

The Probabilistic Convolution Tree: Efficient Exact Bayesian Inference for Faster LC-MS/MS Protein Inference

Oliver Serang*

Thermo Fisher Scientific, Bremen, Germany

Abstract

Exact Bayesian inference can sometimes be performed efficiently for special cases where a function has commutative and associative symmetry of its inputs (called “causal independence”). For this reason, it is desirable to exploit such symmetry on big data sets. Here we present a method to exploit a general form of this symmetry on probabilistic adder nodes by transforming those probabilistic adder nodes into a probabilistic convolution tree with which dynamic programming computes exact probabilities. A substantial speedup is demonstrated using an illustration example that can arise when identifying splice forms with bottom-up mass spectrometry-based proteomics. On this example, even state-of-the-art exact inference algorithms require a runtime more than exponential in the number of splice forms considered. By using the probabilistic convolution tree, we reduce the runtime to $O(k \log(k)^2)$ and the space to $O(k \log(k))$ where k is the number of variables joined by an additive or cardinal operator. This approach, which can also be used with junction tree inference, is applicable to graphs with arbitrary dependency on counting variables or cardinalities and can be used on diverse problems and fields like forward error correcting codes, elemental decomposition, and spectral demixing. The approach also trivially generalizes to multiple dimensions.

Citation: Serang O (2014) The Probabilistic Convolution Tree: Efficient Exact Bayesian Inference for Faster LC-MS/MS Protein Inference. PLoS ONE 9(3): e91507. doi:10.1371/journal.pone.0091507

Editor: Haixu Tang, Indiana University, United States of America

Received: November 29, 2013; **Accepted:** February 12, 2014; **Published:** March 13, 2014

Copyright: © 2014 Oliver Serang. This is an open-access article distributed under the terms of the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original author and source are credited.

Funding: This work was created as an employee of Thermo Fisher Scientific (<http://www.thermofisher.com/>) with no external funding. The funder provided support in the form of salaries for author OS, but did not have any additional role in the study design, data collection and analysis, decision to publish, or preparation of the manuscript. The specific roles of this author are articulated in the author contributions section.

Competing Interests: OS is an employee of Thermo Fisher Scientific whose company funded this study. Unrelated to this study, similar research by Thermo Fisher Scientific includes a pending patent on a specific device implementing this procedure (i.e. it is not a software patent). There are no further patents, products in development or marketed products to declare. This does not alter the author's adherence to all the PLOS ONE policies on sharing data and materials.

* E-mail: oserang@uw.edu

Introduction

In bottom-up mass spectrometry pieces of digested proteins, which are called peptides, are first matched to observed spectral evidence, and the quality of the match between the peptide and the spectrum is scored [1–5]. These scored peptides are then used to perform inference on the proteins, whose present-absent states are usually the variables of interest. The computational cost of inference is non-trivial: some graphs can be processed efficiently, while performing inference on other graphs can be proven to solve the NP-hard minimum set cover problem [2]. Performing efficient but accurate inference on these graphs is important for producing reliable protein inferences.

Figure 1 depicts a simple graphical view of protein identification from tandem mass spectrometry experiment. In figure 1a, the causal flow of information is described graphically: proteins are digested and then fragmented to produce observed spectral evidence. Directed edges between proteins X_1, X_2, \dots and spectral data D_A, D_B, \dots represent causal statistical dependencies: proteins are connected to the MS/MS spectra matching peptides that can be produced according to the model of the digest (e.g. peptide strings resulting from an *in silico* digest of protein sequences using trypsin cleavage rules with up to one missed cleavage). Note that because peptides are usually paired with spectrum in a one-to-one

manner (pairing each peptide to its best-matching spectrum and pairing each spectrum with its best matching peptide to form a “PSM” or peptide-spectrum match), we simply draw the proteins producing that spectral evidence, thereby producing a bipartite graph of proteins (X) to spectral data (D). Note that this one-to-one mapping between peptides and spectra means that this bipartite graph can be viewed as a protein-peptide bipartite graph; however, rather than constrain ourselves to inference with MS/MS intensities, the data used for inference could just as easily constitute precursor MS intensities or even spectra resulting from a top-down experiment.

Proteins with shared spectral data (such as X_1 , X_2 , and X_3 , which share evidence D_A , which could describe an identified “degenerate” peptide) introduce new non-causal dependencies: if the score of the PSM corresponding to spectrum D_A is very high, it is tempting to award a high probability to protein X_1 ; however, proteins X_2 and X_3 compete for this shared evidence, and thereby have a chance to reduce the probability of X_1 . This process (called “explaining away” to describe the fact that the contribution of evidence to a single hypothesis is reduced by competing hypotheses) introduces new dependencies between all pairs of proteins sharing that evidence.

In figure 1b, we illustrate those shared protein-to-protein dependencies with the moral graph [6] (so called because “parent”

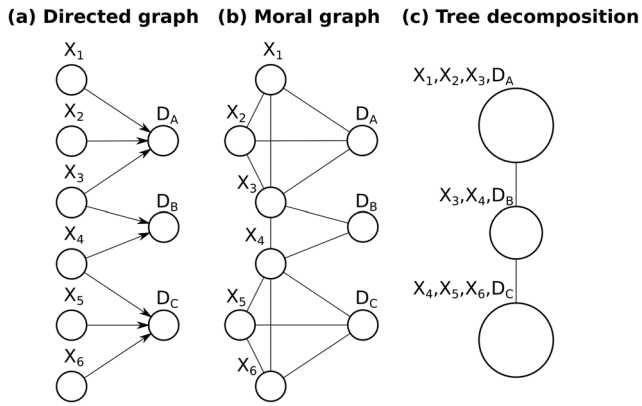


Figure 1. Mass spectrometry: a graphical view. (a) Directed edges between proteins X_1, X_2, \dots, X_6 and spectral data D_A, D_B, D_C represent causal statistical dependencies with spectra that can result from peptides in the adjacent protein. For simplicity, peptide-spectrum-matches (PSMs) are denoted simply using their spectral evidence, thereby producing a bipartite graph of proteins (X) to spectral data (D). Proteins X_1, X_2 , and X_3 share spectral data D_A : if the score of the PSM corresponding to spectrum D_A is very high, it is tempting to award a high probability to protein X_1 ; however, proteins X_2 and X_3 compete for this shared evidence, and thereby have a chance to reduce the probability of X_1 . This process (called “explaining away” to describe the fact that the contribution of evidence to a single hypothesis is reduced by competing hypotheses) introduces new dependencies between all pairs of proteins sharing that evidence. (b) These shared spectral data introduce new non-causal dependencies between proteins with shared successors in (a). These dependencies are visualized in the undirected moral graph. When multiple proteins share spectral evidence, these undirected edges connect all pairs of predecessors, creating a clique in the moral graph $K(X_1, X_2, X_3, D_A)$. (c) The tree decomposition (sometimes called the “junction tree” or “clique tree”) merges the moral graph from (b) without loss of dependencies, so that inference can be performed using Pearl’s belief propagation algorithm. Belief propagation starts at the top clique, which only shares variable X_4 with its neighbor. Therefore, the top clique can perform inference while leaving X_3 as a symbolic, unknown quantity, so that it can be used to send information from the cliques below (X_3 is an information bottleneck, through which the cliques below can influence the top clique). Likewise, the variables X_1 and X_2 can be marginalized out before sending any relevant information to considering the middle clique. This procedure can significantly reduce the runtime by allowing inference to be performed on the cliques rather than on all nodes in the tree; however, each clique represents an inseparable multidimensional distribution over several variables, and thus the cost of processing a single clique is more than exponential in the number of variables. When many proteins share common evidence (*i.e.* share at least one peptide identified by spectral evidence), a large clique is formed in the moral graph and inference becomes intractable in the general case. doi:10.1371/journal.pone.0091507.g001

nodes, which share a common child are “married by an edge”). The moral graph displays these latent dependencies between proteins, as well as the original causal dependencies from (a). Hence, when multiple proteins share spectral evidence, these undirected edges create a clique in the moral graph $K(X_1, X_2, X_3, D_A)$. These cliques can be joined in a tree, which is known as the tree decomposition [7] (also known as the “junction tree” or “clique tree”).

In figure 1c, we show a tree decomposition formed from the moral graph in figure 1b. This tree decomposition is performed by merges nodes in the moral graph so that the resulting graph can be viewed as a tree without losing any edges (lost edges would correspond to ignored dependencies). Posterior probabilities on graphs without cycles (*i.e.* trees) can be computed by visiting each

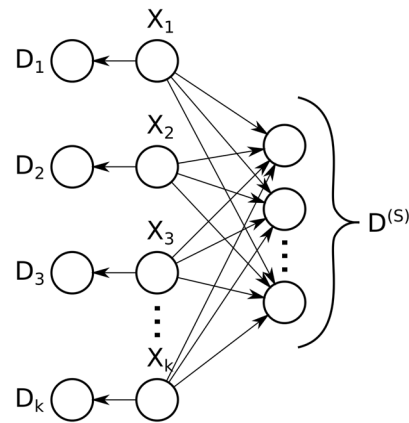


Figure 2. Difficult inference: mass spectrometry-based identification of splice variants. Several proteins X_1, X_2, \dots, X_k matching unique and shared peptide-level evidence. The peptide-level evidence, D is partitioned into unique peptide-level evidence (D_1, D_2, \dots, D_k) as well as a collection of shared peptide level evidence shared by all proteins ($D^{(S)}$). Graphs of this form are typical when searching mass spectra against protein databases containing substantial redundancy (*e.g.* databases with many splice variants or close homologs), because these types of proteins share core similarities but also have unique regions that distinguish them from one another. Inference on this type of graph cannot be performed efficiently through protein clustering, protein pruning, or junction tree decomposition; to date, exact Bayesian protein inference on such splice variant graphs has only been performed in super-exponential time. doi:10.1371/journal.pone.0091507.g002

clique only twice using Pearl’s belief propagation algorithm; however, each clique represents an inseparable multidimensional distribution over several variables, and thus the cost of processing a single clique is more than exponential in the number of variables (its state space is the Cartesian product of the contained variables’ outcomes). Therefore, the runtime of junction tree inference is more than exponential in the size of the largest clique (the “tree width” of the graph is the size of the largest clique minus one).

For this reason, when many proteins share common evidence (*i.e.* in tandem mass spectrometry, at least one shared peptide matching spectra), a joint dependency between all of those proteins is created, and the large resulting clique formed in the moral graph can make inference intractable in the general case. Without modifying it after tree decomposition, exact inference on the graph is more than exponential (that is, it is not in $O(2^n)$) in the number of proteins joined by such evidence [8]. And because these large cliques represent a full joint distribution of dimension n , even sampling procedures like Monte Carlo and Markov chain Monte Carlo (MCMC), which have been successfully applied to protein inference [8–10], cannot saturate the space with samples, and are thus insufficient.

