005026	0.003104

Table 4: **Table showing runtime and parameters for protein inference with and without trimming.** Runtimes for protein inference on the three ground truth data sets. As used within `Fido`, α, β, and γ were found using a golden-section search (Kiefer, 1953).

Model	p	IPRG Data Set Overflow Trimmed	IPRG Data Set Additive Trimmed	18mix Data Set Overflow Trimmed	18mix Data Set Additive Trimmed	Yeast Data Set Overflow Trimmed	Yeast Data Set Additive Trimmed
(A)	1.0	1.22811	—	2.291	2.219	5.357	5.360
	∞	—	—	2.157	2.260	5.282	5.284
(B)	1.0	—	—	3.324	3.580	3.235	3.504
	∞	—	—	—	—	8.141	8.116
(C)	1.0	1.81286	1.82866	3.259	3.348	8.626	8.377
	∞	—	—	3.346	3.262	8.202	8.166

Table 5: **Table showing runtimes for additive trimming vs underflow/overflow trimming for several different protein inference models.** Models (A), (B), and (C) all start with the same graphical model as `Fido` and then they are modified in some way. Models (A) and (C) modify the additive dependency by making $N_j \in \{0, 1\}$ which creates overflow. Models (B) and (C) have an extra additive dependency: $Pr(X_i) = \sum_j E_{i,j}$. The binary $E_{i,j}$ variables exist if peptide j is a constituent of protein i and $Pr(E_{i,j}) = 1$ if $Pr(X_i) = 1$ $Pr(Y_j) = 1$, $Pr(E_{i,j}) = 0$ otherwise. If a model did not converge after one million iterations, its runtime is reported as "–". All `Fido` parameters are the same as in Table 4.

3.4 Comparing the Performance of CPTs Versus Max-convolution Trees and HOP-MAP

Table 6 and Table 7 show the runtime of approximating the max-marginals on some subset of $Y, X_1, \ldots, X_m$. Both methods described above (doing the full Cartesian product on $X_1 + X_2 + \cdots + X_m$ and holding out a single X_i from the Cartesian product) are shown below. Holding out X_i is typically an order of magnitude faster than performing the full Cartesian product while also having less error, the runtime is further reduced when only the marginal on the single X_i variable is calculated.

Table 7 uses the same implementation of HOP-MAP as Table 3 which calculates the max-marginals on $Y, X_1, \ldots, X_m$. Similar to Table 6, both CPT methods are used and the method which holds out X_i sees a similar reduction in both runtime and error. Since HOP-MAP calculates max-marginals on $X_1, X_2, \ldots, X_m$, there is no included time for CPT when holding out then only calculating the max-marginal on X_i.

4. Discussion

Trimmed p-convolution trees come within a log-factor of the best known specialized methods while having no restrictions on the data. If n is held constant and optimal trimming is allowed so that every node in the tree does the same amount of work, trimmed p-convolution trees run in $\Theta(m)$, the same cost as loading the data. In practice, trimmed p-convolution trees are $\approx 26\times$ slower than the HOP-MAP method for m=16,384 (Table 3). HOP-MAP is theoretically better by a log-factor, but the speed-up is also likely due to HOP-MAP using very simple data structures and standard `C++` library sorting methods. Trimmed p-convolution trees are $\approx 10\times$ slower than noisy-or, but they really shine compared to untrimmed p-convolution trees, being $\approx 626\times$ faster for 32768 binary random variables. Other than p-convolution trees, there are currently no methods which can solve all $d = 1$ problems in better-than-exponential time.

In protein inference, we see that lazily trimming the p-convolution trees can result is substantial speed increases (Table 4). While we ran `Fido` on known, published data sets it is likely that in practice the data will be significantly larger and the speed-up from trimming will become more pronounced. When compared across several different models, underflow/overflow trimming does not have a significant cost compared to standard trimming (Table 5. Underflow/overflow trimming tended have significantly higher posteriors on the proteins, including giving several proteins a 100% chance of being present, than standard trimming for $p = 1$. When $p = \infty$, the posteriors from both methods were essentially identical.

