Expression Graphs
Unifying Factor Graphs and Sum-Product Networks

Abram Demski (ademski@ict.usc.edu)*

Institute for Creative Technologies and Department of Computer Science
University of Southern California
12015 Waterfront Dr., Playa Vista, CA 90094 USA

Abstract. Factor graphs are a very general knowledge representation, subsuming many existing formalisms in AI. Sum-product networks are a more recent representation, inspired by studying cases where factor graphs are tractable. Factor graphs emphasize expressive power, while sum-product networks restrict expressiveness to get strong guarantees on speed of inference. A sum-product network is not simply a restricted factor graph, however. Although the inference algorithms for the two structures are very similar, translating a sum-product network into factor graph representation can result in an exponential slowdown. We propose a formalism which generalizes factor graphs and sum-product networks, such that inference is fast in cases whose structure is close to a sum-product network.

1 Motivation

Factor graphs are a graphical model which generalizes Bayesian networks, Markov networks, constraint networks, and other models [4]. New light was shed on existing algorithms through this generalization.[1]

As a result, factor graphs have been treated as a unifying theory for graphical models. It has furthermore been proposed, in particular in [2] and [11], that factor graphs can provide a computational foundation through which we can understand cognitive processes. The present work came out of thinking about potential inadequacies in the Sigma cognitive architecture [11].

Factor graphs have emerged from progressive generalization of techniques which were initially narrow AI. Because they capture a breadth of knowledge about efficient AI algorithms, they may be useful for those AGI approaches which

* This work was sponsored by the U.S. Army. Statements and opinions expressed may not reflect the position or policy of the United States Government, and no official endorsement should be inferred. Special thanks to Paul Rosenbloom and Łukasz Stafiniak for providing comments on a draft of this paper.

[1] The sum-product algorithm for factor graphs provided a generalization of existing algorithms for more narrow domains, often the best algorithms for those domains at the time. The main examples are belief propagation, constraint propagation, and turbo codes [4]. Other algorithms such as mean-field can be stated very generally using factor graphs as well.

seek to leverage the progress which has been made in narrow AI, rather than striking out on an entirely new path. However, this paper will argue that factor graphs fail to support an important class of algorithms.

Sum-product networks (SPNs) are a new type of graphical model which represent a probability distribution through sums and products of simpler distributions [7].[2] Whereas factor graphs may blow up to exponential-time exact inference, SPN inference is guaranteed to be linear in the size of the SPN.

An SPN can compactly represent any factor graph for which exact inference is tractable. When inference is less efficient, the corresponding SPN will be larger. In the worst case, an SPN may be exponentially larger than the factor graph which it represents. On the other hand, being able to represent a distribution as a compact SPN does *not* imply easy inference when converted to a factor graph. There exist SPNs which represent distributions for which standard exact inference algorithms for factor graphs are intractable.

Probabilistic context-free grammars (PCFGs) are an important class of probabilistic model in computational linguistics. In [8], the translation of PCFGs into factor graphs (specifically, into Bayesian networks) is given. This allows general probabilistic inference on PCFGs (supporting complicated queries which special-case PCFG algorithms don't handle). However, the computational complexity becomes exponential due to the basic complexity of factor graph inference.

Sum-product networks can represent PCFGs with bounded sentence length to be represented in an SPN of size cubic in the length, by directly encoding the sums and products of the *inside* algorithm (a basic algorithm for PCFGs) This preserves cubic complexity of inference, while allowing the more general kinds of queries for which [8] required exponential-time algorithms. This illustrates that SPNs can be efficient in cases where factor graphs are not.

More recently, [12] used loopy belief propagation (an *approximate* algorithm for factor graph problems) to efficiently approximate complex parsing tasks beyond PCFGs, but did so by implementing a dynamic programming parse *as* one of the factors. This amounts to using SPN-style inference as a special module to augment factor-graph inference.

The present work explores a more unified approach, to integrate the two types of reasoning without special-case optimization. The resulting representation is related to the *expression tree* introduced in [4]. As such, the new formalism is being referred to as the Expression Graph (EG).