In particular, Bayesian networks similar to the one shown in figure 2 can occur in mass spectrometry-based proteomics when attempting to identify homologous (or, more generally, proteins with sequence similarity). Specifically, these challenges occur in practice when attempting to distinguish between antibodies and other closely related splice variants [11], searching large databases containing non-canonical variants to find aberrant gene products (*e.g.* from samples of cancerous tissue), and in proteomic studies of organisms for which little or low-quality genomic information is available [12]. The entirety of spectral data for this graph, D , is partitioned into two categories: First, all proteins X_1, X_2, \dots, X_k in the weakly connected subgraph are adjacent to a collection of shared evidence (*e.g.* degenerate peptides matching observed

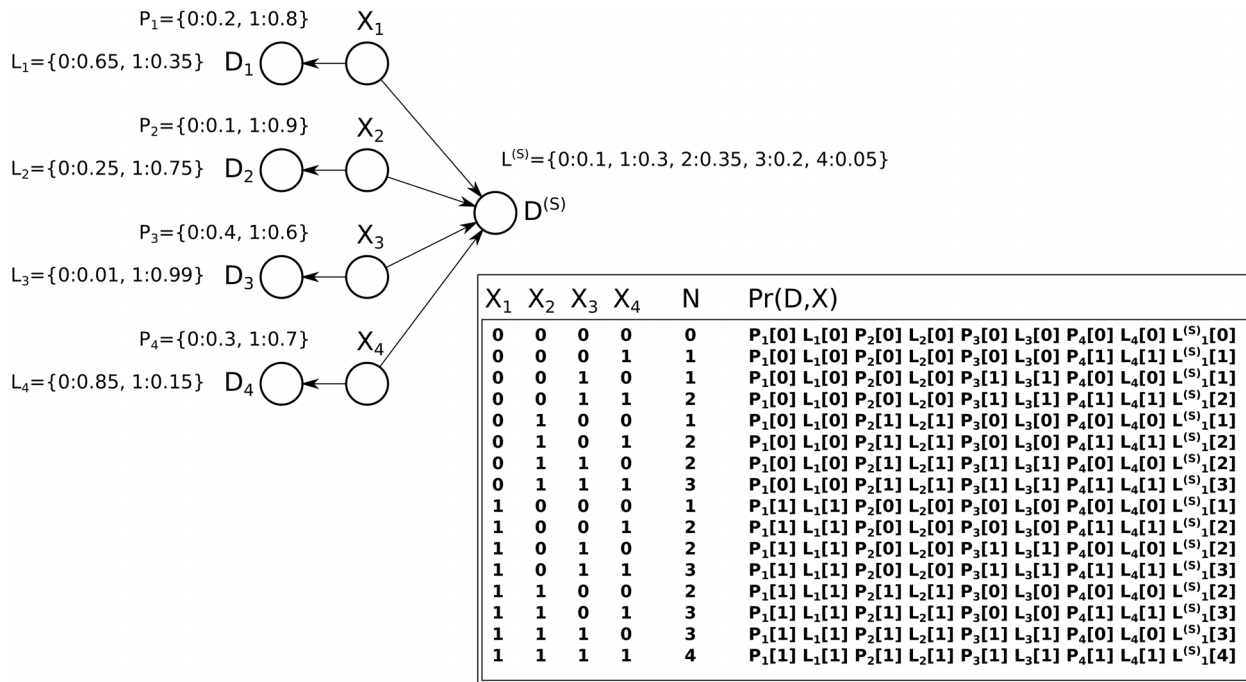


Figure 3. Illustration of the enumeration approach. Super-exponential enumeration is illustrated using a simple digraph. The protein prior for protein X_i is denoted using the vector P_i (written using Python dictionary notation), where $P_i[0] = \Pr(X_i=0)$ and $P_i[1] = \Pr(X_i=1)$. Likelihoods due to unique evidence for protein X_i are denoted L_i , and the likelihood due to shared evidence is shown using $L^{(S)}$, both using the same notation. The scores populating the L_i and $L^{(S)}$ vectors comes from the peptide-level likelihoods indicating the quality of the match between the peptide and any matching spectra (*i.e.* these scores come from the conditionally independent product of PSM scores for that peptide). For example, the prior probability on protein X_1 is 0.8, and a unique peptide corresponding to protein X_1 has the score 0.35 (indicating the relative likelihoods are 0.35 versus 0.65 for the respective hypotheses that the peptide matching spectrum D_1 is created by protein X_1 versus the hypothesis that the peptide is not created by protein X_1). The inset shows the table produced by enumerating all distinct protein configurations, and the resulting joint probability with all data (both unique and shared). This computational cost of this enumeration is in $O(k2^k)$ for k proteins. doi:10.1371/journal.pone.0091507.g003

spectra). These shared data, denoted $D^{(S)}$, produce a large clique in the moralized graph, which indicates that computing and passing the messages will be computationally infeasible on large problems without considering the form of the conditional probability function for the shared data. Second, each protein also has any number of unique evidence, which are found only in a single protein. The evidence (*e.g.* peptides) unique to protein X_i are denoted D_i .

A Previous Model for Bayesian Protein Inference

Previous work on exact Bayesian inference for protein identification [2,13] models every protein with an independent identically distributed prior probability γ that the protein is truly in the sample, a conditional probability α that a present protein would generate a constituent peptide, and a noise model where incorrect peptide identifications were modeled with probability β . Lastly, the event that one present protein is successfully digested into one of its shared constituent peptides does not influence the event that another protein is successfully digested into the same peptide.

Optimizations for more Efficient Protein Inference

Using this model, exact inference can sometimes be performed more efficiently using two optimizations: clustering and pruning. These two optimizations are used in conjunction with factorization, which separates and performs inference individually on weakly connected subgraphs, to perform exact inference more quickly.

The first optimization, clustering, merges together any collection of k protein nodes $X_{i_1}, X_{i_2}, \dots, X_{i_k}$ that contain identical peptide sets. Because the probability that a peptide is absent is the probability that it came from neither the noise model nor from any proteins:

$$(1-\beta) \prod_{j: X_{ij}=1} (1-\alpha) = (1-\beta)(1-\alpha)^{|\{j: X_{ij}=1\}|}$$

$$= (1-\beta)(1-\alpha)^{\sum_j X_{ij}}$$

These proteins were clustered because they share *identical* peptide sets, such that all spectral evidence depending on these proteins actually depends on the number of protein present $N = |\{j : X_{ij} = 1\}|$ rather than on the actual set of present proteins. This is because addition is a commutative and associative operation, and so rather than enumerate the power-set of proteins, marginalization can instead be performed on the number of proteins present:

$$\Pr(D, X_1 = x_1)$$

$$= \sum_{x_2} \sum_{x_3} \dots \sum_{x_k} \Pr(D, X_1 = x_1, X_2 = x_2, \dots, X_k = x_k)$$

$$= \sum_n \Pr(D|N=n) \Pr(N=n) \Pr(X_1 = x_1|N=n),$$

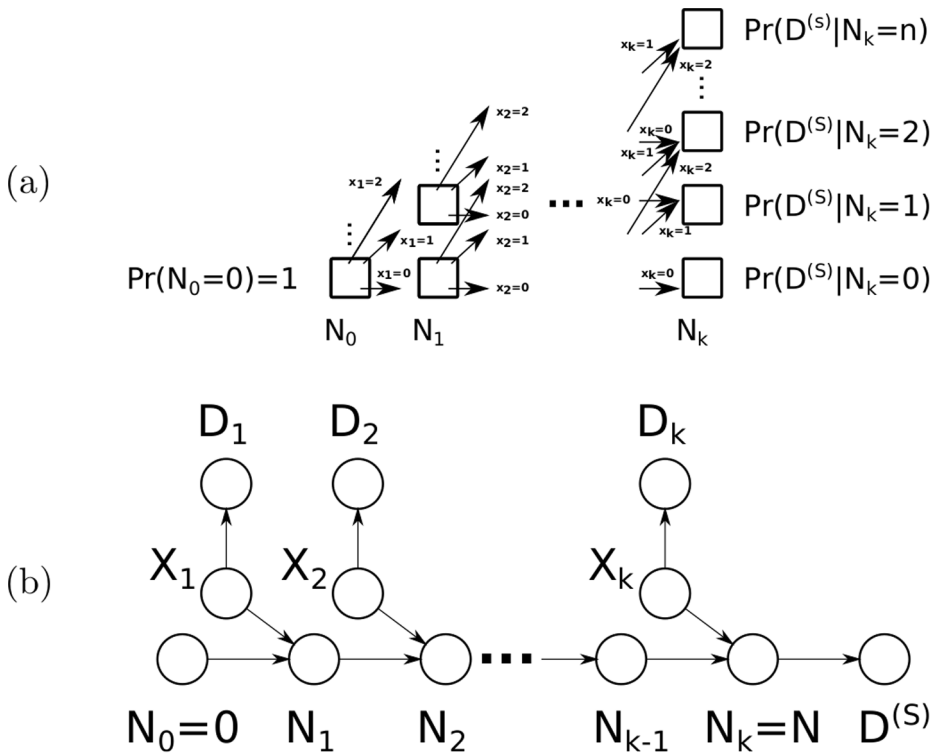


Figure 4. A quadratic dynamic programming approach and its generalization. (a) A dynamic programming approach to solving the problem from figure 2. In this approach all values of N , the information on which $D^{(S)}$ depends, are computed after successively including every next variable X_i . This allows paths in the exponential tree generated by the power-set to be merged when they result in the same value $N = n$, and thus allows a forward-backward algorithm to compute inference in quadratic time and space. (b) A general path graph can be constructed whenever the operation performed by the node N , on which the shared data $D^{(S)}$ depends, can be decomposed as a series of consecutive operations that aggregate X_i one at a time. This corresponds to operators with commutative and associative properties. The resulting transformation resembles Heckerman’s temporal transformation, which also uses quadratic time and space.
doi:10.1371/journal.pone.0091507.g004

where $\Pr(N = n) = \binom{k}{n} \gamma^n (1 - \gamma)^{k-n}$ and the conditional probability for an arbitrary protein i in the cluster $\Pr(X_i = x_i | N = n) = \frac{n}{k}$. Thus enumeration, which has a computational cost that exceeds 2^k , is reduced to $k + 1$ steps. However,

clustering cannot be performed unless the proteins clustered are adjacent to *identical* sets; the graph in figure 2 does not have this property, because proteins are potentially distinguishable by their unique evidence D_1, D_2, \dots, D_k .
The second optimization, pruning, exploits the fact that this type of protein-peptide emission model (sometimes referred to as a

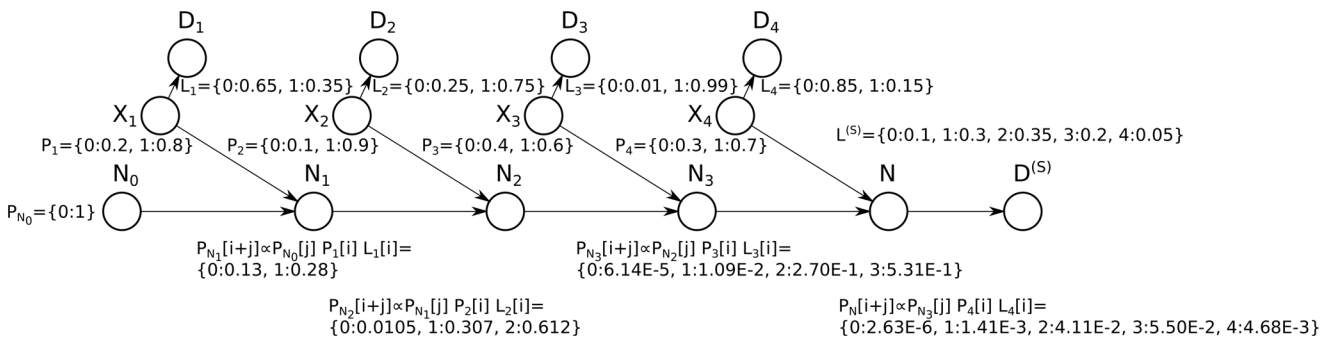


Figure 5. Illustration of the quadratic dynamic programming approach. The quadratic dynamic programming approach is illustrated using the digraph from figure 3: One by one, each protein is added to the initially empty total number of present proteins, represented by the random variable $N^{(0)}$. Thus, the probability distribution for each partial sum $N_i = X_1 + X_2 + \dots + X_i$ is computed and stored in the vector P_{N_i} . Finally, the shared evidence $D^{(S)}$ is included, as it depends exclusively on the number of present proteins $N = X_1 + X_2 + \dots$. Inference for a particular protein (e.g. protein X_1) could be performed easily by performing another forward pass with the constraint that $X_1 = 1$, and all protein posteriors would thus be computed in cubic time with the number of proteins (k proteins $\times O(k^2)$ steps per protein); however, a subsequent right-to-left pass could be used to compute all protein posteriors in $O(k^2)$ time via the forward-backward algorithm.
doi:10.1371/journal.pone.0091507.g005

