

Products of Weighted Logic Programs

Shay B. Cohen, Robert J. Simmons and Noah A. Smith

CMU-LTI-08-009

Language Technologies Institute
School of Computer Science
Carnegie Mellon University
5000 Forbes Ave., Pittsburgh, PA 15213
www.lti.cs.cmu.edu

© 2008, Shay B. Cohen and Robert J. Simmons and Noah A. Smith

Abstract. Weighted logic programming, a generalization of bottom-up logic programming, is a successful framework for specifying dynamic programming algorithms. In this setting, proofs correspond to the algorithm's output space, such as a path through a graph or a grammatical derivation, and are given a weighted score, often interpreted as a probability, that depends on the score of the base axioms used in the proof. The desired output is a function over all possible proofs, such as a sum of scores or an optimal score. We describe the **PRODUCT** transformation, which can merge two weighted logic programs into a new one. The resulting program optimizes a product of proof scores from the original programs, constituting a scoring function known in machine learning as a "product of experts." Through the addition of intuitive constraining side conditions, we show that several important dynamic programming algorithms can be derived by applying **PRODUCT** to weighted logic programs corresponding to *simpler* weighted logic programs. This report is an extended version of [3].

1 Introduction

Weighted logic programming has found a number of applications in fields such as natural language processing, machine learning, and computational biology as a technique for declaratively specifying dynamic programming algorithms. Weighted logic programming is a generalization of bottom-up logic programming, with the numerical scores for proofs often interpreted as probabilities, implying that the weighted logic program implements *probabilistic* reasoning.

We describe a program transformation, **PRODUCT**, that is of special interest in weighted logic programming. **PRODUCT** transforms two weighted logic programs into a new one that implements probabilistic inference under an unnormalized probability distribution built as a product of the input programs' distributions, known in machine learning as a "product of experts." While this property has been exploited in a variety of ways in applications, there has not, to our knowledge, been a formal analysis or generalization in terms of the weighted logic programming representation.

The contribution of this paper is a general, intuitive, formal setting for dynamic programming algorithms that process two or more conceptually distinct structured inputs. Indeed, we show that many important dynamic programming algorithms can be derived using simpler "factor" programs and the **PRODUCT** transformation.

The paper is organized as follows. In §2 we give an overview of weighted logic programming. In §3 we describe products of experts, a concept from machine learning that motivates our framework. In §4 we describe our framework and its connection to product of experts. In §5 we give derivations of several well-known algorithms using our framework.

2 Weighted Logic Programming

To motivate weighted logic programming, we begin with a logic program for single-source connectivity on a directed graph, shown in Fig. 1. In the usual bottom-up interpretation of this program, an initial database would describe the edge relation and one (or more) roots as axioms of the form `initial(a)` for some `a`, and repeated forward inference would be applied on the two rules above to find the least database closed under those rules. However, in traditional logic programming this program can *only* be understood as a program calculating connectivity over a graph. Solving a different but structurally similar problem, such as a single-source shortest path, requires a rather different program to be written, and most solutions that have been presented require some form of non-deterministic committed choice [13, 11].

Traditional logic programming is interpreted over Boolean values. A proof is a tree of valid inferences, and a valid proof is one where all of the leaves of the proof tree are axioms which are known to be true, and a true atomic proposition is one that has at least one valid proof. In weighted logic programming we generalize this notion: axioms, proofs, and atomic propositions are talked about as "having values" rather than just "being true/valid." A Boolean logic program takes the value of a proof to be the *conjunction* of the value of its axioms (so the proof has value "true" as long as all the propositions at the leaves are true-valued axioms), and takes the value of a proposition to be the *disjunction* of the values of its proofs (so an atomic proposition has value "true" if it has one true-valued proof). The single-source connectivity program would describe the graph in Fig. 2 by assigning `T` as the value of all the existing edges and the proposition `initial(a)`.

2.1 Non-Boolean Programs

With Weighted logic programming, the axioms and propositions can be understood as having non-Boolean values. In Fig. 3, axioms of the form `edge(X,Y)` are given a value corresponding to the cost along the edge in the graph, and the axiom `initial(a)` is given the value 0. If we take the value or "score" of a proof to be the *sum* of the values of its axioms, and then take the value of a proposition to be the *minimum* score over all possible proofs, then the program from Fig. 1 describes *single-source shortest path*. We replace the operators `:-` (disjunction) and `,` (conjunction) with `min =` and `+`, respectively, and interpret the program over the non-negative numbers. With a specific execution strategy, the result is Dijkstra's single-source shortest path algorithm.

$$\text{reachable}(Q) \text{ :- } \text{initial}(Q). \tag{1}$$

$$\text{reachable}(Q) \text{ :- } \text{reachable}(P), \text{edge}(P, Q). \tag{2}$$

Fig. 1: A simple bottom-up logic program for graph reachability.

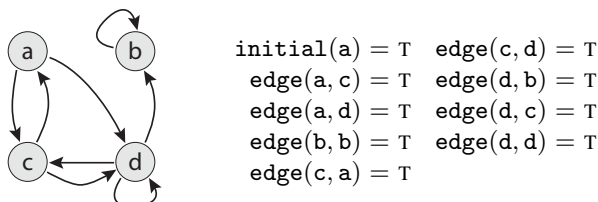


Fig. 2: A directed graph and the corresponding initial database.

Whereas Fig. 3 describes a cost graph, in Fig. 4 weights on edges are to be interpreted as *probabilities*, so that the graph can be seen as a Markov model or probabilistic finite-state network over which random walks are well-defined.¹ If we replace :- (disjunction) and , (conjunction) with $\max =$ and \times , then the value of $\text{reachable}(X)$ for any X is the probability of the most likely path from a to X . For instance, $\text{reachable}(a)$ ends up with the value 1, and $\text{reachable}(b)$ ends up with value 0.16, corresponding to the path from $a \rightarrow d \rightarrow b$, whose weight is $\text{initial}(a) \times \text{edge}(a, d) \times \text{edge}(d, b)$.

If we keep the initial database from Fig. 4 but change our operators from $\max =$ and \times to $+=$ and \times , the result is a program for *summing* over the probabilities of all distinct paths that start in a and lead to X , for each vertex X . This quantity is known as the “path sum” [29]. The path sum for b , for instance, is 10—this is not a probability, but rather a sum of probabilities of many paths, some of which are prefixes of each other.

These three related weighted logic programs are useful generalizations of the reachability logic program. Fig. 5 gives a generic representation of all four algorithms in the Dyna language [6]. The key difference among them is the *semiring* in which we interpret the weights.² Reachability uses the Boolean semiring $\langle \{T, F\}, \vee, \wedge, F, T \rangle$, single-source shortest path uses $\langle \mathbb{R}_{\geq 0} \cup \{\infty\}, \min, +, \infty, 0 \rangle$, the most-probable path variant uses $\langle [0, 1], \max, \times, 0, 1 \rangle$, and the probabilistic path-sum variant uses $\langle \mathbb{R}_{\geq 0} \cup \{\infty\}, +, \times, 0, 1 \rangle$. We require, following Goodman [12], that the semirings we use be **complete**. Complete semirings are semirings with the additional property that they are closed under finite products and infinite sums—in our running example, this corresponds to the idea that there may be infinite paths through a graph, all with finite length. Complete semirings also have the property that infinite sums behave like finite ones—they are associative and commutative, and the multiplicative operator distributes over them.