Basic implementations of the CPT to solve $Y = X_1 + X_2 + \cdots + X_m$ show promise of a fast approximation for max-convolution. We see that holding out a single X_i then taking the Cartesian product of $X_1 + \cdots + X_{i-1} + X_{i+1} + \cdots + X_m$ and the top values in X_i reduces the error on the max-marginal of X_i to around the same level as Evergreen with FFT convolution while having a runtime reduced by at least one order of magnitude. The accuracy on Y is not as great as on X_i when holding out X_i; however, one can solve for Y in the same manner by solving for $-X_i = -Y + X_1 + \cdots + X_{i-1} + X_{i+1} + \cdots + X_m$. If the max-marginal of more than one variable, or zero error, is desired then CPTs are still no

m, n	k	Method	Runtimes Time (s)	Variable	Error L1	L2	MSE
m=128 n=32	—	EVG Naïve	0.3103				
	—	EVG	0.2178	Y	2.188×10^{-07}	1.719×10^{-08}	1.27×10^{-12}
				X	4.647×10^{-06}	1.114×10^{-06}	3.924×10^{-11}
	4096	CPT	0.02138	Y	0.0004575	6.485×10^{-05}	1.675×10^{-05}
				X	0.05803	0.02889	0.0238
	4096	CPT holdout X_i	0.002333	Y	0.0003757	3.909×10^{-05}	6.073×10^{-06}
				$X_{j\neq i}$	0.05718	0.02746	0.02615
	4096	CPT X_i holdout X_i	0.0006089	X_i	5.045×10^{-07}	1.425×10^{-07}	1.598×10^{-12}
	8192	CPT	0.04089	Y	0.0004534	6.178×10^{-05}	1.52×10^{-05}
				X	0.05762	0.02791	0.02292
	8192	CPT holdout X_i	0.004173	Y	0.0003726	3.827×10^{-05}	5.824×10^{-06}
				$X_{j\neq i}$	0.05673	0.02647	0.02563
	8192	CPT X_i holdout X_i	0.0006119	X_i	5.045×10^{-07}	1.425×10^{-07}	1.598×10^{-12}
	16384	CPT	0.07863	Y	0.0004513	6.035×10^{-05}	1.45×10^{-05}
				X	0.05762	0.02791	0.02218
	16384	CPT holdout X_i	0.0078	Y	0.000369	3.738×10^{-05}	5.556×10^{-06}
				$X_{j\neq i}$	0.05673	0.02647	0.02515
	16384	CPT X_i holdout X_i	0.0007036	X_i	5.045×10^{-07}	1.425×10^{-07}	1.598×10^{-12}
m=256 n=32	—	EVG Naïve	1.214				
	—	EVG	0.5504	Y	1.38×10^{-07}	8.825×10^{-09}	6.285×10^{-13}
				X	3.423×10^{-06}	8.25×10^{-07}	2.513×10^{-11}
	4096	CPT	0.04264	Y	0.0002358	3.377×10^{-05}	9.084×10^{-06}
				X	0.0571	0.02659	0.02575
	4096	CPT holdout X_i	0.004865	Y	0.0002043	2.049×10^{-05}	3.338×10^{-06}
				$X_{j\neq i}$	0.05848	0.02905	0.02731
	4096	CPT X_i holdout X_i	0.0007699	X_i	3.241×10^{-07}	9.994×10^{-08}	5.602×10^{-13}
	8192	CPT	0.08222	Y	0.0002342	3.202×10^{-05}	8.167×10^{-06}
				X	0.05678	0.02615	0.02545
	8192	CPT holdout X_i	0.008955	Y	0.0002033	2.014×10^{-05}	3.224×10^{-06}
				$X_{j\neq i}$	0.05815	0.02809	0.02697
	8192	CPT X_i holdout X_i	0.0008241	X_i	3.241×10^{-07}	9.994×10^{-08}	5.602×10^{-13}
	16384	CPT	0.1577	Y	0.0002332	3.106×10^{-05}	7.683×10^{-06}
				X	0.05634	0.02569	0.02494
	16384	CPT holdout X_i	0.01693	Y	0.0002021	1.973×10^{-05}	3.094×10^{-06}
				$X_{j\neq i}$	0.05815	0.02809	0.02668
	16384	CPT X_i holdout X_i	0.0008906	X_i	3.224×10^{-07}	9.89×10^{-08}	5.566×10^{-13}