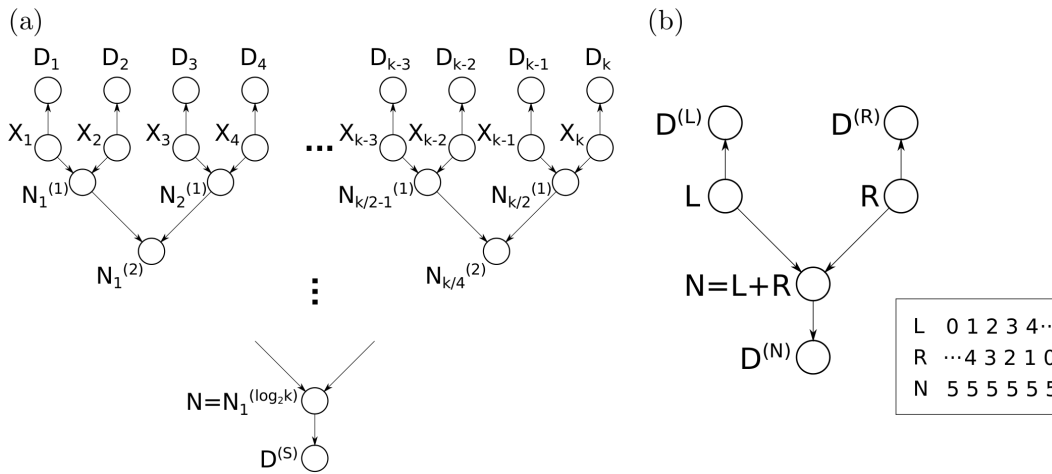


Figure 6. Faster dynamic programming using the convolution tree. (a) An alternate transformation for efficiently computing posteriors for all proteins. Instead of unrolling the commutative and associative operator one protein at a time as performed by the quadratic dynamic programming algorithm, variables are paired successively, resulting in a tree with depth $\log_2(k)$ (when k is a power of 2). (b) Inference on this tree can be performed by solving a minimal ternary node structure and then proceeding inductively: all nodes (except for the proteins themselves) have two parent subtrees, $(L, D^{(L)})$ and $(R, D^{(R)})$, and one child. The parent subtrees connect the node of interest to all data reachable through the parents above (partitioned into $D^{(L)}$ and $D^{(R)}$, respectively), and the child subtree connects to all data reachable below, (denoted $D^{(N)}$). The joint probability with all data above can be passed as messages from parents to children, and the likelihoods given data below (that is, all data reachable through a downward edge out of a given node) can be passed upward from child to parents. Each of these three messages turns out to be a convolutions (shown in inset). For example, all ways that $N = 5$ can be computed by a shifted and reflected dot product, which finds all L and R with a sum of 5. Thus the prior probability for N can be seen as a vector equal to the convolution of the prior probabilities of prior probabilities for L and R . These convolutions can be performed with fast Fourier transform (FFT) in $i \log(i)$ time (where i is the size of the possible state space of N). If the vectors are very sparse, then a standard discrete Fourier transform-based (DFT) convolution may be faster. doi:10.1371/journal.pone.0091507.g006

“noisy-or” model with symmetric parameters) produces independent nodes when a peptide is *not* observed (*i.e.* when a peptide has a zero probability of matching any observed spectra). Thus, these zero-score shared peptides found in $D^{(S)}$ can be copied so that

each protein has its own unique copy. If all shared peptides from $D^{(S)}$ have zero scores, the resulting graph can thus be solved in linear time because all proteins are now independent. Unfortunately, as was the case with clustering, this optimization fails on

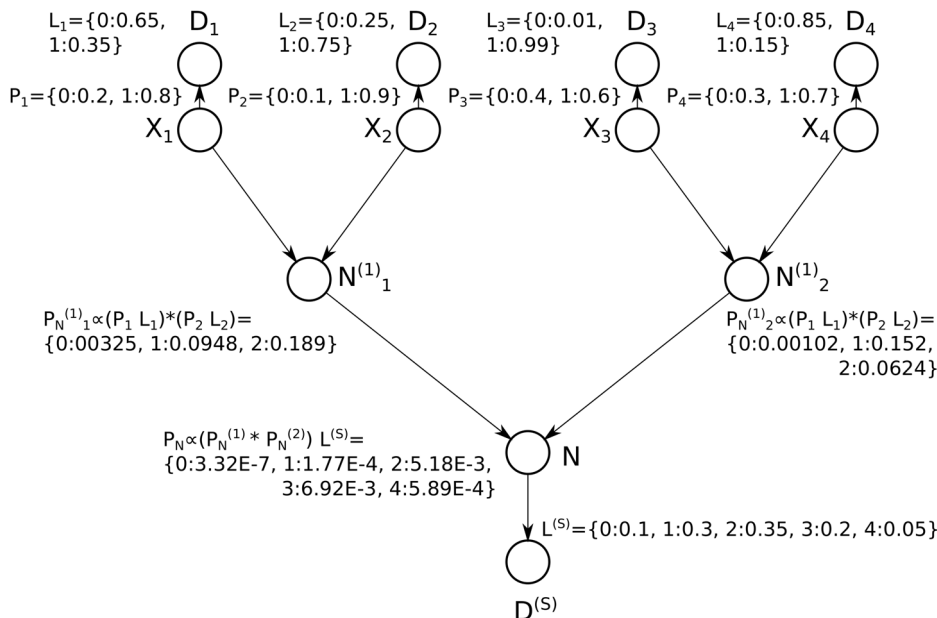


Figure 7. Illustration of the probabilistic convolution tree. The convolution tree is illustrated using the digraph from figure 3 and figure 5: Messages are passed down the tree (via step 1). A subsequent pass would send messages up the tree (step 2), computing the protein posteriors in sub-quadratic time. Note that the normalized vector P_N is equivalent to the distribution $\forall n, \Pr(D, N = n)$, and is identical to the normalized vector of the same name computed by the quadratic algorithm illustrated in figure 5. doi:10.1371/journal.pone.0091507.g007

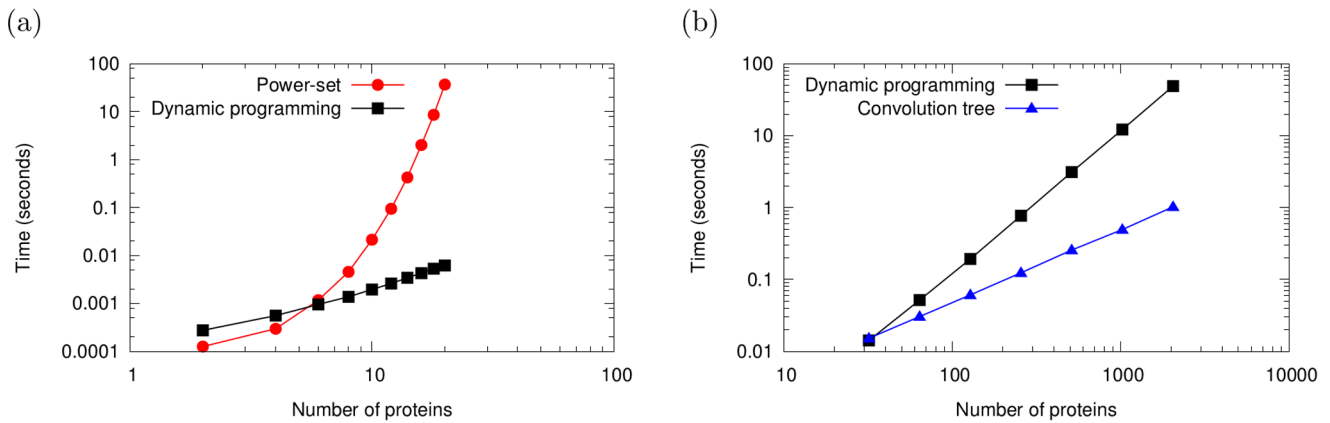


Figure 8. Runtime comparison between the algorithms. (a) A comparison of power-set enumeration and the quadratic dynamic programming approach on small problems of the form from figure 2. Note that axes are log-scaled, and so a widening gap between the curves indicates a super-linear speedup for the algorithm producing the lower curve. (b) A comparison of quadratic dynamic programming and the convolution tree approach on larger problems of this form. The convolution tree achieves a super-linear speedup and a super-linear reduction in memory consumption, making it applicable to much larger problems than either the quadratic dynamic programming approach or power-set enumeration. On very small problems (requiring substantially less than one second of runtime), the more sophisticated dynamic programming approaches have higher overhead, and are therefore slightly slower. doi:10.1371/journal.pone.0091507.g008

figure 2 except in the rare case when *all* peptides in $D^{(S)}$ have zero scores (in practice, it is rare that all such peptides will have zero scores on a large problem). Approximations of the posterior probabilities can be computed by pruning or removing some select peptides that have nonzero scores; however, this can result in lower accuracy when searching spectra against complex protein databases, which have many more such shared peptides [3].

As noted above, the junction tree algorithm, clustering optimization, and pruning optimization do not solve the problem from figure 2 in sub-exponential time; on the contrary, the most efficient exact Bayesian algorithm that has been demonstrated for this type of spliciform graph enumerates the power-set [8]. Furthermore, clustering relies heavily on the assumptions in the original model, and requires that the protein prior γ must be the same for all proteins, and the peptide emission probability α must be the same for all peptides (and the method used to duplicate pruned peptides in the supplement of [13] assumes the noise model β is identical for all peptides). In a similar manner, pruning is tied to the use of noisy-or peptide nodes. Likewise, other optimizations are rigidly tied to specific graph topologies. For example, inference can be performed in linear time on polytrees that exclusively use noisy-or nodes by decomposing the noisy-or

nodes in an iterative fashion and performing belief propagation [6,14]; however, the graph from figure 2 is a polytree only when there is exactly one node of shared evidence in $D^{(S)}$ (for completeness, when there are *zero* nodes of shared evidence, the graph can also be solved efficiently by partitioning it into a collection of disjoint polytrees).