Weighted logic programming arose in the computational linguistics community [12] after it was argued by Shieber, Schabes, and Pereira [25] and Sikkil [26] that many parsing algorithms for non-deterministic grammars could be represented as deductive logic programs, and McAllester [22] showed that this representation facilitates reasoning about asymptotic complexity. Other developments include a connection between weighted logic programs and hypergraphs [18], optimal A^* search for maximizing programs [9], semiring-general agenda-based implementations [7], improved k -best algorithms [16], and program transformations to improve efficiency [5].

¹ For each vertex, the out-going edges’ weights must be non-negative and sum to a value less than or equal to one. Remaining probability mass is assumed to go to a “stopping” event, as happens with probability 0.1 in vertex b in Fig. 4.

² An algebraic semiring consists of five elements $\langle \mathbb{K}, \oplus, \otimes, \mathbf{0}, \mathbf{1} \rangle$, where \mathbb{K} is a domain closed under \oplus and \otimes , \oplus is a binary, associative, commutative operator, \otimes is a binary, associative operator that distributes over \oplus , $\mathbf{0} \in \mathbb{K}$ is the \oplus -identity, and $\mathbf{1} \in \mathbb{K}$ is the \otimes -identity.

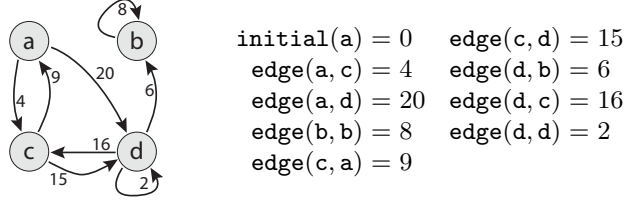


Fig. 3: A cost graph and the corresponding initial database.

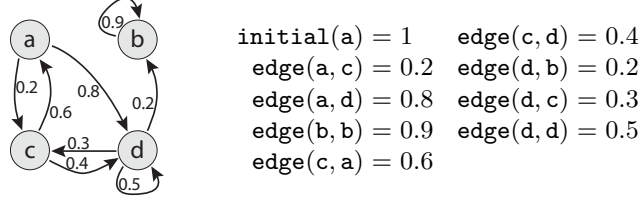


Fig. 4: A probabilistic graph and the corresponding initial database. With stopping probabilities made explicit, this would encode a Markov model.

2.2 Formal Definition

A weighted logic program is a set of “Horn equations” [5] describing a set of declarative, usually recursive equations over an abstract semiring:

$$\text{consequent}(\mathbf{U}) \oplus = \text{antecedent}_1(\mathbf{W}_1) \otimes \dots \otimes \text{antecedent}_n(\mathbf{W}_n).$$

Here \mathbf{U} and the \mathbf{W}_i are sequences of variables X_1, \dots, X_k . If $\mathbf{U} \subseteq \bigcup_{i=1}^n \mathbf{W}_i$ for every rule, then the program is *range-restricted* or *fully grounded*.

A weighted logic program is specified on an arbitrary semiring, and can be interpreted in any semiring $\langle \mathbb{K}, \oplus, \otimes, \mathbf{0}, \mathbf{1} \rangle$, as previously described.

Typically the proof and value of a *specific* theorem are desired. We assume that this theorem is called `goal` and takes zero arguments. A computationally uninteresting but perfectly intuitive way to present a weighted logic program is

$$\text{goal} \oplus = \text{axiom}_1(\mathbf{W}_1) \otimes \dots \otimes \text{axiom}_n(\mathbf{W}_n).$$

The value of the proposition/theorem `goal` is a semiring-sum over all of its proofs, starting from the axioms, where the value of any single proof is the semiring-product of the axioms involved. This is effectively encoded using the inference rules as a sum of products of sums of products of ... sums of products, exploiting distributivity and shared substructure for efficiency. Dynamic programming algorithms, useful for problems with a large degree of shared substructure, are often encoded as weighted logic programs.

In many practical applications, as in our reachability example in Section 2.1, values are interpreted as probabilities to be maximized or summed or costs to be minimized.

3 Products of Experts

In machine learning, probability models learned from example data are often used to make predictions. For example, to predict the value of a random variable Y (ranging over values denoted y in a domain denoted \mathcal{Y} ; here, \mathcal{Y} corresponds to the set of proofs) given that random variable X has an observed value $x \in \mathcal{X}$ (here, \mathcal{X} ranges over initial databases, i.e., sets of axioms), the Bayes decision rule predicts:

$$\hat{y} = \operatorname{argmax}_{y \in \mathcal{Y}} p(Y = y \mid X = x) = \operatorname{argmax}_{y \in \mathcal{Y}} \frac{p(Y = y, X = x)}{p(X = x)} \quad (5)$$

$$\text{reachable}(\mathbf{Q}) \oplus= \text{initial}(\mathbf{Q}). \quad (3)$$

$$\text{reachable}(\mathbf{Q}) \oplus= \text{reachable}(\mathbf{P}) \otimes \text{edge}(\mathbf{P}, \mathbf{Q}). \quad (4)$$

Fig. 5: The logic program from Fig. 1, rewritten to emphasize that it is generalized to an arbitrary semiring.

$$\text{reachable}_1(\mathbf{Q}_1) \oplus= \text{initial}_1(\mathbf{Q}_1). \quad (7)$$

$$\text{reachable}_1(\mathbf{Q}_1) \oplus= \text{reachable}_1(\mathbf{P}_1) \otimes \text{edge}_1(\mathbf{P}_1, \mathbf{Q}_1). \quad (8)$$

$$\text{reachable}_2(\mathbf{Q}_2) \oplus= \text{initial}_2(\mathbf{Q}_2). \quad (9)$$

$$\text{reachable}_2(\mathbf{Q}_2) \oplus= \text{reachable}_2(\mathbf{P}_2) \otimes \text{edge}_2(\mathbf{P}_2, \mathbf{Q}_2). \quad (10)$$

Fig. 6: Two identical “experts” for generalized graph reachability, duplicates of the program in Fig. 5.

In other words, the prediction \hat{y} should be the element of \mathcal{Y} that maximizes $p(Y = y \mid X = x)$, the likelihood of the event $Y = y$ given that the event $X = x$ has happened. By the definition of conditional probability, this quantity is equivalent to the ratio of the *joint* probability that $X = x \wedge Y = y$ to the *marginal* probability that $X = x$. Dynamic programming algorithms are available for solving many of these maximization problems, such as when Y ranges over paths through a graph or grammar derivations.