[2] Case-factor diagrams [6] are almost exactly the same as sum-product networks, and have historical precedence. However, the formalism of sum-product networks has become more common. Despite their similarities, the two papers [6] and [7] use very different mathematical setups to justify the new graphical model and the associated inference algorithm. (A reader confused by one paper may benefit from trying the other instead.)

2 Factor Graphs

A *factor graph* (FG) is a bipartite graph where one set of nodes represents the variables $x_1, x_2, ...x_n \in \mathbf{U}$, and the other set of nodes represent real-valued multivariate functions $F_1, F_2, ...F_m$. A link exists between a factor node and a variable node when the variable is an argument to the factor. This represents a function D, the product of all the factors:

$$D(\mathbf{U}) = \prod_{i=1}^{m} F_i(\mathcal{A}_i)$$

where $\mathcal{A}_i$ represents the tuple of argument variables associated with factor F_i.

The global function D can represent anything, but we will only discuss the representation of probability functions in this article.

Representing the factorization explicitly allows factor graphs to easily capture the distributions represented by other graphical models like Bayesian networks and Markov networks whose graph structure implies a factorization. The links correspond conveniently with messages in several message-passing algorithms, most famously the sum-product algorithm for factor graphs, which generalizes several important algorithms.

Inference algorithms allow us to compute various things with these networks, most notably marginal probabilities and maximum-probability states. Exact inference using the most common algorithms is exponential in the treewidth, which is (roughly) a measure of how far the graph is from being tree-structured. As a result, nontrivial models usually must rely on approximate inference techniques, of which there are many.

Building up complicated functions as a product of simpler ones turns out to be very powerful. Intuitively, we can think of the factors as giving us probabilistic *constraints* linking variables together. (In fact, this is a strict generalization of constraint-based reasoning.) These constraints can provide a great deal of representational power, but this power comes at the cost of potentially intractable inference.

For further details, the reader is directed to [4].

3 Sum-Product Networks

A *sum-product network* (SPN) is a directed acyclic graph with a unique root. Terminal nodes are associated with *indicator variables*. Each domain variable in $\mathbf{U}$ has an indicator for each of its values; these take value 1 when the variable takes on that value, and 0 when the variable is in a different value.

The root and the internal nodes are all labeled as *sum nodes* or *product nodes*. A product node represents the product of its children. The links from a sum node to its children are weighted, so that the sum node represents a weighted sum of its children. Thus, the SPN represents an expression formed out of the indicator variables via products and weighted sums. As for factor graphs, this expression

could represent a variety of things, but in order to build an intuition for the structure we shall assume that this represents a probability distribution over U.

The *scope* of a node is the set of variables appearing under it. That is: the scope of a leaf node is the variable associated with the indicator variable, and the scope of any other node is the union of the scopes of its children.

Two restrictions are imposed on the tree structure of an SPN. It must be *complete*: the children of any particular sum node all have the same scope as each other. They must also be *decomposable*: the children of the same product node have mutually exclusive scopes. These properties allow us to compute any desired probability in linear time. It's possible to compute *all* the marginal probabilities in linear time by differentiating the network, an approach adapted from [1].[3]

For further details, the reader is directed to [7].

If we think of factor graphs as generalized constraint networks, we could think of SPNs as generalized decision trees – or, for a closer analogy, binary decision diagrams [3]. These represent complexity by splitting things into cases, in a way which can be evaluated in one pass rather than requiring back-tracking search as with constraint problems.

The thrust of this paper is that both kinds of representation are necessary for general cognition. To accomplish this, we generalize SPNs to also handle constraint-like factor-graph reasoning.

4 Expression Graphs

In order to compactly represent all the distributions which can be represented by SPNs or FGs, we introduce the *expression graph* (EG).

An expression graph is little more than an SPN with the network restrictions lifted: a directed acyclic graph with a unique root, whose non-terminal nodes are labeled as sums or products. The terminal nodes will hold functions rather than indicators; this is a mild generalization for convenience. For discrete variables, these functions would be represented as tables of values. For the continuous case, some class of functions such as Gaussians would be chosen. These terminal functions will be referred to as *elemental functions*. We will only explicitly work with the discrete case here. Expression graphs represent complicated functions build up from the simple ones, as follows:

$$N(\mathbf{A}) = \begin{cases} \sum_{i=1}^{n} C_i(\mathbf{A_i}) \text{ if } N \text{ is a sum node} \\ \prod_{i=1}^{n} C_i(\mathbf{A_i}) \text{ if } N \text{ is a product node} \end{cases}$$

Where C_i are the n children of node N, and $\mathbf{A}$ is the union of their arguments $\mathbf{A}_i$. From now on, we will not distinguish strongly between a node and the function associated with the node. The root node is D, the global distribution.