The Need for Further Optimization of Inference

Because the optimization strategies mentioned above are limited to certain graphs or require all peptides to use identical parameters α and β , it can be difficult to use these optimization techniques to realize efficient inference using a modified or extended protein inference model: for example, the next generation of models could include arbitrary categorical priors on how many proteins are present or peptide-specific emission and noise models, which may even depend on the number of present predecessor proteins (rather than constraining that predecessors contribute in a simple multiplicative manner as modeled by noisy-or nodes). These more general priors and emission models can be used to more objectively model the process by which splice variants data are produced by mass spectrometric analysis, and thus permit inference techniques that more accurately model the manner

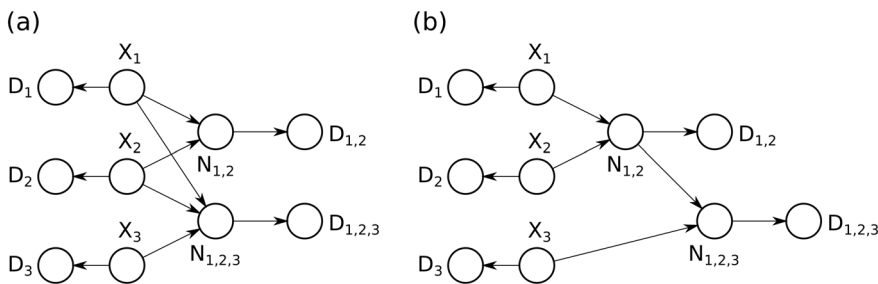


Figure 9. A graph and its cascaded equivalent. (a) A Bayesian network with probabilistic adder nodes $N_{1,2}$ and $N_{1,2,3}$. (b) The resulting cascaded graph of probabilistic adder nodes transforms the graph into an equivalent Bayesian network that can be solved efficiently as a convolution tree. Graphs that do not cascade into polytrees (*i.e.* graphs that have loops even after cascading nodes as shown here) can be solved with a slightly modified junction tree inference algorithm: junction tree clique nodes that consist of a single probabilistic adder node and its inputs can pass messages through convolution tree nodes (without realizing the full conditional probability distribution). doi:10.1371/journal.pone.0091507.g009

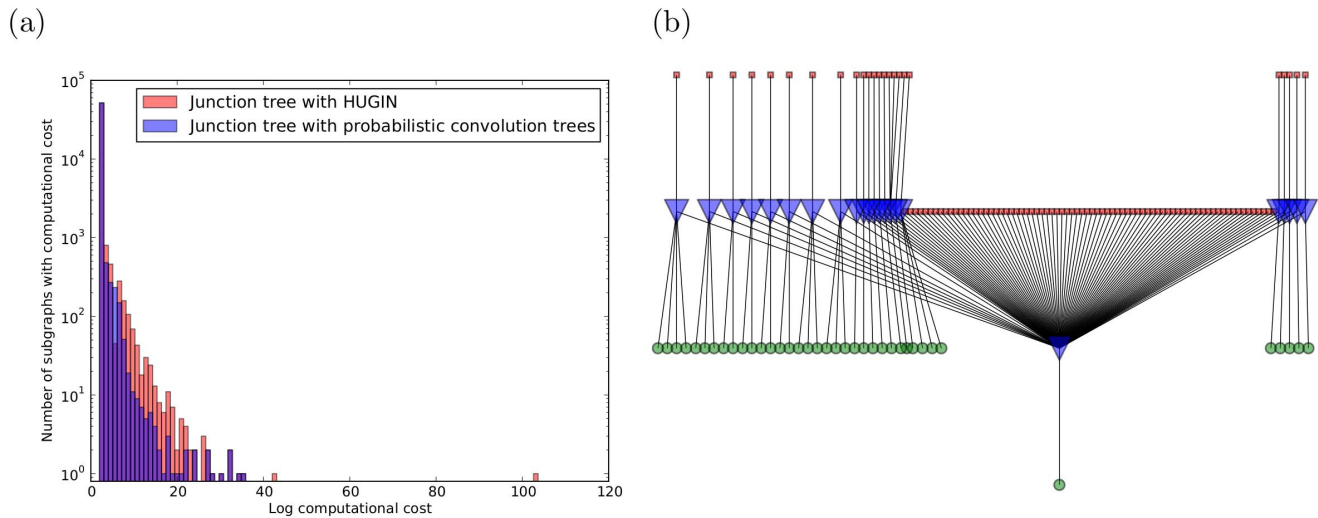


Figure 10. Runtime benefit of convolution tree-based junction tree over HUGIN-based junction tree on HeLa data. (a) Distribution of log runtimes for different connected subgraphs (24 fractions). A HUGIN-based junction tree implementation is compared to a probabilistic convolution tree-based junction tree implementation. The cost of inference will be dominated by a few outlier graphs, which do not decompose effectively using the junction tree. As a result, some connected subgraphs would require an impractical number of steps when using the HUGIN algorithm. This runtime can be improved by using probabilistic convolution trees, while still achieving the exact result. (b) One difficult connected subgraph from panel (a). Proteins are shown as red squares, probabilistic adders are shown as blue inverted triangles (these are the nodes can make use of probabilistic convolution trees), spectral evidence is shown as green circles. This subgraph would require $2^{103.629}$ steps using the HUGIN junction tree. In contrast, the probabilistic convolution can solve this same subgraph in $2^{12.045}$ steps (and achieve an exact result). doi:10.1371/journal.pone.0091507.g010

with which proteins compete for shared peptide data. For instance, perhaps a sixth present predecessor protein does not substantially increase the peptide's chances compared to five present proteins, or perhaps genetic or biochemical analysis has demonstrated that it is improbable for the gene in question to express more than 7 splice forms in a short period of time. Such extensions can be implemented with *probabilistic adder* nodes, which add two probabilistic quantities. As we demonstrate in this paper, these probabilistic adder nodes can be more flexible and general than symmetric noisy-or nodes (symmetric noisy-or nodes can be trivially emulated using probabilistic adder nodes, but the converse is not true). Generalizing to probabilistic adder models offers a substantial increase in model flexibility. Because they are more general than noisy-or nodes, more problems can be decomposed into a polytree of probabilistic adder nodes (*e.g.* the graph in figure 2, even when using peptide-specific emission and noise parameters).

In this paper we present a dynamic programming method for computing exact posteriors for efficiently solving the problem in figure 2 using a more flexible probabilistic adder model. The first algorithm presented performs inference in quadratic time and quadratic space, which is compatible with a peptide-specific emission model, with a peptide-specific noise model, and with non-identical protein priors. We then generalize the algorithm we use to reveal it is an instance of the transform proposed by Heckerman[15–17].

Lastly, we propose the probabilistic convolution tree, a data structure which uses dynamic programming to perform exact inference on polytrees of probabilistic adder nodes in $O(k \log(k)^2)$ time and $O(k \log(k))$ space, and compare performance of the three algorithms, power-set enumeration, quadratic dynamic programming, and the probabilistic convolution tree, on problems of the form shown in figure 2. The probabilistic convolution tree is applicable to more general graphs, can be combined with junction

tree inference, and can even be used to efficiently compute posteriors when using an arbitrary categorical prior on the number of present variables.

Materials and Methods

Using graphs of the type shown in figure 2, we compare the three exact inference methods described in this paper that can be applied with peptide-specific probabilities: power-set enumeration, quadratic dynamic programming, and the proposed probabilistic convolution tree approach.

For each problem of a given size k , each protein is adjacent to a unique peptide and all proteins are adjacent to between 1 and 9 shared peptides (chosen using uniform integers in $\{1, 2, \dots, 9\}$). Each unique peptide j has its own α_j and β_j chosen uniformly $\in [0.1, 0.9]$ and a random probability score (*i.e.* the probability it matched an observed spectrum) chosen uniformly $\in [0.1, 0.9]$. Each protein i has a protein-specific prior γ_i chosen uniformly $\in [0.01, 0.99]$ (this is useful to test the numeric stability of the algorithm, because the only division operation is based on this collection of protein priors). Lastly, the shared data $D^{(S)}$ has an arbitrary categorical distribution in the number of present proteins $N = \sum_i X_i$ (each entry chosen randomly in $[0, 1]$ and normalized to sum to 1).

Runtimes were compared using python implementations of each of the three algorithms on a Core i3 laptop. For all problems timed, the three methods achieve identical results (to the 8 decimal points that by default are printed out by python's numpy package).

Results

The Brute-force Algorithm: Power-set Enumeration

Power-set enumeration is quite simple: the likelihood is computed for every possible joint state for all proteins $X = x$.

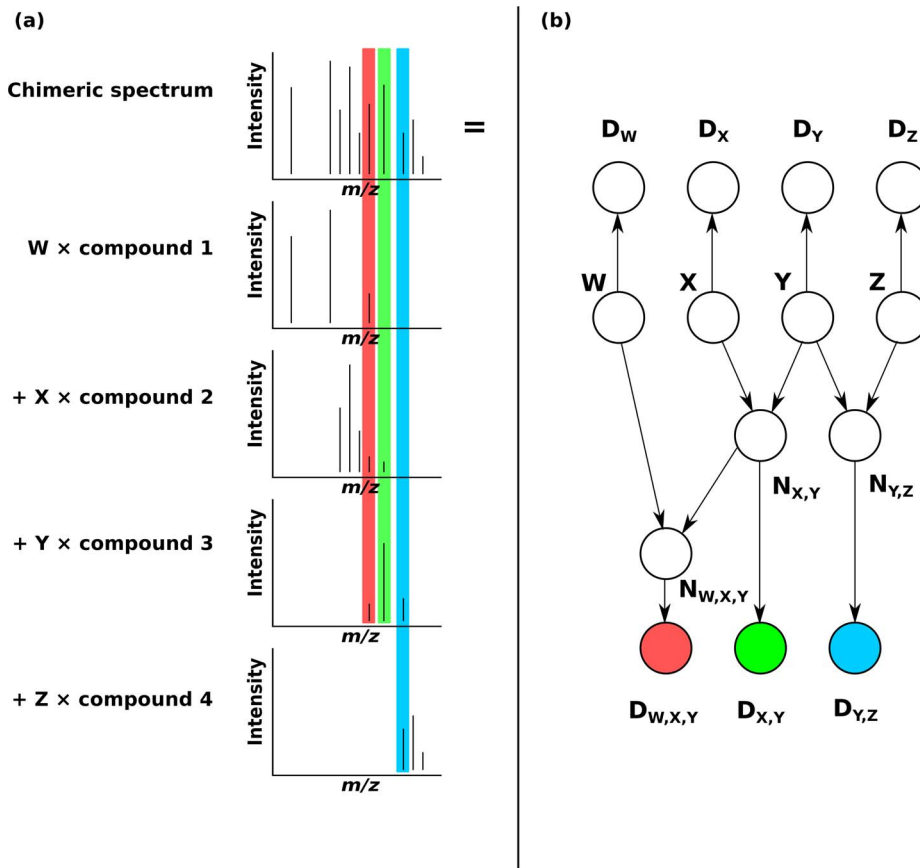


Figure 11. Decomposition of a spectrum into its constituent compounds. (a) A probabilistic demixing (a problem highly related to deconvolution) problem from mass spectrometry. An observed chimeric spectrum with data D is composed of a linear combination of four different compounds and with unknown relative abundances W, X, Y, Z , which we want to infer. Three m/z values that can receive contributions from multiple compounds are labeled with the background colors red, green, and blue. (b) The resulting cascaded graph of probabilistic adder nodes. The variables W, X, Y, Z are discretized into relative abundances of interest. Conditional probabilities individually treat each intensity as proportional to the abundance of the compound that produces it. Data unique to each compound are labeled D_W, D_X, D_Y, D_Z , and are conditionally independent given W, X, Y, Z . Shared evidence nodes are colored to correspond to the background colors from (a). Probabilistic adder nodes are cascaded to build a tree for probabilistic inference, enabling the computation of a posterior distribution for the relative abundance of each compound.
doi:10.1371/journal.pone.0091507.g011

$$\begin{aligned} \Pr(X_1 = x_1 | D) &\propto \Pr(D, X_1 = x_1) \\ \Pr(D, X_1 = x_1) &= \sum_{x_2} \sum_{x_3} \cdots \sum_{x_k} \Pr(D, X_1 = x_1, X_2 = x_2, \dots, X_k = x_k) \\ \Pr(D, X_1 = x_1, X_2 = x_2, \dots, X_k = x_k) &= \Pr(D^{(S)} | N = \sum x) \prod_i \Pr(D_i | X_i = x_i) \end{aligned}$$

Illustrative example of the exponential enumeration algorithm. Figure 3 illustrates this super-exponential inference algorithm on the small graph of the type shown in figure 2. Four proteins with different prior probabilities and unique spectral evidence are marginalized via brute force: for each of the 2^4 protein configurations, $\Pr(X = x, D)$ (the joint probability of the protein configuration x and all observed data) can be enumerated in super-exponential time (*i.e.* it takes exponential time for each protein multiplied by the number of proteins involved). The values in the column containing $\forall x, \Pr(X = x, D)$ can then be

used to compute marginals for a particular protein (*i.e.* protein X_1). This is accomplished by separately computing the denominator and then the numerator from equation 1. The denominator can be computed by summing over all values in the column (*i.e.* $\Pr(D) = \sum_x \Pr(X = x, D)$), and then the numerator for the protein X_1 can be found by computing the sum of values in the same column, but only where $X_1 = 1$ (*i.e.* $\Pr(D, X_1 = 1) = \sum_{x: x_1 = 1} \Pr(X = x, D)$). A small savings is achieved by passing over the table one time to compute the total sum, ($\Pr(D)$ used in the denominator of equation 1), as well as the constrained sum (*i.e.* the numerator) for each protein, thereby computing each value in the column $\forall x, \Pr(X = x, D)$ only once. Thus the cost of computing posterior probabilities for k proteins is in $O(k2^k)$ time (k proteins $\times 2^k$ steps per protein).