Of recent interest are probability models p that take a *factored form*, for example:

$$p(X = x, Y = y) \propto p_1(X = x, Y = y) \times \dots \times p_n(X = x, Y = y) \quad (6)$$

where \propto signifies “proportional to” and suppresses the means by which the probability distribution is re-normalized to sum to one. This kind of model is called a *product of experts* [14]. Intuitively, the probability of an event under p can only be large if “all the experts concur,” i.e. if the probability is large under each of the p_i . Any single expert can make an event arbitrarily unlikely (even impossible) by giving it very low probability.

The attraction of such probability distributions is that they modularize complex systems [17, 20]. They can also offer computational advantages when solving Eq. 5 [1]. Further, the expert factors can often be trained (i.e., estimated from data) separately, speeding up expensive but powerful machine learning methods [27, 4, 28].

This idea is still useful even when not dealing with probabilities. Suppose each expert p_i is a function $\mathcal{X} \times \mathcal{Y} \rightarrow \{0, 1\}$ that returns 1 if and only if the arguments x and y satisfy some constraints; it implements a relation. Then the “product” relation is just the intersection of all pairs $\langle x, y \rangle$ for which all the expert factors’ relations hold.

To the best of our knowledge, there has been no attempt to formalize the following intuitive idea about products of experts: algorithms for summing and maximizing mutually-constrained pairs of product-proof values should resemble the individual algorithms for each of the two separate proofs’ values. Our formalization is intended to aid in algorithm development as new kinds of random variables are coupled, with a key practical advantage: the expert factors are known because they fundamentally *underlie* the main algorithm. Indeed, we call our algorithms “products” because they are derived from “factors.”

4 Products of Weighted Logic Programs

In this section, we will motivate products of weighted logic programs in the context of the running example of generalized graph reachability. We will then define the **PRODUCT** transformation precisely and describe the process of specifying new algorithms as constrained versions of product programs.

Fig. 6 defines two “experts,” copies of the graph reachability program from Fig. 5. We are interested in a new predicate $\text{reachable}_{1 \circ 2}(\mathbf{Q}_1, \mathbf{Q}_2)$, which for any particular X and Y should be equal to the product of $\text{reachable}_1(X)$ and $\text{reachable}_2(Y)$. We could define the predicate by adding the following rule to the program in Fig. 6:

$$\text{reachable}_{1 \circ 2}(\mathbf{Q}_1, \mathbf{Q}_2) \oplus= \text{reachable}_1(\mathbf{Q}_1) \otimes \text{reachable}_2(\mathbf{Q}_2).$$

$$\text{reachable}_{1\circ 2}(\mathbb{Q}_1, \mathbb{Q}_2) \oplus = \text{initial}_1(\mathbb{Q}_1) \otimes \text{initial}_2(\mathbb{Q}_2). \quad (11)$$

$$\text{reachable}_{1\circ 2}(\mathbb{Q}_1, \mathbb{Q}_2) \oplus = \text{reachable}_2(\mathbb{P}_2) \otimes \text{edge}_2(\mathbb{P}_2, \mathbb{Q}_2) \otimes \text{initial}_1(\mathbb{Q}_1). \quad (12)$$

$$\text{reachable}_{1\circ 2}(\mathbb{Q}_1, \mathbb{Q}_2) \oplus = \text{reachable}_1(\mathbb{P}_1) \otimes \text{edge}_1(\mathbb{P}_1, \mathbb{Q}_1) \otimes \text{initial}_2(\mathbb{Q}_2). \quad (13)$$

$$\text{reachable}_{1\circ 2}(\mathbb{Q}_1, \mathbb{Q}_2) \oplus = \text{reachable}_{1\circ 2}(\mathbb{P}_1, \mathbb{P}_2) \otimes \text{edge}_1(\mathbb{P}_1, \mathbb{Q}_1) \otimes \text{edge}_2(\mathbb{P}_2, \mathbb{Q}_2). \quad (14)$$

Fig. 7: Four rules that, in addition to the rules in Fig. 6, define the product of experts of reachable_1 and reachable_2 .

Input: A logic program \mathcal{P} and a set \mathcal{S} of pairs of predicates (p, q) .

Output: A program \mathcal{P}' that extends \mathcal{P} , additionally computing the product predicate $p \circ q$ for every pair $(p, q) \in \mathcal{S}$ in the input.

```

1:  $\mathcal{P}' \leftarrow \mathcal{P}$ 
2: for all pairs  $(p, q)$  in  $\mathcal{S}$  do
3:   for all rules in  $\mathcal{P}$ , of the form  $p(\mathbf{W}) \oplus = A_1 \otimes \dots \otimes A_n$  do
4:     for all rules in  $\mathcal{P}$ , of the form  $q(\mathbf{X}) \oplus = B_1 \otimes \dots \otimes B_m$  do
5:       let  $r \leftarrow [p \circ q(\mathbf{W}, \mathbf{X}) \oplus = A_1 \otimes \dots \otimes A_n \otimes B_1 \otimes \dots \otimes B_m]$ 
6:       for all pairs of antecedents in  $r$   $(\mathbf{s}(\mathbf{Y}), \mathbf{t}(\mathbf{Z}))$  such that  $(\mathbf{s}, \mathbf{t}) \in \mathcal{S}$  do
7:         remove the antecedents  $\mathbf{s}(\mathbf{Y})$  and  $\mathbf{t}(\mathbf{Z})$  from  $r$ 
8:         insert the antecedent  $\mathbf{s} \circ \mathbf{t}(\mathbf{Y}, \mathbf{Z})$  to  $r$ 
9:       end for
10:      add  $r$  to  $\mathcal{P}'$ 
11:     end for
12:   end for
13: end for
14: return  $\mathcal{P}'$ 

```

Fig. 8: This figure describes **PRODUCT**, a non-deterministic program transformation that adds new rules to WLP \mathcal{P} that compute the product of experts of predicates from the original program. We implicitly rename variables to avoid conflicts between rules.

This program is a bit simplistic, however; it merely describes calculating the “experts” independently and then combining them at the end. The key to the **PRODUCT** transformation is that the predicate of $\text{reachable}_{1\circ 2}$ can alternatively be calculated by adding the following four rules to Fig. 6:

$$\begin{aligned} \text{reachable}_{1\circ 2}(\mathbb{Q}_1, \mathbb{Q}_2) &\oplus = \text{initial}_1(\mathbb{Q}_1) \otimes \text{initial}_2(\mathbb{Q}_2). \\ \text{reachable}_{1\circ 2}(\mathbb{Q}_1, \mathbb{Q}_2) &\oplus = \text{initial}_1(\mathbb{Q}_1) \otimes \text{reachable}_2(\mathbb{P}_2) \otimes \text{edge}_2(\mathbb{P}_2, \mathbb{Q}_2). \\ \text{reachable}_{1\circ 2}(\mathbb{Q}_1, \mathbb{Q}_2) &\oplus = \text{reachable}_1(\mathbb{P}_1) \otimes \text{edge}_1(\mathbb{P}_1, \mathbb{Q}_1) \otimes \text{initial}_2(\mathbb{Q}_2). \\ \text{reachable}_{1\circ 2}(\mathbb{Q}_1, \mathbb{Q}_2) &\oplus = \text{reachable}_1(\mathbb{P}_1) \otimes \text{edge}_1(\mathbb{P}_1, \mathbb{Q}_1) \otimes \\ &\quad \text{reachable}_2(\mathbb{P}_2) \otimes \text{edge}_2(\mathbb{P}_2, \mathbb{Q}_2). \end{aligned}$$

Then, because $\text{reachable}_{1\circ 2}(\mathbb{Q}_1, \mathbb{Q}_2)$ was defined above to be the product of $\text{reachable}_1(\mathbb{Q}_1)$ and $\text{reachable}_2(\mathbb{Q}_2)$, it we can replace the premises $\text{reachable}_1(\mathbb{Q}_1)$ and $\text{reachable}_2(\mathbb{Q}_2)$ with the single premise $\text{reachable}_{1\circ 2}(\mathbb{Q}_1, \mathbb{Q}_2)$ to obtain the factored program in Fig. 7. This program computes over pairs of paths in two graphs.