[3] In [7], a weaker requirement of *consistency* replaces decomposability. However, without full decomposability, the inference by differentiation can give wrong results. For example, the SPN representing $.5x_1^2 + .5\overline{x_1}$ is acceptable by their definition. Differentiation would have it that $x_1 = true$ is twice as likely as $x_1 = false$, whereas the two are equally likely by evaluation of the network value at each instantiation. We therefore do not consider the weaker requirement here.

The scope of a node is defined as the set arguments in its associated function, inheriting the definitions of complete and decomposable which were introduced for SPNs. Unlike in the case of SPNs, we do not enforce these properties.

A simple model expressing $P(x, y)$ as the product of two functions $F(x)G(x, y)$ (for example,) becomes the network in Figure 1. This gives us a small example of the sorts of expressions which are tractable enough to be useful, but, are not allowed by the restrictions on SPN structure. (The expression is not decomposable, and would need to be re-written as a sum of all the possible cases to be an SPN, obscuring the factorization.)

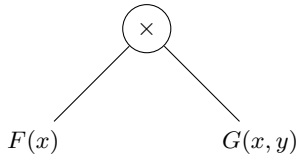

Fig. 1. The expression graph for $F(x)G(x, y)$

On the other hand, a mixture distribution on x defined by the expression $.4f(x) + .6g(x)$ looks like Figure 2. Unlike for SPNs, we represent the weights as terminal nodes (these can be thought of as constant functions, with empty scope). This difference simplifies the form of later formulas. The weighted-edge formalism could be used instead with a little modification to the algorithms.

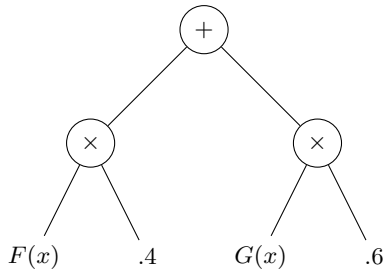

Fig. 2. The mixture distribution $.4F(x) + .6G(x)$

Several models existing in the literature can be interpreted as expression graphs, putting them in a more unified formalism. One way of combining factor graphs and SPNs is to use SPNs as compact representations *of* the local factors, as was done in [9]. This allows use of the expressive power of factor graphs at the higher level, while taking advantage of the efficiency of SPNs to represent the local interactions. The paper notes that this allows a single framework for

inference, and constructs SPNs in terms of context-sensitive factors in CRFs (a type of factor graph).

The reverse case can be observed in [10], which uses sum-product networks with factor graphs as leaves in order to represent indirect variable interactions while ensuring that the overarching structure is tractable. This is used as part of an SPN structure-learning algorithm, which takes advantage of existing factor-graph structure learning to capture certain types of variable interactions which took too long to discover in a previous (pure SPN) structure learner.

5 Exact Inference

The goal of inference is to compute the marginal probability distribution of one or more variables, typically given some evidence. Evidence restricts the value of some variables. We can handle this by modifying the definition of the domain variables to restrict their possible values. It therefore suffices to consider the inference problem when no evidence is present.

The marginal for a variable x is defined as follows:

$$\sum_{\mathbf{U}-\{x\}} D(\mathbf{U})$$

Unfortunately, this computation is exponential time in the number of variables. We would like to re-arrange the summation to simplify it.

To deal with a slightly more general case, let's assume that we have a set of variables $\mathbf{X}$ we want to find the joint marginal for.

Supposing that we had *completeness*, we could push down the summation through sum nodes N:

$$\sum_{\mathbf{A}-\mathbf{X}} N(\mathbf{A}) = \sum_{i=1}^{n} \sum_{\mathbf{A}-\mathbf{X}} C_i(\mathbf{A})$$

Here, $\mathbf{A}$ is the scope of the parent, which (by completeness) is also the scope of each child. (As in the previous section, C_i will represent the children of N.)