Table 6: **Table showing runtimes for Evergreen versus CPT.** This table shows the error and runtime of CPT versus Evergreen (EVG). Both are measured against a version of Evergreen (labeled as "EVG Naïve") which uses exact convolution (*i.e.* not FFT convolution). EVG Naïve is not used in practice, but is used here as a benchmark to evaluate the trade-off between accuracy (where EVG Naı"ve is perfect) and speed. EVG uses fast p-norm convolution and is used in practice. Runtimes labeled "CPT" includes calculating the Cartesian product for $X_1 + X_2 + \cdots + X_m$ and the max-marginals for all $Y, X_1, \ldots, X_m$. Entries which "holdout X_i" use the method which holds out X_i from the Cartesian product then performs the Cartesian product between the top values of X_i and $X_1 + \cdots + X_{i-1} + X_{i+1} + \cdots + X_m$. "CPT holdout X_i" runtime is the time to perform the Cartesian products and get max-marginals on all random variables. "CPT X_i holdout X_i" runtime is the time to perform the Cartesian products and only get the max-marginals on Y and X_i, this does not calculate the indices so it can be done $\in o(k \cdot m)$. The k value for the selection on X_i and the k value for the selection on $X_1 + \cdots + X_{i-1} + X_{i+1} + \cdots + X_m$ are selected so that their product is the k value used in the Cartesian product on $X_1 + X_2 + \cdots + X_m$. When calculating the error for the CPT method which holds out X_i we look at the error for both X_i and $X_{j\neq i}$ separately, since it is possible that only a single $X_{j\neq i}$ is used in the large Cartesian product, whereas we are guaranteed to see several different values of X_i. All runtimes and errors were averaged over ten trials.

		Runtimes			Error		
m	k	Method	Time (s)	Variable	L1	L2	MSE
m=16	—	HOP-MAP	9.75e-05				
	16	CPT	5.35e-05	Y	0.02499	0.0108	0.002159
				X	0.06576	0.0465	0.01705
	16	CPT holdout X_i	4.37e-05	Y	0.0274	0.01182	0.002806
				$X_{j\neq i}$	0.0003973	0.000281	2.765×10^{-06}
	16	CPT X_i holdout X_i	6.117e-05	X_i	0.0003973	0.000281	2.765×10^{-06}
	64	CPT	0.0001238	Y	0.01191	0.004832	0.0005036
				X	0.02709	0.01916	0.004311
	64	CPT holdout X_i	8.8e-05	Y	0.01416	0.005671	0.0006905
				$X_{j\neq i}$	0.00028	0.000198	2.352×10^{-06}
	64	CPT X_i holdout X_i	8.267e-05	X_i	0.00028	0.000198	2.352×10^{-06}
m=32	—	HOP-MAP	0.0001931				
	32	CPT	0.0001256	Y	0.01845	0.006892	0.001642
				X	0.07433	0.05256	0.02642
	32	CPT holdout X_i	8.6e-05	Y	0.01997	0.007664	0.002057
				$X_{j\neq i}$	0.0004595	0.0003249	5.678×10^{-06}
	32	CPT X_i holdout X_i	0.0001012	X_i	0.0004595	0.0003249	5.678×10^{-06}
	160	CPT	0.0004626	Y	0.0116	0.004167	0.0006211
				X	0.04185	0.02959	0.01142
	160	CPT holdout X_i	0.0002773	Y	0.012	0.00428	0.0006515
				$X_{j\neq i}$	2.52×10^{-05}	1.782×10^{-05}	1.905×10^{-08}
	160	CPT X_i holdout X_i	0.0001158	X_i	2.52×10^{-05}	1.782×10^{-05}	1.905×10^{-08}
m=64	—	HOP-MAP	0.000321				
	64	CPT	0.000407	Y	0.01247	0.004052	0.001097
				X	0.1908	0.1349	0.03865
	64	CPT holdout X_i	0.0002862	Y	0.01266	0.004168	0.001181
				$X_{j\neq i}$	0.0002379	0.0001682	7.224×10^{-07}
	64	CPT X_i holdout X_i	0.0001699	X_i	0.0002379	0.0001682	7.224×10^{-07}
	384	CPT	0.001268	Y	0.009196	0.002875	0.0005692
				X	0.1309	0.09255	0.02005
	384	CPT holdout X_i	0.000747	Y	0.009022	0.002821	0.0005386
				$X_{j\neq i}$	0.0001566	0.0001107	6.604×10^{-07}
	384	CPT X_i holdout X_i	0.0002647	X_i	0.0001566	0.0001107	6.604×10^{-07}
m=128	—	HOP-MAP	0.0006896				
	128	CPT	0.001012	Y	0.008139	0.002329	0.000717
				X	0.1809	0.1279	0.04728
	128	CPT holdout X_i	0.000639	Y	0.008078	0.002345	0.0007213
				$X_{j\neq i}$	6.757×10^{-05}	4.778×10^{-05}	3.575×10^{-08}
	128	CPT X_i holdout X_i	0.0002919	X_i	6.757×10^{-05}	4.778×10^{-05}	3.575×10^{-08}
	896	CPT	0.004787	Y	0.00655	0.001788	0.0004215
				X	0.09668	0.06837	0.03074
	896	CPT holdout X_i	0.003069	Y	0.006537	0.001788	0.0004222
				$X_{j\neq i}$	3.273×10^{-05}	2.315×10^{-05}	1.607×10^{-08}
	896	CPT X_i holdout X_i	0.0004419	X_i	3.273×10^{-05}	2.315×10^{-05}	1.607×10^{-08}
m=256	—	HOP-MAP	0.001097				
	256	CPT	0.00338	Y	0.004864	0.001243	0.0004021
				X	0.1993	0.1409	0.0594
	256	CPT holdout X_i	0.001761	Y	0.004752	0.001233	0.0003961
				$X_{j\neq i}$	5.467×10^{-06}	3.866×10^{-06}	4.191×10^{-10}
	256	CPT X_i holdout X_i	0.0005929	X_i	5.467×10^{-06}	3.866×10^{-06}	4.191×10^{-10}
	2048	CPT	0.01884	Y	0.004131	0.0009855	0.0002539
				X	0.1211	0.08561	0.04654
	2048	CPT holdout X_i	0.009243	Y	0.004128	0.0009991	0.0002613
				$X_{j\neq i}$	5.167×10^{-06}	3.653×10^{-06}	4.164×10^{-10}
	2048	CPT X_i holdout X_i	0.0008625	X_i	5.167×10^{-06}	3.653×10^{-06}	4.164×10^{-10}