4.1 The **PRODUCT** Transformation

The **PRODUCT** program transformation is shown in Fig. 8. For each desired product of experts, where one “expert,” the predicate p , is defined by n rules and the other expert q by m rules, the transformation defines the product of experts for $p \circ q$ with $n \times m$ new rules, the cross product of inference rules from the first and second experts. The value of a coupled proposition $p \circ q$ in \mathcal{P}' will be equal to the semiring product of p ’s value and q ’s value in \mathcal{P} (or, equivalently, in \mathcal{P}').

Note that lines 6–8 are non-deterministic under certain circumstances, because if the antecedent of the combined program is $\mathbf{a}(\mathbf{X}) \otimes \mathbf{a}(\mathbf{Y}) \otimes \mathbf{b}(\mathbf{Z})$ and the algorithm is computing the product of \mathbf{a} and \mathbf{b} , then the resulting antecedent could be either $\mathbf{a} \circ \mathbf{b}(\mathbf{X}, \mathbf{Z}) \otimes \mathbf{a}(\mathbf{Y})$ or $\mathbf{a} \circ \mathbf{b}(\mathbf{Y}, \mathbf{Z}) \otimes \mathbf{a}(\mathbf{X})$. Our procedure arbitrarily selects one of the possibilities.

$$\text{reachable}_{1 \circ 2}(Q_1, Q_2) \oplus = \text{initial}_1(Q_1) \otimes \text{initial}_2(Q_2). \quad (15)$$

$$\text{reachable}_{1 \circ 2}(Q_1, Q_2) \oplus = \text{reachable}_{1 \circ 2}(P_1, P_2) \otimes \text{edge}_1(P_1, Q_1) \otimes \text{edge}_2(P_2, Q_2). \quad (16)$$

Fig. 9: By removing all but these two rules from the product of experts in Fig. 7, we require both paths to have the same number of steps.

$$\text{reachable}_{1 \circ 2}(Q) \oplus = \text{initial}_1(Q) \otimes \text{initial}_2(Q). \quad (17)$$

$$\text{reachable}_{1 \circ 2}(Q) \oplus = \text{reachable}_{1 \circ 2}(P) \otimes \text{edge}_1(P, Q) \otimes \text{edge}_2(P, Q). \quad (18)$$

Fig. 10: By further constraining the program in Fig. 9 to require that the $Q_1 = Q_2$ at all points, we require both paths to be identical.

4.2 Constraining the Product of Experts

Any program \mathcal{P}' that comes out after applying **PRODUCT** on \mathcal{P} computes the product of experts of p and q (where $(p, q) \in \mathcal{S}$). More specifically, any ground instances of $p(\mathbf{X})$ and $q(\mathbf{Y})$ have the same value in \mathcal{P} and \mathcal{P}' , and the value of $p \circ q(\mathbf{X}, \mathbf{Y})$ in \mathcal{P}' is $p(\mathbf{X}) \otimes q(\mathbf{Y})$.³ However, the same program could have been implemented more straightforwardly by merely introducing a new inference rule for the goal.

Yet, the output of the **PRODUCT** transformation is a starting point for describing dynamic programming algorithms that do two similar actions—traversing a graph, scanning a string, parsing a sentence—at the same time and in a coordinated fashion. Exactly what “coordinated fashion” means depends on the problem, and answering that question determines how the problem is constrained.

If we return to the running example of generalized graph reachability, the program as written has eight rules, four from Fig. 6 and four from Fig. 7. Two examples of constrained product programs are given in Fig. 9 and Fig. 10. In the first example in Fig. 9, the only change is that all but two rules have been removed from the program in Fig. 7. Whereas in the original product program $\text{reachable}_{1 \circ 2}(Q_1, Q_2)$ corresponded to the product of the weight of the “best” path from the initial state or states of graph 1 to Q_1 and the weight of the “best” path from the initial state or states of graph 2 to Q_2 , the new program computes the best paths from the two origins to the two destinations with the additional requirement that the paths be the *same length*—the rules that were deleted allowed for the possibility of a prefix on one path or the other.

If our intent is for the two paths to not only have the same length but to visit vertices in the same sequence, then we can further constrain the program to only define $\text{reachable}_{1 \circ 2}(Q_1, Q_2)$ where $Q_1 = Q_2$, at which point it might as well be written $\text{reachable}_{1 \circ 2}(Q)$. This is what is done in Fig. 10.

The choice of paired predicates \mathcal{S} is important for the final WLP which **PRODUCT** returns and it also limits the way we can add constraints to derive a new WLP. Automatically deriving \mathcal{S} from data in a machine learning setting is an open question for future research. When **PRODUCT** is applied on two copies of the same WLP (concatenated together to a single program), a natural schema for selecting paired predicates arises, in which we pair a predicate from one program with the same predicate from the other program. This natural pairing leads to the derivation of several useful, known algorithms, to which we turn in Section 5.

We conclude this section with a proof that the **PRODUCT** transformation results in a product of experts calculation.

Theorem 1. *Let \mathcal{P} be a weighted logic program over a set of predicates \mathcal{R} , and let \mathcal{S} be a set of pairs of predicates from \mathcal{P} . Then after applying **PRODUCT** on $(\mathcal{P}, \mathcal{S})$, resulting in a new program \mathcal{P}' , the following holds:*

1. *Every ground instance $p(\mathbf{X})$, where $p \in \mathcal{R}$, has the same value in \mathcal{P} and \mathcal{P}' .*
2. *For every $(p, q) \in \mathcal{S}$, the value $p \circ q(\mathbf{X}, \mathbf{Y})$ in \mathcal{P}' is $p(\mathbf{X}) \otimes q(\mathbf{Y})$.*

³ See proof below.

Proof: The first condition holds trivially because \mathcal{P}' is stratified: none of the new rules in \mathcal{P}' are ever used to compute values of the form $p(\mathbf{X})$, where $p \in \mathcal{R}$. Hence, their value is identical to their value in \mathcal{P} .