Time and space requirements. Power-set enumeration can be performed in $O(2^k k)$ time: there are 2^k protein configurations, each of which is applied to a running total of joint probabilities for k variables. Despite its inefficient runtime, power-set enumeration requires only $O(k)$ space (enumeration of the power-set can be performed by incrementing a large base-2 number, and therefore does not require storing the entire power-set).

A Quadratic Dynamic Programming Algorithm for Exact Inference

Figure 4a demonstrates the idea behind the quadratic dynamic programming algorithm: Each protein is added (with its state still a random distribution) one at a time, resulting in layers of partial sums. These partial sums work toward building the sum of the present proteins $N = \sum_i X_i$. At each layer i in the table, we compute the distribution on the random variable N_i (the random variable for the partial sum defined by $N_i = \sum_{j \leq i} X_j$). Because the variable X_i is still unknown, every possible values of X_i is associated with edges connecting cells in layer $i-1$ to the appropriate cell in layer i . For example, the an edge labeled $X_1 = 1$ connects the 0 row of N_0 (i.e. the event that the random variable $N_0 = 0$) to the 1 row of N_1 (i.e. the event that the random variable $N_1 = 1$), because $N_i = N_{i-1} + X_i$. Each edge for $X_i = x_i$ is weighted by the product between the prior that the protein has that assignment and the unique likelihood contribution that arises from that protein taking that assignment:

$$e_{i,x_{i+1}} = \Pr(X_{i+1} = x_{i+1}) \Pr(D_{i+1} | X_{i+1} = x_{i+1}).$$

These layers are built so that the final layer is the cumulative sum of all proteins: $N_k = N_{k-1} + X_k = N_{k-2} + X_{k-1} + X_k = \dots = \sum_i X_i = N$. In this manner, all possible paths to arrive at every possible cumulative sum $N = n, 0 \leq k \leq n$ are available in the graph shown in figure 4a.

After the graph is initialized, two passes are performed: one from the left and one from the right (these passes are sometimes denoted a “forward-backward” algorithm, a special class of junction tree message passing on a path graph commonly used by hidden Markov models (HMMs)). Denote row j of layer i as $N_i[j]$, which indicates that $\sum_{m \leq i} X_m = j$. The pass from the left computes the marginal probability of arriving at this given cell through all possible weighted left-to-right paths, denoted $N_i[j].fromLeft$ for row j of layer i . The algorithm starts with $N_0[0].fromLeft \leftarrow 1$, because N_0 has a 100% probability that it is 0 (the outcome for the layer is indicated by the row number). These values are used to fill the entire table by propagating left-to-right in the following manner:

$$N_i[j].fromLeft \leftarrow \sum_{m, x_i: m+x_i=j} e_{i-1, x_i} N_{i-1}[m].fromLeft.$$

The second pass, from right to left, is performed in an almost identical manner. For each value in the table, we compute the marginal probability of arriving at this cell through all possible weighted right-to-left paths. Before starting the right-to-left pass, every node in layer k is initialized with the appropriate likelihood due to the shared data: $N_k[j] \leftarrow \Pr(D^{(S)} | N = j)$. Then the same propagation is performed, but from right to left:

$$N_i[j].fromRight \leftarrow \sum_{m, x_{i+1}: m-x_{i+1}=j} e_{i, x_{i+1}} N_{i+1}[m].fromRight.$$

The result is that every cell in the table computes the marginal probability of all paths passing through it from left to right and from right to left (this is a standard “forward-backward” algorithm). The likelihood of being in a given node can be computed by $N_i[j].likelihood = N_i[j].fromLeft \times N_i[j].fromRight$.

Thus, the posterior that any variable is in a certain state $X_i = x_i$ is computed by the total weight of all paths that pass through edges assigning $X_i = x_i$:

$$\Pr(X_i = x_i | D) \propto \sum_j N_{i-1}[j].fromLeft e_{i-1, x_i} N_i[j + x_i].$$

For the three preceding equations, out-of-bound indices (e.g. querying $N_1[-1].fromLeft$ and $N_1[-1].fromRight$) should return zero.

Note that figure 4a is drawn in a way that underscores its applicability to inference problems where each variable has more than two possible states (as indicated by more than two edges radiating out of each node, and with labels not limited to binary states).

Figure 4b generalizes this approach and merges each layer $N_i[0], N_i[1] \dots$ into a single variable N_i . The result is a graph with a tree decomposition that is visibly simple; in fact, it closely resembles a hidden Markov model (HMM), for which the tree decomposition is trivial using a similar forward-backward algorithm [18]. This graph generalizes the dynamic programming performed in figure 4a so that it can be applied to similar problems by transforming the graph and then performing junction tree inference. The generalized algorithm is an instance of Heckerman’s transform [15,15,17], which is sometimes referred to as a “temporal transform”.

From the generalized figure 4b, it is easy to understand the algorithm described above: in figure 4a, passing through the node $N_i^{(j)}$ indicates that the partial sum $N_i = \sum_{m < i} X_m = j$. For this reason, we can see that propagation from the left accumulates the priors and unique likelihood contributions that would lead to $N_i = j$.

$$\begin{aligned} & \Pr(D_1, D_2, \dots, D_i, N_i = n_i) = \\ & \sum_{n_{i-1}} \sum_{x_i} \Pr(N_i = n_i | N_{i-1} = n_{i-1}, X_i = x_i) \\ & \Pr(D_1, D_2, \dots, D_{i-1}, N_{i-1} = n_{i-1}) \Pr(D_i | X_i = x_i) \Pr(X_i = x_i) \end{aligned}$$

Likewise, propagation from the right computes the remaining likelihood terms, along with the shared likelihood:

$$\begin{aligned} & \Pr(D_{i+1}, D_{i+2}, \dots, D_k, D^{(S)}, N_i = n_i) = \\ & \sum_{n_{i+1}} \sum_{x_{i+1}} \Pr(N_i = n_i | N_{i-1} = n_{i-1}, X_i = x_i) \\ & \Pr(D_{i+2}, \dots, D_k, D^{(S)}, N_{i+1} = n_{i+1}) \\ & \Pr(D_{i+1} | X_{i+1} = x_{i+1}) \Pr(X_{i+1} = x_{i+1}). \end{aligned}$$

For both equations above, we exploit the fact that $\Pr(N_i = n_i | N_{i-1} = n_{i-1}, X_i = x_i)$ is 1 if and only if $n_i = n_{i-1} + x_i$ (and is otherwise 0). For each of the two equations above, this allows us to collapse the nested sum into a single sum over the variable with a smaller domain (if all variables X_i are binary, as in the case of distinguishing splice variants, then x_i will be have a smaller domain than n_i).

Lastly, we see that the posterior for a protein, which is always proportional to its joint probability with the data, is computed by

the forward-backward probability passing through edges labeled $X_i = x_i$:

$$\begin{aligned} \Pr(X_i = x_i, D) &= \\ &\sum_{n_{i-1}, n_i} \Pr(N_i = n_i | N_{i-1} = n_{i-1}, X_i = x_i) \\ &\Pr(D_1, D_2, \dots, D_{i-1}, N_{i-1} = n_{i-1}) \\ &\Pr(D_{i+1}, D_{i+2}, \dots, D_k, D^{(S)}, N_i = n_i) \\ &\Pr(D_i | X_i = x_i) \Pr(X_i = x_i), \end{aligned}$$

which is given by the algorithm.

Illustrative example of the quadratic algorithm. Figure 5 illustrates the forward pass of the quadratic inference algorithm on the small graph shown in figure 3. The posterior probability for an arbitrary protein (e.g.n X_1) can be easily found by passing the messages left-to-right once (to compute the denominator of equation 1), and then by passing passing messages left-to-right once more with the constraint that $X_1 = 1$ (to compute the numerator of equation 1); however, by using right-to-left pass (i.e. the “forward-backward” algorithm), the marginal probabilities of all proteins can be computed in roughly the same amount of time required to compute one protein posterior in this manner (i.e. quadratic time).

Time and space requirements. This approach (and, in general, Heckerman’s transform) reduces the runtime to $O(k^2)$, but the table used for dynamic programming uses $O(k^2)$ space. This time-space trade-off can be considered almost universally favorable compared to power-set enumeration: for problems when k^2 would become too large to store in the RAM of a modern computer, k would be so large that $2^k k$ would result in an astronomically large runtime for power-set enumeration; however, space will likely be the limiting factor in applying Heckerman’s transform, and so like other $O(k^2)$ algorithms, it cannot be applied as-is to very large problems as mentioned in [19]; for completeness it should also be noted that the VE1 algorithm [19] is not well-suited to the problem in figure 2 because it is query based, meaning that it performs best when only a small subset of the variables need to be solved.

The Probabilistic Convolution Tree: a more Efficient Dynamic Programming Algorithm

The probabilistic convolution tree, shown in figure 6 is similarly motivated to the dynamic programming shown in figure 4; however, where the quadratic dynamic programming algorithm constructs a chain $N_0 = 0, N_1 = X_1 + N_0, N_2 = N_1 + X_2, \dots, N_k = N_{k-1} + X_k$, the probabilistic convolution tree proposes a divide-and-conquer approach. First we present an algorithm applicable when k is a power of 2, and then generalize it. For this reason, all logarithms are base 2: $\log = \log_2$.

Probabilistic adder nodes are partitioned into multiple layers: $N^{(i)} = (N_1^{(i)}, N_2^{(i)}, \dots)$, where $N_j^{(i)} = N_{2j-1}^{(i-1)} + N_{2j}^{(i-1)}$. In this manner, every node in the tree from layer 1 $\dots \log(k)$ has exactly one child and two parents. Layer 0 is composed of the parent-free variables X_1, X_2, \dots, X_k ; each nodes X_i in this layer has two children: their unique data D^i and the probabilistic adder node in layer one, $N_{[i/2]}^{(1)}$, which depends on them.