Similarly, if we had *decomposability*, we could push down the sums through product nodes:

$$\sum_{\mathbf{A}-\mathbf{X}} N(\mathbf{A}) = \prod_{i=1}^{n} \sum_{\mathbf{A}_i-\mathbf{X}} C_i(\mathbf{A}_i)$$

Here, $\mathbf{A}$ is the scope of the parent, and the $\mathbf{A}_i$ are the scopes of the children. By decomposability, the $\mathbf{A}_i$ must be mutually exclusive, so that we can apply the distributive rule to push the sum down through the product. This reduces the complexity of the computation by allowing us to sum over the sets of variables $\mathbf{A}_i$ separately, and then combine the results.

Since we do *not* in general have a graph which is complete and decomposable, we need to adjust for that. The adjustment at sum nodes is computationally easy,

augmenting the values from children with a multiplier to account for summing out the wider scope required by the parent:

$$\sum_{\mathbf{A}-\mathbf{X}} N(\mathbf{A}) = \sum_{i=1}^{n} \sum_{\mathbf{A}-\mathbf{A}_i-\mathbf{X}} \sum_{\mathbf{A}_i-\mathbf{X}} C_i(\mathbf{A}_i) \tag{1}$$
$$= \sum_{i=1}^{n} \left(\prod_{y \in \mathbf{A}-\mathbf{A}_i-\mathbf{X}} |\mathcal{V}(y)| \right) \sum_{\mathbf{A}_i-\{x\}} C_i(\mathbf{A}_i)$$

Where $\mathcal{V}(y)$ is the set of valid values for variable y.

The adjustment for non-decomposable products is less computationally convenient:

$$\sum_{\mathbf{A}-\mathbf{X}} N(\mathbf{A}) = \sum_{\mathbf{B}-\mathbf{X}} \prod_{i=1}^{n} \sum_{\mathbf{A}_i-\mathbf{B}-\mathbf{X}} C_i(\mathbf{A}_i) \tag{2}$$

Where $\mathbf{B}$ is the set of variables that appear in more than one of $\mathbf{A}_i$. (Note that we do not have to worry that some variables might appear in no $\mathbf{A}_i$, because the scope of the parent was defined as the union of the scopes of the children.)

What this equation says is just that we cannot push down the summation over a particular variable if that variable is shared between several children of a product node. This fails to satisfy the conditions for the distributive law. As a result, we have to sum this variable out *at* the offending product node.

Applying these equations recursively, we can create a dynamic-programming style algorithm which computes the desired marginal. This proceeds first in a downward pass, in which we mark which variables we need to avoid summing out at which nodes. Then, we pass messages up through the network. The messages are multidimensional arrays, giving a value for each combination of marked variables.

Algorithm 1 *To find* $\sum_{\mathbf{U}-\mathbf{X}} D(\mathbf{U})$*:*

1. *Mark variables* $\mathbf{X}$ *at the root.*
2. *For non-decomposable products, mark shared variables in the product node.*
3. *Propagate marks downward, marking a variable in a child whenever it is marked in the parent and occurs in the scope of the child.*
4. *Set the messages* $\mathcal{M}(N)$ *where* N *is a terminal node to be* $\sum_{\mathbf{H}} N(\mathbf{A})$*, where* $\mathbf{A}$ *are the arguments of the function* N *and* $\mathbf{H}$ *are any unmarked arguments.*
5. *Propagate up messages* $\mathcal{M}(N) =$

$$\begin{cases} \sum_{i=0}^{n} \pi_i \sum_{\mathbf{H}_i} \mathcal{M}(C_i) & \text{if } N \text{ is a sum node} \\ \sum_{\mathbf{H}} \prod_{i=0}^{n} \mathcal{M}(C_i) & \text{if } N \text{ is a product node} \end{cases}$$

where C_i *are the children of node* N*,* $\mathbf{H}_i$ *is the set of dimensions marked in* C_i *but not marked in* N*,* $\mathbf{H}$ *is the union of the* $\mathbf{H}_i$*, and* π_i *is the multiplier from Equation 1 adjusted to remove marked variables:* $\prod_{x \in \mathbf{A}-\mathbf{A}_i-\mathbf{X}-\mathbf{M}} |\mathcal{V}(x)|$ *with* $\mathbf{A}$ *as the arguments of* N*,* $\mathbf{A}_i$ *those of* C_i*, and* $\mathbf{M}$ *the marked variables of* N*.*

This algorithm bears a resemblance to the "expression trees" mentioned in [4]. The variable marking procedure also brings to mind "variable stretching" from that paper: we are marking out a portion of the graph in which we need to keep track of a variable in order to enable local message-passing computation.