By distributivity of the semiring at hand, we know that $p(\mathbf{X}) \otimes q(\mathbf{Y})$ is the sum: $\bigoplus_{t,r} v(t) \otimes v(r)$ where t and r range over proofs of $p(\mathbf{X})$ and $q(\mathbf{Y})$ respectively, with their values being $v(t)$ and $v(r)$. This implies that in order to prove 2, we need to show that there is a bijection between the set of proofs A for $p \circ q(\mathbf{X}, \mathbf{Y})$ in \mathcal{P}' and the set of pairs of proofs B for $p(\mathbf{X})$ and $q(\mathbf{Y})$ such that for every $s \in A$ and $(t, r) \in B$ we have $v(s) = v(t) \otimes v(r)$.

Using structural induction over the proofs, we first show that every pair of proofs $(t, r) \in B$ has a corresponding proof $s \in A$ with the needed value. In the base case, where the proofs t and r include a single step, the correspondence follows trivially. Let $(t, r) \in B$. Without loss of generality, we will assume that both t and r contain more than a single step in their proofs. In the last step of its proof, t used a rule of the form

$$p(\mathbf{X}) \oplus= a_1(\mathbf{X}_1) \otimes \dots \otimes a_n(\mathbf{X}_n) \quad (19)$$

and r used a rule in its last step of the form

$$q(\mathbf{Y}) \oplus= b_1(\mathbf{Y}_1) \otimes \dots \otimes b_m(\mathbf{Y}_m) \quad (20)$$

Let t_i be the subproofs of $a_i(\mathbf{X}_i)$ and let r_j be subproofs of $b_j(\mathbf{Y}_j)$. It follows that **PRODUCT** creates from those two rules a single inference rule of the form:

$$p \circ q(\mathbf{X}, \mathbf{Y}) \oplus= c_1(\mathbf{W}_1) \otimes \dots \otimes c_p(\mathbf{W}_p) \quad (21)$$

where $c_i(\mathbf{W}_i)$ is either $a_k(\mathbf{Y}_k)$ for some k , or $b_k(\mathbf{Y}_k)$ for some k , or $a_k \circ b_\ell(\mathbf{X}_k, \mathbf{Y}_\ell)$ for some k, ℓ .

We resolve each case as following:

1. If $c_i(\mathbf{W}_i) = a_k(\mathbf{Y}_k)$ then we set $s_i = t_k$.
2. If $c_i(\mathbf{W}_i) = b_k(\mathbf{Y}_k)$ then we set $s_i = r_k$.
3. If $c_i(\mathbf{W}_i) = a_k \circ b_\ell(\mathbf{X}_k, \mathbf{Y}_\ell)$ then according to the induction hypothesis, we have a proof for $a_k \circ b_\ell(\mathbf{X}_k, \mathbf{Y}_\ell)$ such that its value is $v(t_k) \otimes v(r_\ell)$. We set s_i to be that proof.

Since we have shown there is a proof for each antecedent of $p(\mathbf{X}) \circ q(\mathbf{Y})$, we have shown that there is a proof for $p(\mathbf{X}) \circ q(\mathbf{Y})$. That its value is indeed $p(\mathbf{X}) \otimes q(\mathbf{Y})$ is concluded trivially from the induction steps.

The reverse direction for the constructing the bijection is similar, using again structural induction over proofs. \square

5 Examples

In this section, we describe three classes of algorithms that can be understood as constrained products of simpler weighted logic programs.

5.1 Edit Distance

Edit distances [19] are important measures of the difference between two strings, and they underlie many algorithms in computational biology and computational linguistics. The DNA fragment “ACTAGCACTTAG” can be encoded as a set of axioms $s(a, 1)$, $s(c, 2)$, $s(t, 3)$, \dots , $s(g, 12)$, and we can describe a specification of the dynamic program for edit distance by using the product of a trivial “cursor” program that scans over a string, described in Fig. 11.

We generally interpret Fig. 11 over the the “cost minimization” semiring, replacing $\oplus=$ with $\min =$ and \otimes with $+$. The value of all the axioms of the form $\text{start}(P)$ (giving the starting position) or $s(C, P)$ (indicating the position of a character) is 0, but the value of staycost is some finite, nonzero value representing the penalty if the cursor stays in one place. Note that under this semiring interpretation, the rule (23) will never be used; the value of staycost only becomes relevant when the **PRODUCT** of the scanning program with itself is determined.

The output of **PRODUCT** on two copies of the scanning program is shown in Fig. 12, though three rules are removed.⁴ One important change is made to clause 30, the addition of the *side condition* $C_1 = C_2$, which requires that

⁴ The three removed rules are the two different combinations of clause 22 and clause 23 in Fig. 11, as well as the combination of clause 23 with itself. These three rules are redundant if we are computing in the minimum-cost semiring.

$$\text{dist}(P) \oplus = \text{start}(P). \quad (22)$$

$$\text{dist}(P) \oplus = \text{dist}(P) \otimes \text{staycost}. \quad (23)$$

$$\text{dist}(P + 1) \oplus = \text{dist}(P) \otimes s(C, P). \quad (24)$$

Fig. 11: A program for scanning over a string.

$$\text{dist}_{1o2}(P_1, P_2) \oplus = \text{start}_1(P_1) \otimes \text{start}_2(P_2). \quad (25)$$

$$\text{dist}_{1o2}(P_1, P_2 + 1) \oplus = \text{start}_1(P_1) \otimes \text{dist}_2(P_2) \otimes s(C_2, P_2). \quad (26)$$

$$\text{dist}_{1o2}(P_1 + 1, P_2) \oplus = \text{dist}_1(P_1) \otimes s(C_1, P_1) \otimes \text{start}_2(P_2). \quad (27)$$

$$\text{dist}_{1o2}(P_1, P_2 + 1) \oplus = \text{dist}_{1o2}(P_1, P_2) \otimes s_2(C_2, P_2) \otimes \text{staycost}_1. \quad (28)$$

$$\text{dist}_{1o2}(P_1 + 1, P_2) \oplus = \text{dist}_{1o2}(P_1, P_2) \otimes s_1(C_1, P_1) \otimes \text{staycost}_2. \quad (29)$$

$$\text{dist}_{1o2}(P_1 + 1, P_2 + 1) \oplus = \text{dist}_{1o2}(P_1, P_2) \otimes s_2(C_1, P_1) \otimes s_2(C_2, P_2) \boxed{\text{if } C_1 = C_2.} \quad (30)$$

Fig. 12: Edit distance derived from the PRODUCT transformation on two copies of Fig. 11, with a side condition (boxed).

when both P_1 and P_2 advance, the character at position P_1 in string 1 and the character at position P_2 in string two must be identical. This captures the essential requirement of the edit distance calculation: changing a symbol in one string to an identical symbol in the other string incurs no “edit” and no cost. It is worth noting that the clause 30 could have equivalently been written by unifying C_1 with C_2 :

$$\text{dist}_{1o2}(P_1 + 1, P_2 + 1) \oplus = \text{dist}_{1o2}(P_1, P_2) \otimes s_2(C, P_1) \otimes s_2(C, P_2).$$

The first and last clauses in the edit distance program (25 and 30) are the essential ones; the other five clauses essentially describe extensions to edit distance that can be added, turned off, or modified to obtain different edit distances. Clause 26 allows a penalty-free prefix to be added to the second string, for instance matching string 1, “BOVIK” against string 2, “HARRY BOVIK,” and clause 27 allows a penalty-free prefix to be deleted from the first string. Clause 28 describes that insertions can be made with cost staycost_1 , for instance in matching string one “H. BOVIK” against “H. Q. BOVIK”, and clause 29 describes deletions from the first string to reach the second.