Because of this consistent structure after layer 0, the entire tree can always be viewed from a single probabilistic adder node N as

having two parents L and R as parents. These parents, along with the data reachable through them, constitute subtrees $(L, D^{(L)})$ and $(R, D^{(R)})$. Each node has a single child subtree containing all data below $D^{(N)}$ (Figure 6). $D^{(N)}$ includes the shared data $D^{(S)}$ as well as any other data nodes reachable through the edge down from the node. For instance, the tree as seen from the perspective of node $N_1^{(1)}$ has $L = X_1, R = X_2, D^{(L)} = D_1, D^{(R)} = D_2$, and $D^{(N)} = (D_3, D_4, \dots, D_k, D^{(S)})$. Because this ternary structure is ubiquitous in the tree, a message passing algorithm can be constructed by using only the subtree from figure 6b.

Example: computing prior probabilities. As an example of the motivating idea behind the probabilistic convolution tree, first observe that the prior probability of a node can be constructed easily using the prior probabilities of its parents:

$$\begin{aligned} \Pr(N = n) &= \\ &= \sum_{\ell} \sum_r \Pr(L = \ell) \Pr(R = r) \Pr(N = n | L = \ell, R = r). \end{aligned}$$

And because $N = L + R$, this can be condensed to a single summation:

$$\Pr(N = n) = \sum_{\ell} \Pr(L = \ell) \Pr(R = n - \ell)$$

Priors are known for all nodes in layer 0, and so it is clear by induction that a prior can be computed for every node (layer 0 is the base case and every prior in the next layer can be computed given the priors from the previous layer).

However, as written above, computing the prior for every node in the tree would be at best a constant speedup over the quadratic dynamic programming approach, because the two nodes in second to last layer will have state space $k/2$ and $k/2$, and all pairs must be combined to form the prior on the final node, $N_1^{(\log_2 k)}$ (enumerating these combinations requires quadratic time). However, we can now employ convolution at each node (see figure 6b inset), and so it can be computed in $m \log(m)$ time, where m is the state space of the node N being updated. Note that when the vectors are very sparse, the convolution can be performed more efficiently using a standard discrete Fourier transform (DFT) rather than a fast Fourier transform (FFT). Also note that a vector that can easily be decomposed into a sum of dense vectors whose nonzero indices do not overlap, then Fourier transformation can be performed using a hybrid approach on the results of the decomposition. If a probabilistic statement with a single unassigned variable (e.g. $\Pr(N)$, $\Pr(L)$, or $\Pr(R)$) is considered as a vector (with all outcomes of that variable enumerated in order), then the summation described in the equation above is equivalent to

$$\Pr(N) = \Pr(L) * \Pr(R),$$

where $*$ is the vector convolution operator.

We now proceed to outline the convolution tree algorithm, which operates in a similar manner as this example.

Step 1: passing messages down. First, the convolution tree algorithm computes the joint probability of each node with the data above it. This can be performed almost identically to computation of prior probabilities in the example above. Also, like above, the base case is known for all nodes in the layer 0 (it is

simply the element-wise product between the node’s prior and its likelihood from unique data). Thus in the same manner presented in the example above, the joint probability of the node’s state with all data above, $\Pr(D^{(L)}, D^{(R)}, N)$, can be computed using a convolution:

$$\begin{aligned} \Pr(D^{(L)}, D^{(R)}, N) &= \sum_{\ell} \Pr(D^{(L)}|L=\ell) \\ &\quad \Pr(D^{(R)}|R=n-\ell) \Pr(L=\ell, R=n-\ell, N=n) \\ &= \sum_{\ell} \Pr(D^{(L)}|L=\ell) \Pr(D^{(R)}|R=n-\ell) \\ &\quad \Pr(L=\ell) \Pr(R=n-\ell) \\ &= \sum_{\ell} \Pr(D^{(L)}, L=\ell) \Pr(D^{(R)}, R=n-\ell) \\ &= \Pr(D^{(L)}, L) * \Pr(D^{(R)}, R). \end{aligned}$$

Like the example above where prior probabilities are computed, each node in layer 0 has a known joint probability with the data above. And so, proceeding layer-by-layer, for each node we compute the joint probability of that node with data above it.

Step 2: passing messages up. The second part of the convolution tree algorithm is to pass messages upward through the tree after all messages have been propagated downward (*i.e.* after completion of step 1). Where step 1 passes down the joint probability of each node with the data above it, step 2 computes the likelihood given all data that can be reached below: for the left parent, we compute $\Pr(D^{(R)}, D^{(N)}|L)$ and for the right parent we compute $\Pr(D^{(L)}, D^{(N)}|R)$. Note that we only need values proportional to these probabilities, and so they can be normalized before proceeding to the next layer in the tree for better numeric stability. These messages can be defined thus:

$$\begin{aligned} \Pr(D^{(R)}, D^{(N)}|L=\ell) &= \sum_n \Pr(D^{(N)}|N=n) \Pr(D^{(R)}|R=n-\ell) \Pr(R=n-\ell) \\ \Pr(D^{(L)}, D^{(N)}|R=r) &= \sum_n \Pr(D^{(N)}|N=n) \Pr(D^{(L)}|L=n-r) \Pr(L=n-r). \end{aligned}$$

Where the forward pass (in step 1) represents addition, this backward pass represents subtraction, which presents also convolution, but because one operand is negated, its indices are reversed. Also, the result of the convolution will not start at zero, but will instead start at the minimum value achieved by the subtraction. In this case, we do not want the negative values because they are impossible in the source variable. Thus the above equations can be written as convolutions where one vector is reversed and where a slice is taken from the result (to undo the shift and remove impossible outcomes). So the final results can be computed

$$\begin{aligned} \Pr(D^{(R)}, D^{(N)}|L) &= (\Pr(D^{(R)}, R)[::- -1] * \Pr(D^{(N)}|N)) \\ &\quad [\text{len}(R)-1 : \text{len}(R)-1 + \text{len}(L)] \\ \Pr(D^{(L)}, D^{(N)}|R) &= (\Pr(D^{(L)}, L)[::- -1] * \Pr(D^{(N)}|N)) \\ &\quad [\text{len}(L)-1 : \text{len}(L)-1 + \text{len}(R)], \end{aligned}$$

where $\text{len}(R)$ and $\text{len}(L)$ depict the number of states for R and L , $\text{vector}[:, : -1]$ reverses the vector (*i.e.* it reflects it), and $v[a : b] = (v_a, v_{a+1}, \dots, v_{b-1})$ slices the vector to remove the unwanted shift mentioned in the paragraph above.

As messages were propagated downward before, these messages can be passed back up the tree to form the likelihood given data below for the left and right parent nodes. For example, for the left parent L , its likelihood given all data below will be initialized by the message passed back, $\Pr(D^{(R)}, D^{(N)}|L)$. Now that the node L has its likelihood given the data below, it can pass messages up the tree. This process continues until all messages have been passed up to layer 0, which contains the proteins themselves.

The posterior for node $N=n$ can be computed by multiplying the joint probability with data above by the likelihood given data below that node:

$$\Pr(D, N=n) = \Pr(D_{\text{above}(N)}, N=n) \Pr(D_{\text{below}(N)}|N=n).$$

Similarly, the posterior for protein $X_i = x_i$ can be computed by multiplying the joint probability with data above (for each protein X_i , the data above consists only of its unique spectral evidence D_i) by the likelihood given data below:

$$\Pr(D, X_i = x_i) = \Pr(D_i, X_i = x_i) \Pr(D_{\text{below}(X_i)}|X_i),$$

which is proportional to $\Pr(X_i = x_i|D)$. Note that when a value is proportional to the posterior probability, then the posterior proportionality can be computed by dividing by the sum:

$$\Pr(X_i = x_i|D) = \frac{\Pr(D, X_i = x_i)}{\sum_{x'_i} \Pr(D, X_i = x'_i)}. \quad (1)$$

Like the quadratic dynamic programming algorithm, the convolution tree algorithm described here is applicable to variables with more than two states. It also does not require that the left and right parents of a probabilistic adder node have identical state space, although of course an FFT-based convolution will be faster when this is the case, because it will not need to pad the shorter vector with zeros.

Illustrative example of the probabilistic convolution tree algorithm. Figure 7 illustrates the convolution-based inference algorithm on the a small graph of the type shown in figure 2. Messages are passed down (via step 1). The posterior probability for an arbitrary protein (*e.g.*n X_1) can be easily found by passing the messages down once (to compute the denominator of equation 1), and then by passing messages downward once more with the constraint that $X_1 = 1$ (to compute the numerator of equation 1). however, step 2, which subsequently passes messages back up the tree, reuses the shared computations for these proteins, and computes the posterior for every protein in roughly the time necessary to compute the posterior for any single protein (sub-quadratic time).

Time and space requirements. As implemented, the runtime of the convolution tree can be shown to be $O(k \log(k)^2)$, and the space requirement can be shown to be $O(\log(k))$. Sparse vectors can be convolved more quickly by using the discrete Fourier transform (DFT).

A fixed number of convolutions are performed at all nodes. Layer i will have $\frac{k}{2^i}$ nodes, and the length of the state space for that

convolution is $\leq 2^i$. Thus the cost of FFT convolution for a node at layer i is $2^i \log(2^i)$. The entire runtime can then be computed easily using the expansion $\sum_i^n i = \frac{n(1+n)}{2}$:

$$\sum_{i=1}^{\log(k)} \frac{k}{2^i} 2^i \log(2^i) = k \sum_{i=1}^{\log(k)} i = \frac{k \log(k)(1 + \log(k))}{2},$$

which is on the order of $k \log(k)^2$.

The memory consumption is computed with a similar strategy. The simple implementation of the convolution tree method described here requires $O(k \log(k))$ space to store the two vectors $\Pr(D_{above}, N_j^{(i)})$ and $\Pr(D_{below} | N_j^{(i)})$ for each node $N_j^{(i)}$ in the tree, because at layer i , the length of the vector storing each of these is $\leq 2^i$, and there are $\frac{k}{2^i}$ such nodes in that layer. Thus the total space requirement is

$$\sum_{i=1}^{\log(k)} \frac{k}{2^i} 2^i = k \sum_{i=1}^{\log(k)} 1 = k \log(k).$$

The method can be easily extended when k is not a power of two by simply adding dummy variables X_{k+1}, X_{k+2}, \dots with 100% prior probability of being absent (thus they do not influence the sum) and with no unique evidence). These variables are added until the total number of variables is a power of 2. Using a 0 prior is important for this approach, because it prevents $\sum_i X_i$, and subsequently $\Pr(D^{(S)} | X)$, from being altered by including these dummy variables. Even though this approach is inelegant, it can be easily seen to produce the same order runtime when k is not a power of 2, because in the worst case, k must be roughly doubled to perform inference, which would simply change the constant, but not the order of the runtime.

Extension to linear functions on the integers. The special case of computing posteriors where $D^{(S)}$ depends on N with $N = X_1 + X_2 + \dots + X_k$ (as described in this paper), can be easily generalized to compute posteriors when $N = s_1 X_1 + s_2 X_2 + \dots + s_k X_k$ for fixed integer scaling factors s_1, s_2, \dots, s_k . First observe (without loss of generality) that for any positive integer s_1 , $s_1 X_1$ simply creates a new vector X'_1 with $s_1 - 1$ zeros padded between every entry of X_k :

$$X'_1 [i/s_1] = \begin{cases} X_1 [i], & i/s_1 \text{ is an integer} \\ 0, & \text{else} \end{cases}$$

Second, observe that for any two nodes L and R , the subtraction $L - R$ can be accomplished by reversing R before convolution with L (and recording the fact that the zero index of the resulting array no longer refers to $N=0$, but instead refers to the minimum value achievable by the subtraction). For this reason, scaling X_1 by a negative integer s_1 can be performed by reversing the vector X_1 and then padding with zeros as mentioned above, and then adding normally with the convolution tree (again, in this case, each node in the convolution tree would also keep track of the minimum integer summation value corresponding to the zero index). For completeness, when $s_1 = 0$ (and is thus neither positive nor negative), then the scaled vector $X'_1 = [1.0]$, indicating a 100% probability that X'_k is zero; however, if the fixed value s_1 is known to be zero ahead of time, then there is effectively no edge

connecting X_1 to the scaled summation N , and that input can be ignored with no consequence.