With only about twice as much work, we can compute *all* the single-variable marginals by adding a second set of messages. This should not be a surprise, since the same is true of both SPNs and factor graphs. The solution closely resembles the inside-outside algorithm for PCFGs, with the messages from the previous section constituting the "inside" part.

In order to compute the marginal probabilities, we must first compute a set of partial derivatives in an arithmetic circuit (AC) representing the distribution. (The reader is directed to [1] for the details of this approach.)

We will re-name the messages $\mathcal{M}$ from Algorithm 1 to "upward messages" $\mathcal{M}_u$, with new "downward messages" $\mathcal{M}_d$.

Algorithm 2 *To find all single-variable marginals of D:*

1. *Run Algorithm 1, with $\mathbf{X} = \emptyset$. Keep the messages as $\mathcal{M}_u(N)$.*
2. *Set the downward message for the root node $\mathcal{M}_d(D) = 1$.*
3. *Compute downward messages $\mathcal{M}_d(N) = \sum_{i=0}^{n} \mathcal{C}(P_i, N)$, where P_i are the n parents of N and we define the contribution for each parent $\mathcal{C}(P_i, N) =$*

$$\begin{cases} \pi_i \sum_{\mathbf{H}} \mathcal{M}_d(P_i) & \text{if } P_i \text{ is a sum} \\ \sum_{\mathbf{H}} \mathcal{M}_d(P_i) \prod_{j=0}^{m} \mathcal{M}_u(C_j) & \text{if } P_i \text{ is a product} \end{cases}$$

   where π_i is the same multiplier between parent and child as in the upward messages, C_j are the m other *children of parent P_i, and $\mathbf{H}$ is the set of variables marked in P_i and not in N.*
4. *For each variable $v \in \mathbf{U}$, compute the marginal as the sum of partial derivatives for terminal nodes, and partial derivatives coming from π-adjustments involving that variable:*

$$\begin{aligned} \mathcal{M}_d(v) = &\sum_{i=1}^{n} \sum_{\mathbf{H}_i} F_i \mathcal{M}_d(F_i) \\ &+ \sum_{(P_i, C_j)} \mathcal{M}_d(P_i) \end{aligned}$$

   where the F_i are the terminal nodes, $\mathbf{H}_i$ are the arguments of F_i other than v, $\sum_{(P_i,C_j)}$ is summing over parent-child pairs (P_i, C_j) such that P_i has v in scope and not marked but C_j does not (so that π-adjustments would appear in the messages).

The intuition is that upward messages compute the total value of the corresponding AC, whereas downward messages compute the partial derivative of the total value with respect to individual AC nodes. Each scalar value in the multidimensional message corresponds to an AC node.

This computes the same quantities which we would get by compiling to an AC and differentiating. The technique rolls together the compilation to an AC with the inference in the AC, so that if we apply it to an EG representing a factor

graph, we are doing something very similar to compiling it to an AC and then differentiating (one of the better choices for exact inference in factor graphs). Since the algorithm reduces to SPN inference in the special case that the EG is indeed an SPN, we have the SPN efficiency in that case. In particular, we can get cubic complexity in the parsing problem which was mentioned as motivation.

Because expression graphs also admit the intractable cases which factor graphs allow, it will be desirable to have approximate inference algorithms such as Monte Carlo and variational methods. Variational methods would focus on the approximation of large multidimensional messages by approximate factorization. Monte Carlo would focus on approximating the large summations by sampling. A deep exploration of these possibilities will have to be left for another paper.

As a result of taking the derivatives of the network, this algorithm also gives us the derivatives needed to train the network by gradient-descent learning. However, we won't discuss this in detail due to space limitations.