5.2 Finite-State Algorithms

Specifications of weighted finite-state automata (WFSAs) and transducers (WFSTs) are superficially similar to the reachability problem of Sec. 1, but with edge relations ($\text{edge}(P, Q)$) augmented by symbols (WFSAs: $\text{arc}(P, Q, A)$) or pairs of input-output symbols (WFSTs: $\text{arc}(P, Q, A, B)$). Weighted finite-state machines are widely used in speech and language processing [24].

Weighted Finite State Automata. Fig. 13 describes an algorithm for recognizing paths in a weighted finite state automaton. (With the appropriate semirings, it finds the most probable path or the path-sum.) If the PRODUCT of that algorithm with itself is taken, we can follow similar steps in Sec. 4.2 and add a constraint to clause 36 that requires the two paths’ symbols to be identical, we get the recognizer for the (*weighted*) *intersection* of the two WFSAs (itself a WFSAs). Weighted intersection generalizes intersection, and can be used, for example, to determine whether a specific string (itself an FSA) is in the regular language of the FSA and, in the probabilistic case, its associated probability.

Weighted Finite-State Transducers. Suppose we take the PRODUCT transformation of the WFST recognition algorithm (not shown, but similar to Fig. 13 but using $\text{arc}(P, Q, A, B)$ as arc axioms) with itself and constrain the result by removing all but the three interesting rules (as before) and requiring that B_1 (the “output” along the first edge) always be equal to A_2 (the “input” along the second edge). The result is shown in Fig. 15; this is the recognition algorithm for the WFST resulting from *composition* of two WFSTs. Composition permits small, understandable components to be cascaded and optionally compiled, forming complex but efficient models over strings.

$$\text{goal} \oplus= \text{path}(Q) \otimes \text{final}(Q). \quad (31)$$

$$\text{path}(Q) \oplus= \text{initial}(Q). \quad (32)$$

$$\text{path}(Q) \oplus= \text{path}(P) \otimes \text{arc}(P, Q, A). \quad (33)$$

Fig. 13: The weighted logic program describing (weighted) recognition in a probabilistic finite state automaton.

$$\text{goal}_{1 \circ 2} \oplus= \text{path}_{1 \circ 2}(Q_1, Q_1) \otimes \text{final}_1(Q_2) \otimes \text{final}_2(Q_2). \quad (34)$$

$$\text{path}_{1 \circ 2}(Q_1, Q_2) \oplus= \text{initial}_1(Q_1) \otimes \text{initial}_2(Q_2). \quad (35)$$

$$\text{path}_{1 \circ 2}(Q_1, Q_2) \oplus= \text{path}_{1 \circ 2}(P_1, P_2) \otimes \text{arc}_1(P_1, Q_1, A_1) \otimes \text{arc}_2(P_2, Q_2, A_2) \boxed{\text{if } A_1 = A_2.} \quad (36)$$

Fig. 14: The weighted logic program describing (weighted) recognition by an intersection of two finite state automata, derived from Fig. 13 in the manner of Fig. 9.

$$\text{goal}_{1 \circ 2} \oplus= \text{path}_{1 \circ 2}(Q_1, Q_1) \otimes \text{final}_1(Q_2) \otimes \text{final}_2(Q_2). \quad (37)$$

$$\text{path}_{1 \circ 2}(Q_1, Q_2) \oplus= \text{initial}_1(Q_1) \otimes \text{initial}_2(Q_2). \quad (38)$$

$$\text{path}_{1 \circ 2}(Q_1, Q_2) \oplus= \text{path}_{1 \circ 2}(P_1, P_2) \otimes \text{arc}_1(P_1, Q_1, A_1, B_1) \otimes \text{arc}_2(P_2, Q_2, A_2, B_2) \boxed{\text{if } B_1 = A_2.}$$

Fig. 15: The weighted logic program describing a composition of two finite state transducers, derived from Fig. 13 in the manner of Fig. 9 and Fig. 14.

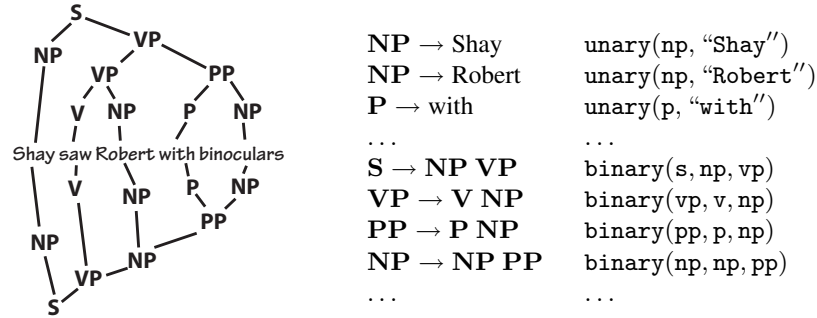


Fig. 16: An ambiguous sentence that can be parsed two ways in English (left), some of the Chomsky normal form rules for English grammar (center), and the corresponding axioms (right). There would also need to be five axioms of the form `string("Shay", 1)`, `string("saw", 2)`, etc.

5.3 Context-Free Parsing

Parsing natural languages is a difficult, central problem in computational linguistics [21]. Consider the sentence "Shay saw Robert with binoculars." One analysis (the most likely in the real world) is that Shay had the binoculars and saw Robert through them. Another is that *Robert* had the binoculars, and Shay saw the binocular-endowed Robert. Fig. 16 shows syntactic parses into noun phrases (NP), verb phrases (VP), etc., corresponding to these two meanings. It also shows part of a context-free grammar describing English sentences in Chomsky normal form [15],⁵ and an encoding of the grammar and string using axioms. A proof corresponds to a CF derivation of the string.