Table 7: **Table showing runtimes for HOP-MAP versus CPT.** Since HOP-MAP only works on binary random variables, $n = 2$. All method are the same as those in Table 6. Note that HOP-MAP calculates the max-marginals on all $Y, X_1, \ldots, X_m$ where the method "CPT X_i holdout X_i" only calculates the max-marginal on X_i. All other CPT methods calculate the same max-marginals as HOP-MAP. All runtimes and errors were averaged over ten trials.

match for HOP-MAP for larger problems, likely due to n being restricted to 2 which may cause less saturation in the supports on Y.

Ideally, each X_i could be held out one at a time in order to get the best approximate max-marginals on $Y, X_1, X_2, \ldots, X_m$; however, holding out every variable and updating all max-marginals after each selection would be quadratic in m. It may be possible for each interior node to keep track of its top k values with all X_i and without each individual X_i that is in its Cartesian product. This method will be more costly than a standard CPT (both in memory and runtime), but the trade-off for accuracy may be worth the cost. Another method to calculate more outcomes in Y is to filter redundant values in the pairwise selections in the interior nodes. Each interior node can have a set which keeps track of the supports in the values it has generated so far. If, in further selections, the same support is seen again it can be filtered from being put into the results. In this way, each interior node can only touch unique outcomes in their k-selection.

In order for trimmed p-convolution trees to match HOP-MAP in theoretical runtime, the forward pass would have to be done $\in O(m \log(m))$ time. This would require a fast algorithm to solve multi-convolution which would allow the p-convolution tree to go straight from the prior on the leaves to calculate the prior on the root. Perhaps this could be done by finding the roots of the polynomials on the leaves and some fast algorithm for multiplying the roots together. However, the backward pass would still be challenging as it requires the priors to be cached on all interior nodes in the tree, but this merits further investigation.

LBP and trimmed p-convolution trees may be used in concert to form "p-convolution forests." These p-convolution forests can be used to efficiently solve probabilistic linear Diophantine equations and other problems which may be represented as graphical models. In just computational biology this includes applications in polyploid genotyping and mapping (Serang et al., 2012), protein identification (Serang, 2014), and metagenomics (or metatranscriptomics or metaproteomics).

Large problems with $p \gg 1$ can suffer numerical error from using fast numerical max-convolution; however, error on large problems may be worth it in order to use both sum-product and max-product inference on the same graphical model. There is also no other method which may use inference somewhere between sum-product and max-product (*e.g.* $1 < p < \infty$).