6 Future Work

The concrete algorithms here have dealt with finite, fixed-size expression graphs, but the motivation section mentioned representation of grammars, which handle sequential information of varying size. Work is in progress applying expression graphs to grammar learning, enabling an expressive class of grammars.

Unlike factor graphs, expression graphs and SPNs can represent structural uncertainty within one graph, by taking a sum of multiple possible structures. Theoretically, structure learning and weight learning can be reduced to one problem. Of course, a graph representing the structure learning problem is too large for practical inference. In [5], infinite SPNs are defined via Dirichlet distributions, and sampling is used to make them tractable. Perhaps future work could define similar infinite EGs to subsume structure learning into inference.

The structure-learning algorithm in [10] is also quite interesting, employing heuristics to split the data into cases or factor the data, alternatively. This could point to two different sets of cognitive mechanisms, dealing independently with sums and products. Sum-like mechanisms include clustering, boosting, and bagging. These deal with complexity by making mixture models. Product-like mechanisms deal with complexity by splitting up the variables involved into sub-problems which may be independent or related by constraints (that is, factoring!). Perhaps distinct psychological processes deal with these two options. In future work, we hope to use this distinction in a cognitive architecture context.

7 Conclusion

It is hoped that this representation will help shed light on things from a theoretical perspective, and also perhaps aid in practical implementation in cases where a mixture of factor-graph style and SPN-style reasoning is required. Expression graphs are a relatively simple extension: from the perspective of a factor graph, we are merely adding the ability to take sums of distributions rather than

only products. From the perspective of SPNs, all we are doing is dropping the constraints on network structure. This simple move nonetheless provides a rich representation.

This formalism helps to illustrate the relationship between factor graphs and sum-product networks, which can be somewhat confusing at first, as sum-product networks are described in terms of indicator variables and representing the network polynomial, concepts which may seem alien to factor graph representations.

Expression graphs improve upon factor graphs in two respects. First, it is a more expressive representation than factor graphs as measured in the kinds of distributions which can be represented compactly. Second, the representation is more amenable to exact inference in some cases, where generic factor graph inference algorithms have suboptimal complexity and must be augmented by special-case optimization to achieve good performance.

References

1. Darwiche, A.: A differential approach to inference in bayesian networks. Journal of the ACM (2003)
2. Derbinsky, N., Bento, J., Yedidia, J.: Methods for integrating knowledge with the three-weight optimization algorithm for hybrid cognitive processing. In: AAAI Fall Symposium on Integrated Cognition (2013)
3. Drechsler, R., Becker, B.: Binary Decision Diagrams: Theory and Implementation. Springer Verlang (1998)
4. Kschischang, F., Frey, B., Loeliger, H.: Factor graphs and the sum-product algorithm. In: IEEE Transactions on Information Theory (2001)
5. Lee, S.W., Watkins, C., Zhang, B.T.: Non-parametric bayesian sum-product network. In: Proc. Workshop on Learning Tractable Probabilistic Models. vol. 1 (2014)
6. McAllester, D., Collins, M., Pereira, F.: Case-factor diagrams for structured probabilistic modeling. In: Proc. UAI-04 (2004)
7. Poon, H., Domingos, P.: Sum-product networks: A new deep architecture. In: Proc. UAI-11 (2011)
8. Pynadath, D., Wellman, M.: Generalized queries on probabilistic context-free grammars. In: Pattern Analysis and Machine Intelligence, IEEE Transactions on, vol. 20, pp. 65–77
9. Ratajczak, M., Tschiatschek, S., Pernkopf, F.: Sum-product networks for structured prediction: Context-specific deep conditional random fields. In: Proc. Workshop on Learning Tractable Probabilistic Models. vol. 1 (2014)
10. Rooshenas, A., Lowd, D.: Learning sum-product networks with direct and indirect variable interactions. In: Proc. Workshop on Learning Tractable Probabilistic Models. vol. 1 (2014)
11. Rosenbloom, P.: The sigma cognitive architecture and system. AISB Quarterly (2013)
12. Smith, D.A., Eisner, J.: Dependency parsing by belief propagation. In: Proceedings of the Conference on Empirical Methods in Natural Language Processing. Association for Computational Linguistics (2008)