In [25], the authors show that parsing with CFGs (and other grammars) can be formalized as a logic program, and in [12] this framework is extended to the weighted case. If weights are interpreted as probabilities, then these

⁵ Chomsky normal form (CNF) means that the rules in the grammar are either binary with two nonterminals or unary with a terminal. We do not allow ϵ rules, which in general are allowed in CNF grammars.

$$\text{goal} \oplus= \text{start}(S) \otimes \text{length}(N) \otimes c(S, 0, N). \quad (39)$$

$$c(X, I - 1, I) \oplus= \text{unary}(X, W) \otimes \text{string}(W, I). \quad (40)$$

$$c(X, I, K) \oplus= \text{binary}(X, Y, Z) \otimes c(Y, I, J) \otimes c(Z, J, K). \quad (41)$$

Fig. 17: CKY: a weighted logic program implementing weighted CKY for algorithms involving weighted context free grammars in Chomsky normal form. Strictly speaking, CKY refers to a naïve bottom-up evaluation strategy for this program.

$$\text{goal}_{1\circ 2} \oplus= \text{length}_1(N_1) \otimes \text{length}_2(N_2) \otimes \quad (42)$$

$$\text{start}_1(S_1) \otimes \text{start}_2(S_2) \otimes c_{1\circ 2}(S_1, 0, N_1, S_2, 0, N_2).$$

$$c_{1\circ 2}(X_1, I_1 - 1, I_1, X_2, I_2 - 1, I_2) \oplus= \text{unary}_1(X_1, W_1) \otimes \text{string}_1(W_1, I_1) \otimes \quad (43)$$

$$\text{unary}_2(X_2, W_2) \otimes \text{string}_2(W_2, I_2).$$

$$c_{1\circ 2}(X_1, I_1 - 1, I_1, X_2, I_2, K_2) \oplus= \text{unary}_1(X_1, W_1) \otimes \text{string}_1(W_1, I_1) \otimes \quad (44)$$

$$\text{binary}_2(X_2, Y_2, Z_2) \otimes c_2(Y_2, I_2, J_2) \otimes c_2(Z_2, J_2, K_2).$$

$$c_{1\circ 2}(X_1, I_1, K_1, X_2, I_2 - 1, I_2) \oplus= \text{unary}_2(X_2, W_1) \otimes \text{string}_2(W_2, I_2) \otimes \quad (45)$$

$$\text{binary}_1(X_1, Y_1, Z_1) \otimes c_1(Y_1, I_1, J_1) \otimes c_1(Z_1, J_1, K_1).$$

$$c_{1\circ 2}(X_1, I_1, K_1, X_2, I_2, K_2) \oplus= \text{binary}_1(X_1, Y_1, Z_1) \otimes \text{binary}_2(X_2, Y_2, Z_2) \otimes \quad (46)$$

$$c_{1\circ 2}(Y_1, I_1, J_1, Y_2, I_2, J_2) \otimes c_{1\circ 2}(Z_1, J_1, K_1, Z_2, K_2, J_2).$$

Fig. 18: The full output of the PRODUCT transformation on two copies of CKY in Fig. 17.

$$\text{goal}_{1\circ 2} \oplus= \text{length}(N) \otimes \text{start}_1(S_1) \otimes \text{start}_2(S_2) \otimes c_{1\circ 2}(S_1, S_2, 0, N). \quad (47)$$

$$c_{1\circ 2}(X_1, X_2, I - 1, I) \oplus= \text{unary}_1(X_1, W_1) \otimes \text{string}_1(W_1, I) \otimes \quad (48)$$

$$\text{unary}_2(X_2, W_2) \otimes \text{string}_2(W_2, I).$$

$$c_{1\circ 2}(X_1, X_2, I, K) \oplus= \text{binary}_1(X_1, Y_1, Z_1) \otimes \text{binary}_2(X_2, Y_2, Z_2) \otimes \quad (49)$$

$$c_{1\circ 2}(Y_1, Y_2, I, J) \otimes c_{1\circ 2}(Z_1, Z_2, J, K).$$

Fig. 19: The program in Fig. 18 constrained to require parsing two different sentences with the same parse tree.

two semiring interpretations can either find the “weight” of the parse with maximum weight or the total weight of all parse trees (a measure of the “total grammaticality” of a sentence). In this section, we give the specification of the weighted CKY algorithm [2], which is a dynamic programming algorithm for parsing using a context-free grammar in Chomsky normal form. The CKY algorithm is shown in Fig. 17. We show that fundamental algorithms for weighted (probabilistic) parsing can be derived as constrained PRODUCTs of CKY.

The unconstrained PRODUCT of CKY with itself (Fig. 18) is not inherently interesting. It is worth noting, however, that clause 46 there was a choice as to how to merge the c_1 and c_2 possibilities. The choice would not have existed if, instead of the presentation of CKY in Fig. 17, a common *binarized* variant of the algorithm, which introduces a new predicate in order to have at most two antecedents per Horn equation, had been fed to PRODUCT. The choice that we made in pairing was consistent with the choice that is forced in the binarized CKY program.

Product of Grammars. Fig. 19 describes a more interesting constrained version of Fig. 18. In particular, in all cases the constraints $I_1 = I_2, J_1 = J_2, K_1 = K_2, N_1 = N_2$ are added, so that instead of writing $c_{1\circ 2}(X_1, I_1, J_1, X_2, I_2, J_2)$ we just write $c_{1\circ 2}(X_1, X_2, I, J)$. This program simultaneously parses two different sentences using two different grammars, but both parses must have the same *structure*. This constraint, then, can be compared to the constraints placed on the product of two graph-reachability programs to ensure that both paths have the same length.

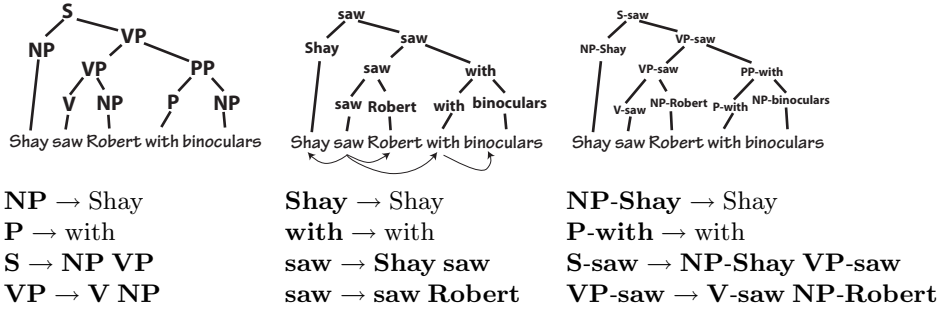


Fig. 20: On the left, the grammar previously shown. In the middle, a context-free *dependency grammar*, whose derivations can be seen as parse trees (above) or a set of dependencies (below). On the right, a lexicalized grammar. Sample rules are given for each grammar.

$$\text{goal}_{1\circ 2} \oplus= \text{length}(N) \otimes \text{start}_1(S_1) \otimes \text{start}_2(S_2) \otimes c_{1\circ 2}(S_1, S_2, 0, N). \quad (50)$$

$$c_{1\circ 2}(X_1, X_2, I - 1, I) \oplus= \text{unary}_1(X_1, W_1) \otimes \text{unary}_2(X_2, W_2) \otimes \text{string}(W, I). \quad (51)$$

$$c_{1\circ 2}(X_1, X_2, I, K) \oplus= \text{binary}_1(X_1, Y_1, Z_1) \otimes \text{binary}_2(X_2, Y_2, Z_2) \otimes c_{1\circ 2}(Y_1, Y_2, I, J) \otimes c_{1\circ 2}(Z_1, Z_2, J, K). \quad (52)$$

Fig. 21: Constraining Fig. 18 to simultaneously parse the *same* sentence with two grammars.