There are times where only allowing sum-product inference may lead to incorrect results. Let $x' = max(\sum_y f(x, y))$ and $x^* = \text{argmax}_x(f(x, y))$ be the maximum value of the marginal and max-marginal, respectively, of a two-dimensional distribution (in which one or more axis may be flattened from other multivariate distributions). While it is true that $f(x^*, y) > f(x', y)$, it may not be true that $x^* > x'$. Here, the marginalization using sum-product inference has led to x' becoming the mode of the marginal distribution, when in reality x^* should be the mode. If the sum-product inference is performed in a loopy manner where the result is multiplied back in and then marginalized out repeatedly, this may lead to the true mode, x^*, having probability of zero. Losing the mode of the distribution may lead to inaccurate results.

Further speed-ups may be obtained from making the FFTs even more sparse. Instead of trimming on just the ends of the arrays, it is possible to trim in the middle as well to create several non-zero partitions in the arrays (Stockham Jr., 1966). Then, all pairs of non-zero partitions across the two arrays would be convolved. This will lead to smaller convolutions;

however, the number of FFTs required is quadratic in the number of non-zero partitions so removing all zero entries may lead to poor performance.

5. Code Availability

The `C++11` code for the `EvergreenForest` engine, its modules, demos, and utilities for visualizing graphs in Python are freely available under an MIT software license and can be downloaded at `https://bitbucket.org/orserang/evergreenforest`. The entire library is implemented in a header-only fashion, so the essential components of each module can be included via a single `#include` statement. There is also a stand-alone modeling language for `EvergreenForest` allowing users to take advantage of the engine without using `C++`.

6. Funding

This material is based on work supported by the National Science Foundation under grant no. 1845465.

Acknowledgments

We are grateful to Karol Węgrzycki for his great discussion on the topic of min and max-convolution. We are thankful to Kyle Lucke for his discussion and help with running `EvergreenForest` on protein inference models. Thanks to Max Thibeau for his work on the `EvergreenForest` modeling language.

References

R.E. Barlow and K.D. Heidtmann. Computing k-out-of-n system reliability. *IEEE Transactions on Reliability*, 33(4):322–323, 1984.

L.A. Belfore. An $O(n(log_2(n))^2)$ algorithm for computing the reliability of k-out-of-n: G and k-to-l-out-of-n: G systems. *IEEE Transactions on Reliability*, 44(1):132–136, 1995.

C Berrou, A. Glavieux, and P. Thitimajshima. Near shannon limit error-correcting coding and decoding: Turbo-codes. 1. *Proceedings of ICC '93 - IEEE International Conference on Communications*, 2:1064–1070 vol.2, 1993.

B. Chazelle. The soft heap: an approximate priority queue with optimal error rate. *Journal of the ACM (JACM)*, 47(6):1012–1027, 2000.

J. W. Cooley and J. W. Tukey. An algorithm for the machine calculation of complex Fourier series. *Mathematics of computation*, 19(90):297–301, 1965.

J. K. Eng, T. A. Jahan, and M. R. Hoopman. Comet: An open-source ms/ms sequence database search tool. *PROTEOMICS*, 13:22–24, 2013.

G. N. Frederickson. An optimal algorithm for selection in a min-heap. *Information and Computation*, 104(2):197–214, 1993.

D. Heckerman and J.S. Breese. Causal independence for probability assessment and inference using Bayesian networks. *Systems, Man and Cybernetics, Part A: Systems and Humans, IEEE Transactions on*, 26(6):826–831, 1996.

F. Heyl and O. Serang. TRIOT: Faster tensor manipulation in `C++11`. *The Art, Science, and Engineering of Programming*, 1, 2017.

Y. M. Jhanker, M. F. Kadir, R. I. Khan, and R. Hasan. Proteomics in drug discovery. *Journal of Applied Pharmaceutical Science*, 2(08):01–12, 2012.

L. Käll, J. Canterbury, J. Weston, W. S. Noble, and M. J. MacCoss. A semi-supervised machine learning technique for peptide identification from shotgun proteomics datasets. *Nature Methods*, 4:923–25, 2007.

H. Kaplan, L. Kozma, O. Zamir, and U. Zwick. Selection from heaps, row-sorted matrices and $X + Y$ using soft heaps. *Symposium on Simplicity in Algorithms*, pages 5:1–5:21, 2019.

J. Kiefer. Sequential Minimax Search for a Maximum. *Proceedings of the American Mathematical Society*, 4(3):502–506, 1953.

J. Klimek, J. S. Eddes, L. Hohmann, J. Jackson, A. Peterson, S. Letarte, P. R. Gafken, J. E. Katz, P. Mallick, H. Lee, A. Schmidt, R. Ossola, J. K. Eng, R. Aebersold, and D. B. Martin. The standard protein mix database: a diverse data set to assist in the production of improved peptide and protein identification software tools. *Journal of Proteome Research*, 7(1):96–1003, 2008.