Thus scaling X_1 by s_1 can be accomplished by simply permuting the indices of X_1 (into a potentially larger result vector): the vector X_1 is first reversed if $s_1 < 0$ and is then padded with zeros as described above.

For simple implementation, single-input single-output scaling nodes (with input X_1 and output X'_1 and a fixed parameter s_1) can be used to first transform any X_1 into X'_1 , and then fed into a convolution tree node with $N = X'_1 + X'_2 + \dots + X'_k$. Thus we can model an integer-scaled sum without any modification to the convolution tree data structure. Messages passed backward through these scaling nodes simply undo the deterministic permutation of indices mapping X_1 to X'_1 .

Discussion

Performance on Randomly Generated Problems

Figure 8 shows a runtime comparison between the three algorithms using a more general model, which is not restricted to noisy-or nodes and instead uses probabilistic adder nodes. Figure 8a compares power-set enumeration with the quadratic dynamic programming method on smaller problems ($k \in \{2, 4, 6, 8, \dots, 20\}$). Figure 8b compares quadratic dynamic programming with the convolution tree method on larger problems ($k \in \{32, 64, 128, 256, \dots, 2048\}$). Note that both axes are log-scaled and so a growing gap between the two series represents a super-linear speedup in the runtime. On larger problems (e.g. 4096 proteins), the quadratic dynamic programming runs out of memory on a 4 GB computer.

Both the quadratic dynamic programming and the convolution tree have runtimes far superior to power-set enumeration. But moreover, the convolution tree offers scalability to substantially larger problems than the quadratic dynamic programming approach. For example, computing exact posteriors for 32768 proteins takes only 28.03 seconds, while the quadratic dynamic programming cannot even be run.

Cascading Trees for more General Application

It should be noted that the convolution tree method can easily be applied when including node-specific data $D_j^{(i)}$, which depends only on the node $N_j^{(i)}$ in the tree (as long as the resulting graph is still a tree): The modified method would simply multiply (element-wise) the likelihood given data below $\Pr(D_{below} | N_j^{(i)})$ by the unique likelihood $\Pr(D_j^{(i)} | N_j^{(i)})$ when passing messages up and multiply (element-wise) the joint probability with data above $\Pr(D_{above}, N_j^{(i)})$ by the unique likelihood $\Pr(D_j^{(i)} | N_j^{(i)})$ when passing messages down. This allows nearly identical runtime (point-wise multiplication is cheaper than convolution, which is already performed by the algorithm). On graphs where data is shared in a manner such that it is cascaded (i.e. $D_{\{1,2\}}$ depends on the sum of $X_1 + X_2$, and $D_{\{1,2,3\}}$ depends on the sum of $X_1 + X_2 + X_3$, and so forth), the sums can be arranged by a simple greedy algorithm so that a probabilistic adder node $N_{1,2}$ has predecessors X_1, X_2 and then a second probabilistic adder node $N_{1,2,3}$ has predecessors $N_{1,2}, X_3$ (Figure 9). Thus, cascading makes it possible to use the convolution tree even when the shared data $D^{(S)}$ do not have identical predecessors as shown in figure 2. Furthermore, more general cascading can factor out shared computation so that data $D_{\{1,2,3\}}$, which depends on $X_1 + X_2 + X_3$ and $D_{\{2,3,4\}}$, which depends on $X_2 + X_3 + X_4$ can be factored into $N_{\{2,3\}} = X_2 + X_3$ and where $D_{\{2,3,4\}}$ depends on $N_{\{2,3\}} + X_4$ and

$D_{\{1,2,3\}}$ depends on $X_1 + N_{\{2,3\}}$. Cascading probabilistic adder nodes allows inference in the same runtime and memory usage derived in this manuscript when the cascaded nodes form a tree.

Compatibility with Belief Propagation and Junction Tree Inference

When the graph contains loops (*i.e.* when the cascaded graph does *not* form a tree), variables can be merged into larger joint variables, and then the sum of these variables can then be fed into a convolution tree. Essentially, this demonstrates the potential to use convolution trees as specialized cliques within classic junction tree inference [8]: When inference is performed, the full joint conditional probability table would not be generated for any clique node in the junction tree with two properties: 1) The clique node contains only a probabilistic adder node and all of the probabilistic adder node's predecessors. 2) The edges connecting the specialized clique node to other clique nodes in the junction tree would also need to carry messages of a single variable only, because the convolution tree does not allow arbitrary joint distributions of its inputs as messages passed in. 3) Messages sent along edges into the specialized clique node have disjoint variable sets (*i.e.* each variable occurs along only one message). This last criterion can always be satisfied by inserting a temporary clique node containing only the variable X as an intermediary between the specialized clique node and any other clique nodes that send messages containing the variable X .

Posterior probabilities and messages passed out of a specialized convolution tree clique node would be computed using the convolution tree, and would simply pass the likelihood of all data preceding the edge along which the message is passed (*i.e.* all data found by moving backward against the direction of message passing). For the “head” node (*i.e.* the adder), the message out would be the joint probability above. For any “input” node, the message out would be the likelihood below.

Such “intelligent” junction trees can likewise feature approximation clique nodes for use when large cliques do not meet the requirements to be specialized convolution tree clique nodes, *e.g.* clique nodes that perform mathematical approximations similar to pruning or clique nodes that use sampling methods like Monte Carlo or Markov chain Monte Carlo (MCMC). In this manner, protein inference could be performed using the benefits of three approaches: junction tree inference (which can break apart large connected graphs), convolution tree clique nodes (which can allow the junction tree to perform efficient exact inference when large probabilistic adder cliques are encountered), and approximations (available as a last resort to prevent a single remaining large clique from prohibiting inference on the entire junction tree). Approximations (which can be inaccurate and slow) would thus be avoided when exact answers can be computed efficiently. Such junction trees are examples of the recommended extensions to the collapsed Gibbs sampler mentioned in [8]. In a similar manner, pruning may be performed by first finding expensive cliques in the junction tree (*i.e.* large cliques that cannot be solved using the convolution tree) and pruning only the peptides that depend on them. This is an enhancement of the score-driven pruning algorithm defined in [13], which needed to prune *all* peptides at or below the score of the peptides responsible for computational expense.

In addition to easy compatibility with collapsed Gibbs sampling, the probabilistic convolution tree can be also be easily used with iterative approximation methods that pass messages in the original graph rather than in the junction tree: these methods include loopy belief propagation [20], variational methods [21], and expectation propagation [22]. Networks that employ a large

number of probabilistic adder nodes can pass these messages very rapidly and thus arrive at an approximation very quickly, even on graphs whose tree decompositions contain large cliques, which thus do not offer significant speedup compared to brute force. Importantly, the probabilistic convolution tree can efficiently pass messages forward and backward through nodes with many predecessors, enabling these iterative approximation procedures to not longer be limited by the maximum number of predecessors from any node.

Performance Improvement for Junction Tree Inference on a High-coverage HeLa Sample

As stated above, junction tree inference allows the belief propagation algorithm to be applied to generalized from trees to graphs by first performing a tree decomposition of a graph and then passing messages through the tree decomposition. Here we show the runtime speedup introduced by using probabilistic convolution trees within the message passing step of junction tree inference.

Figure 10 demonstrates the practical benefit of using a junction tree that utilizes probabilistic convolution trees to perform belief propagation. This is demonstrated using a high-coverage (24 fractions) HeLa SILAC data set [23]. Proteins were digested with trypsin and searched against the highly redundant International Protein Index (IPI) database (using a precursor mass tolerance of 10 ppm and a fragment mass tolerance of 0.6 Da). Identified spectra were then processed with Percolator using a maximum delta Cn of 0.05.

Protein identification was performed using FidoCT (Fido with convolution trees) in the alpha release of Proteome Discoverer 2.0. Low-scoring PSMs (those with score <1%) were pruned to reduce graph connectivity as described in [13]. The resulting graph was factorized into separate connected subgraphs, and each connected subgraph was processed using two variants of exact junction tree inference: first, the state-of-the-art HUGIN [24] junction tree algorithm and second a novel junction tree approach that passes messages by using probabilistic convolution trees. The HUGIN junction tree algorithm, as described for mass spectrometry-based proteomics in [8], performs tree decomposition by merging variables into cliques (illustrated in figure 1), and then performs message passing between these clique nodes. Note that the HUGIN junction tree inference cliques require time and space that grows super-exponentially with the size of (*i.e.* the number of variables contained in) the clique. Thus the size of the largest clique dominates the computational cost; however, when probabilistic convolution trees are used, cliques formed by probabilistic adders have sub-quadratic rather than super-exponential time.

Figure 10a shows the distribution of (log-scaled) runtimes for each connected subgraph when using a standard HUGIN algorithm versus using the probabilistic convolution tree-based approach. One particular connected subgraph requires $2^{103.629}$ steps using the HUGIN junction tree, but requires only $2^{12.045}$ steps when using probabilistic convolution trees within the junction tree inference algorithm. This connected subgraph is depicted in figure 10b; even though its treewidth is high, exact inference can be easily performed using probabilistic convolution trees.

Potential Impact of Convolution Trees on Protein Inference

In proteomics, the convolution tree could make it feasible to query protein databases with much greater sequence similarity than is currently possible, due to the large number of shared

dependencies introduced (as shown in figure 10). Moreover, the convolution tree could be used to iteratively perform protein inference and model peptide detectability, because it can offer substantially better runtimes on large or highly complex data sets; on such data sets, iterative numeric methods (e.g. ProteinProphet [25]) have been demonstrated to be unstable [2,13,26], heuristics and human intuition can break down [27], approximations (such as pruning peptides with nonzero scores) may be forced to yield inaccurate results [3], and sampling on its own (without exploitation of d -separation and mathematical properties like those introduced in this paper) cannot yield accurate probabilities in a feasible amount of time [2,3,8].

The convolution tree can also be used to efficiently place arbitrary categorical priors on the number of present variables or on the sum of variables. Without this advance, such priors would not be considered because they are too inefficient for large data sets: by creating a dependency between all proteins, such a prior would render factorization impossible. Without factorization, even a runtime quadratic in the number of variables (e.g. using the quadratic dynamic programming approach) could potentially become the factor limiting efficiency (not to mention the limitations of the quadratic space requirement). A sub-quadratic method with low space complexity could be used to bring the applicability of such priors to many graphical inference problems.

Application to Probabilistic Generating Functions for Partition Combinatorics and Linear Diophantine Equations

The convolution tree can also be trivially applied to classic partition problems from combinatorics [28]. Given some target value t and elements v_1, v_2, \dots , compute the total number of distinct integer tuples (N_1, N_2, \dots) that satisfy $N_1 v_1 + N_2 v_2 + \dots = t$. Traditionally, generating functions have been used to great success for such problems; however, converting the generating function solution to the desired result involves computing a particular polynomial coefficient from generating function, and can be time-consuming. Furthermore, the probabilistic convolution tree framework relaxes the strict equality required by generating function variants: the probabilistic convolution tree would not require the sum be *exactly* the target value. Instead, it would be compatible with an arbitrary likelihood function that weights values by the quality of the approximation between the target t and the sum $N_1 v_1 + N_2 v_2 + \dots$. In practice, this problem can be used to compute the total number of unique ways to make change for a given amount of money, and can also be applied to deciding if an observed target mass t can be made from specified set of elements (or, more generally, components) with respective masses v_1, v_2, \dots (and to compute the marginal distribution on the quantity of each element). A generalized two-dimensional variant could also constrain the total valence of the elements employed used.

Likewise, the probabilistic convolution tree can be applied to the highly related problem of linear Diophantine equations, which seek integers N_1, N_2, \dots such that $N_1 v_1 + N_2 v_2 + \dots = t$ for some target value t . The probabilistic convolution tree can decide if the equation is feasible (using integers N_1, N_2, \dots), as well as to compute marginal probabilities for each integer coefficient N_1, N_2, \dots . Although the problem of finding *whether* such an integer solution exists is known to be polynomial, the full problem of computing a joint solution in the integers (N_1, N_2, \dots) is NP-complete [29].

Generalization to Multiple Dimensions and Applicability to Knapsack Problems

The convolution tree trivially generalizes to multidimensional problems where there is an additive behavior in all dimensions. For instance, as mentioned above when mentioning the applicability to elemental decomposition, we can consider all variables $(X_1, X_2, \dots, X_k, N_1^{(1)}, N_2^{(1)}, \dots, N_1^{(2)}, \dots, N_1^{(3)} \dots)$ to have two dimensions. In the case of elemental decomposition, each variable can store a two-dimensional random distribution as a matrix where the row index maps to the discretized mass, the column maps to the discretized valence, and the value in the matrix cell at row r and column c is the joint probability that the variable simultaneously has the mass corresponding to r and the valence corresponding to c . Note that the algorithm is identical with two exceptions: First, when adding two parent variables L and R during a forward pass (step 1), the result will now be computed via a two-dimensional convolution. Second, after two-dimensional convolution is performed during the backward pass (in step 2), a contiguous submatrix is retrieved via a two-dimensional slice (rather than a one-dimensional slice).

As is the case for the univariate algorithm, the resulting convolutions can be performed using two alternative approaches: First, in the case when matrices are sparse, convolution can be performed efficiently using direct convolution when the joint distribution of L and the joint distribution of R . Alternatively, when the matrices representing these two-dimensional distributions are dense this two-dimensional convolution can be performed by using more sophisticated approaches [30]. The algorithm is the same in any number of dimensions as long as multidimensional convolution is used throughout, and as long multidimensional slices are taken in step 2. In this manner, it can be thought of as a polynomial time approximation scheme (PTAS) for a probabilistic generalization of the NP-hard knapsack problems. Likewise, it can efficiently solve other combinatorics problems (where counting is used probabilistically or non-probabilistically). The multidimensional variant can also be used to merge variables and remove loops from cascaded graphs that are not trees.

Application to using Truncated Sums in Forward Error Correction

Because the convolution tree method can be used with any cascaded graph of probabilistic adder nodes, then it can be applied to myriad other problems. One simple example is the extension of probabilistic forward error correcting codes to efficiently utilize a greater variety of error correcting information. Using the notation from [31] the unique data D_i would correspond to y_i , and a subset of those probabilistic adder nodes in the tree could influence node-specific $D_j^{(i)}$, which would allow inclusion of data that depends on the sum of nodes above, or optionally, a truncated sum, from which a probabilistic adder node can compute multi-bit summary statistics about the bit string (parity is an example of such a single-bit summary statistic). For certain coding schemes that use moderate block sizes, this method could be used to infer an optimal (i.e. *maximum a posteriori*) estimate for the input binary bitstring, as well as probabilistic confidence estimates for each bit in the bitstring. Turbo decoding and low-density parity-check codes (LDPC), which are both popular inference methods for forward error correction, have been shown to be instances of Pearl's loopy belief propagation [31,32] (loopy belief propagation is described above). Because convolution trees can be used within belief propagation (see "Compatibility with belief propagation and junction tree inference" above), they can also be used with loopy

belief propagation to perform more efficient message passing of probabilistic information related to cardinality, parity, sums, and other probabilistic adder structures that cannot be efficiently accomplished with noisy-or nodes. And because the operations performed in the probabilistic convolution tree are basic digital signal processing operations (FFT, element-wise product, etc.), and thus could potentially be implemented efficiently as an integrated circuit.

Application to Probabilistic Demixing of Chimeric Mass Spectra

The convolution tree can also be applied to demixing problems. Figure 11 depicts a classic example from mass spectrometry: four compounds of unknown relative abundance contribute to a chimeric spectrum. By discretizing the possible relative abundances (in the future it also may be possible to extend some of the ideas presented in this manuscript to continuous problems), probabilistic adder nodes can be cascaded to compute posterior probability distributions on the relative abundances of each compound without jointly enumerating the four-dimensional space of all possible relative abundances. This graph could even be augmented with an arbitrary prior on the number of compounds present. A simplification would threshold peaks in the chimeric spectrum into two categories (“intense” and “not intense”) and then perform inference using a convolution tree whose base variables are binary,

similar to the formulation for protein inference. Regardless of whether binary variables or binned continuous variables are used, an arbitrary likelihood model could then be used to evaluate the match between the observed peak (observed from the actual data) and the latent abundance variable for that peak (note that these likelihood functions can be peak-specific).

Availability

The python script demonstrating the power-set enumeration, quadratic dynamic programming, and convolution tree approaches is available at <https://bitbucket.org/orserang/convolutiontree>.

Acknowledgments

This research was a product of discussions with Frank Ciesinski, Kai Fritzsche, Christoph Henrich, Alexander Makarov, Robert Malek, and Torsten Ueckert regarding efficient protein inference via mass spectrometry. Thanks to Ute Comberg for searching the spectra and pre-processing the HeLa data, and to the reviewers, whose suggestions helped to improve this manuscript.

Author Contributions

Conceived and designed the experiments: OS. Performed the experiments: OS. Analyzed the data: OS. Contributed reagents/materials/analysis tools: OS. Wrote the paper: OS.

References

- Nesvizhskii AI (2010) A survey of computational methods and error rate estimation procedures for peptide and protein identification in shotgun proteomics. *Journal of Proteomics* 73: 2092–2123.
- Serang O, Noble WS (2012) A review of statistical methods for protein identification using tandem mass spectrometry. *Statistics and Its Interface* 5: 3–20.
- Li YF, Radivojac P (2012) Computational approaches to protein inference in shotgun proteomics. *BMC bioinformatics* 13: S4.
- Huang T, Wang J, Yu W, He Z (2012) Protein inference: a review. *Briefings in bioinformatics* 13: 586–614.
- Kelchtermans P, Bitremieux W, De Grave K, Degroev S, Ramon J, et al. (2013) Machine learning applications in proteomics research: How the past can boost the future. *Proteomics*.
- Koller D, Friedman N (2009) *Probabilistic graphical models: principles and techniques*. MIT Press.
- Robertson N, Seymour PD (1984) Graph minors. III. Planar tree-width. *Journal of Combinatorial Theory, Series B* 36: 49–64.
- Serang O, Noble WS (2012) Faster mass spectrometry-based protein inference: Junction trees are more efficient than sampling and marginalization by enumeration. *Computational Biology and Bioinformatics, IEEE/ACM Transactions on* 9: 809–817.
- Li YF, Arnold RJ, Li Y, Radivojac P, Sheng Q, et al. (2008) A Bayesian approach to protein inference problem in shotgun proteomics. In: Vingron M, Wong L, editors, *Proceedings of the Twelfth Annual International Conference on Computational Molecular Biology*. Berlin, Germany: Springer, volume 12 of *Lecture Notes in Bioinformatics*, pp. 167–180.
- Li YF, Arnold RJ, Radivojac P, Tang H (2012) Protein identification problem from a Bayesian point of view. *Statistics and Its Interface* 5: 21–37.
- Rappsilber J, Mann M (2002) What does it mean to identify a protein in proteomics? *Trends in biochemical sciences* 27: 74–78.
- McDowell G, Gaun A, Steen H (2013) iFASP: Combining isobaric mass tagging with filter-aided sample preparation. *Journal of Proteome Research* 12: 3809–3812.
- Serang O, MacCoss MJ, Noble WS (2010) Efficient marginalization to compute protein posterior probabilities from shotgun mass spectrometry data. *Journal of Proteome Research* 9: 5346–5357.
- Pearl J (1982) Reverend Bayes on inference engines: A distributed hierarchical approach. In: *Proceedings of the Second National Conference on Artificial Intelligence*. pp. 133–136.
- Heckerman D (1993) Causal independence for knowledge acquisition and inference. In: *Proceedings of the Ninth international conference on Uncertainty in artificial intelligence*. pp. 122–127.
- Heckerman D, Breese J (1996) Causal independence for probability assessment and inference using Bayesian networks. *Systems, Man and Cybernetics, Part A: Systems and Humans, IEEE Transactions on* 26: 826–831.
- Heckerman D, Breese J (1994) A new look at causal independence. In: *Proceedings of the Tenth international conference on Uncertainty in artificial intelligence*. pp. 286–292.
- Baum L, Petrie T (1966) Statistical inference for probabilistic functions of finite state Markov chains. *The annals of mathematical statistics* 37: 1554–1563.
- Zhang N, Poole D (1996) Exploiting Causal Independence in Bayesian Network Inference. *Journal of Artificial Intelligence Research* 5: 301–328.
- Murphy K, Weiss Y, Jordan M (1999) Loopy belief propagation for approximate inference: An empirical study. In: *Proceedings of the Fifteenth conference on Uncertainty in artificial intelligence*. pp. 4067–475.
- Jordan MI, Ghahramani Z, Jaakkola TS, Saul LK (1999) Introduction to variational methods for graphical models. *Machine Learning* 37: 183–233.
- Minka T (2001) Expectation propagation for approximate Bayesian inference. In: *Proceedings of the Seventeenth conference on Uncertainty in artificial intelligence*. pp. 362–369.
- Cox J, Mann M (2008) MaxQuant enables high peptide identification rates, individualized p-value range mass accuracies and proteome-wide protein quantification. *Nature Biotechnology* 26: 1367–1372.
- Andersen SK, Olesen KG, Jensen FV (1990) HUGIN, a shell for building Bayesian belief universes for expert systems, San Francisco, CA, USA: Morgan Kaufmann Publishers Inc. pp. 332–337.
- Nesvizhskii AI, Keller A, Kolker E, Aebersold R (2003) A statistical model for identifying proteins by tandem mass spectrometry. *Analytical Chemistry* 75: 4646–4658.
- Reiter L, Claassen M, Schrimpf SP, Jovanovic M, Schmidt A, et al. (2009) Protein identification false discovery rates for very large proteomics data sets generated by tandem mass spectrometry. *Molecular and Cellular Proteomics* 8: 2405–2417.
- Serang O, Moruz L, Hoopmann MR, Käll L (2012) Recognizing uncertainty increases robustness and reproducibility of mass spectrometry-based protein inferences. *Journal of Proteome Research* 11: 5586–5591.
- Andrews G (1998) *The theory of partitions*, volume 2. Cambridge University Press.
- Pottier L (1991) Minimal solutions of linear diophantine systems: bounds and algorithms. In: *Rewriting Techniques and Applications*. pp. 162–173.
- Nussbaumer H, Quandalle P (1978) Computation of convolutions and discrete Fourier transforms by polynomial transforms. *IBM Journal of Research and Development* 22: 134–144.
- McEliece R, MacKay D, Cheng J (1998) Turbo decoding as an instance of Pearl’s belief propagation algorithm. *Selected Areas in Communications, IEEE Journal on* 16: 140–152.
- Shokrollahi A (2004) LDPC codes: An introduction. In: *Coding, cryptography and combinatorics*, Springer. pp. 85–110.