P. Kreitzberg, K. Lucke, and O. Serang. Selection on $X_1 + X_2 + \cdots + X_m$ with layer-ordered heaps. *arXiv preprint arXiv:1910.11993*, 2020a.

P. Kreitzberg, J. Pennington, K. Lucke, and O. Serang. Fast exact computation of the k most abundant isotope peaks with layer-ordered heaps. *Analytical Chemistry*, 92(15): 10613–10619, 2020b. doi: 10.1021/acs.analchem.0c01670.

P. Kreitzberg, K. Lucke, J. Pennington, and O. Serang. Selection on $X_1 + X_2 + \cdots + X_m$ via Cartesian product trees. *PeerJ Computer Science*, 7:e483, 2021.

J. Lee, H. Choi, C. M. Colangelo, D. Davis, M.R. Hoopmann, L. Käll, H. Lam, S.H. Payne, Y. Perez-Riverol, M. The, et al. Abrf proteome informatics research group (iprg) 2016 study: Inferring proteoforms from bottom-up proteomics data. *Journal of biomolecular techniques: JBT*, 29(2):39, 2018.

R.J. McEliece, D.J.C. MacKay, and J. Cheng. Turbo decoding as an instance of Pearl's "belief propagation" algorithm. *Selected Areas in Communications, IEEE Journal on*, 16 (2):140–152, 1998.

Q. Morris. Personal communications, 2011.

K.P. Murphy, Y. Weiss, and M.I. Jordan. Loopy belief propagation for approximate inference: An empirical study. In *Proceedings of the Fifteenth conference on Uncertainty in artificial intelligence*, pages 467–475, 1999.

A. Pandey and M. Mann. Proteomics to study genes and genomes. *Nature*, 405:837–846, 2000.

G. Papachristoudis and J. Fisher. Adaptive belief propagation. In *International Conference on Machine Learning*, pages 899–907, 2015.

J. Pearl. Reverend Bayes on inference engines: A distributed hierarchical approach. In *Proceedings of the Second National Conference on Artificial Intelligence*, pages 133–136, 1982.

J. Pennington, P. Kreitzberg, K. Lucke, and O. Serang. Optimal construction of a layer-ordered heap. *arXiv preprint arXiv:2007.13356*, 2020.

J. Pfeuffer and O. Serang. A bounded p-norm approximation of max-convolution for sub-quadratic Bayesian inference on additive factors. *Journal of Machine Learning Research*, 17(36):1–39, 2016.

J. Pfeuffer, T. Sachsenberg, T.M.H. jeerd T. Dijkstra, O. Serang, K. Reinert, and O. Kohlbacher. Epifany: A method for efficient high-confidence protein inference. *Journal of proteome research*, 19(3):1060–1072, 2020.

S.R. Ramakrishnan, C. Vogel, J.T. Prince, R. Wang, Z. Li, L.O. Penalva, M. Myers, E.M. Marcotte, and D.P. Miranker. Integrating shotgun proteomics and mrna expression data to improve protein identification. *Bioinformatics*, 25(11):1397–1403, 2009.

O. Serang. The probabilistic convolution tree: Efficient exact Bayesian inference for faster LC-MS/MS protein inference. *PLoS ONE*, 9(3):e91507, 2014.

O Serang. Optimally selecting the top k values from X + Y with layer-ordered heaps. *PeerJ Computer Science*, 2021.

O. Serang, M. J. MacCoss, and W. S. Noble. Efficient marginalization to compute protein posterior probabilities from shotgun mass spectrometry data. *Journal of Proteome Research*, 9(10):5346–5357, 2010.

O. Serang, M. Mollinari, and A.A.F. Garcia. Efficient exact maximum a posteriori computation for Bayesian SNP genotyping in polyploids. *PLoS ONE*, 7(2):e30906, 2012.

T.G. Stockham Jr. High-speed convolution and correlation. In *Proceedings of the April 26-28, 1966, Spring joint computer conference*, pages 229–233, 1966.

D. Tarlow, I. E. Givoni, and R. S. Zemel. HOP-MAP: Efficient message passing with high order potentials. In *International Conference on Artificial Intelligence and Statistics*, pages 812–819, 2010.

D. Tarlow, K. Swersky, R. S. Zemel, R. P. Adams, and B. J. Frey. Fast exact inference for recursive cardinality models. *arXiv preprint arXiv:1210.4899*, 2012.