Lexicalized CFG Parsing. An interesting variant of the previous rule involves *lexicalized grammars*, which are motivated in Fig. 20. Instead of describing a grammar using nonterminals denoting phrases (e.g., NP and VP), we can define a (context-free) *dependency grammar* [10] that encodes the syntax of a sentence in terms of parent-child relationships between words. In the case of the example of Fig. 20, the arrows below the sentence in the middle establish “saw” as the root of the sentence; the word “saw” has three children (arguments and modifiers), one of which is the word “with,” which in turn has the child “binoculars.”

The simplest approach to describing a dependency grammar is to define it as a Chomsky normal form grammar where the nonterminal set is equivalent to the set of terminal symbols (so that the terminal “with” corresponds to a unique nonterminal **with**, and so on) and where all rules have the form $P \rightarrow P C$, $P \rightarrow C P$, and $W \rightarrow w$ (where X is the nonterminal version of terminal x).

Fig. 21 describes a further constrained program that, instead of parsing two unique strings in two different grammars with the same structure, parses a single string in two different grammars with the same structure. This new grammar recognizes a string if and only if both of the original grammars recognize it with isomorphic trees—a kind of “derivation intersection.” (This is not to be confused with intersections of context-free *languages*, which are not in general context-free languages [15].)

If we encode the regular grammar in the unary_1 and binary_1 relations and encode a dependency grammar in the unary_2 and binary_2 relations, then the product is a *lexicalized grammar*, like the third example from Fig. 20. In particular, it describes a lexicalized context-free grammar with a product of experts probability model [17], because the weight given to the production $A-X \rightarrow B-X C-Y$, for instance, is the semiring-product of the weight given to the production $A \rightarrow B C$ and the weight given to the dependency based production $X \rightarrow X Y$. If, instead of the axioms of the form $\text{binary}_1(X_1, Y_1, Z_1)$ and $\text{binary}_2(X_2, Y_2, Z_2)$ there were axioms of the form $\text{binary}_{1\circ 2}(X_1, X_2, Y_1, Y_2, Z_1, Z_2)$ and clause 49 was changed accordingly, then the result would be a general lexicalized CKY [8]. This is an instance of a general pattern of modifications to the output of PRODUCT that we term “axiom generalization” in §6.3.

Synchronous Parsing. Another extension to context-free parsing, *synchronous parsing*, can be derived using PRODUCT from two instances of CKY. Here two strings are parsed, each in a different alphabet with a different grammar (e.g., a French sentence and its English translation). A synchronous derivation consists of two trees and a correspondence between their nodes; different degrees of isomorphism may be imposed (e.g., in natural language, reordering is

common, but dependencies tend to be mirrored through word-level translation). Constraining the **PRODUCT** of CKY with itself with side conditions to impose a one-to-one correspondence of nonterminals leads to a weighted logic program for a formalism known as inversion transduction grammar [30]. Constraining the **PRODUCT** of a more general CKY that includes empty unary rules “ $X \rightarrow \epsilon$ ” and leaving rules that pair unary with binary antecedents removes the requirement that the sentences have the same length. In addition, the **PRODUCT** of the lexicalized CKY with itself leads to WLPs for more complex parsers like those described in [23] and [31] for more expressive formalisms.

Interestingly, building a synchronous parser from the already-binarized CKY gives the generalization of the “hook” trick from [8] without any extra effort (cf. Zhang and Gildea, 2006).

As with lexicalized parsing, we can keep the factorized form for the new grammar rules [27], or we can make the new rules axioms, granting more freedom in the model [30]. See §6.3 for more about this.

6 Variations on PRODUCT

The previous sections of this paper concentrate on *constrained* products of weighted logic programs. Furthermore, we always were interested in the product of two “experts” with similar structure—the constrained product of two finite-state transducers is a composition of those transducers, the constrained product of two CFG parsers is a lexical parser. In this section, we will elaborate on variations on this theme that are also worth considering: generalizing the form of side conditions, running **PRODUCT** on two experts with different structure, and generalizing the axioms to take away the requirement that our “experts” be factorable into a product of two experts.

6.1 More General Side Conditions

An obvious extension that is worth noting is that we can add constraints that are more interesting than the equality constraints we considered earlier. Our constraints before were essentially “Boolean,” in the sense that they either left the value of the proof the same if they were satisfied, or else they changed the value of the proof to $\mathbf{0}$ if they were not satisfied (where $\mathbf{0}$ is the zero element of whatever semiring we are in). A more general constraint might be defined as one that *reduces* the value of a semiring.

What do we mean by “reduce”? We can define a relation $<$ on semirings as $a < b = \exists c. a \oplus c = b$. If semirings exhibit the property that

- $<$ is a partial order
- $\forall a. \mathbf{0} < a$ (the zero of the semiring is the least element)

... then a quantity c is a *constraining value* if and only if $\forall a. c \otimes a \leq a$.

A good example of using a more general side condition is in the edit distance example from Section 5.1. In Clause 30 of Fig. 12, the constraint $\text{if } C_1 = C_2$ was added to the rule to require that the symbol C_1 in the source string and the symbol C_2 in the destination string be identical. This constraint is ripe for generalization.

Say we had instead used a constraint $\text{rot13}(C_1, C_2)$ which had defined axioms $\text{rot13}(a, n)$, $\text{rot13}(b, o)$, etc., all with value 0 (which, recall, is the 1 of the edit distance semiring). Then we would be trying to determine the edit distance between a string and a ROT-13 encrypted version of itself! More realistically, we can use this to express the idea of a *replacement cost* that we may want to be less than the cost of a deletion followed by an insertion. In the DNA matching program, we could replace $\text{if } C_1 = C_2$ with $\text{swap}(C_1, C_2)$ and define the axioms $\text{swap}(A, A)$, $\text{swap}(C, C)$, $\text{swap}(G, G)$, and $\text{swap}(T, T)$ to have value 0. Then, if we wanted to set a lower cost c for the action of replacing A in the source string with T in the destination string, we could define a new axiom $\text{swap}(A, T)$ with value c .

6.2 Product of Experts Training and Decoding

Until now, we have presented the **PRODUCT** transformation as a transformation which operates within a single weighted logic program, on a subset of rules, which are coupled and also possibly constrained. Sometimes, it is more convenient to view **PRODUCT** as a binary operator which is applied on two different weighted logic programs, \mathcal{P}_1 and \mathcal{P}_2 . In that case, the set \mathcal{S} of pairs of predicates to be coupled in Fig. 8, \mathcal{S} is a subset of the Cartesian product

$$\text{path}_{1 \circ 2}(\mathcal{Q}_1, \mathcal{Q}_2) \oplus = \text{initial}_{1 \circ 2}(\mathcal{Q}_1, \mathcal{Q}_2) \quad (54)$$

$$\text{path}_{1 \circ 2}(\mathcal{Q}_1, \mathcal{Q}_2) \oplus = \text{path}_{1 \circ 2}(\mathcal{P}_1, \mathcal{P}_2) \otimes \text{arc}_{1 \circ 2}(\mathcal{P}_1, \mathcal{P}_2, \mathcal{Q}_1, \mathcal{Q}_2, \mathbf{A}_1, \mathbf{A}_2) \quad (55)$$

Fig. 22: Weighted finite state transducers as the product of two weighted finite state machines.

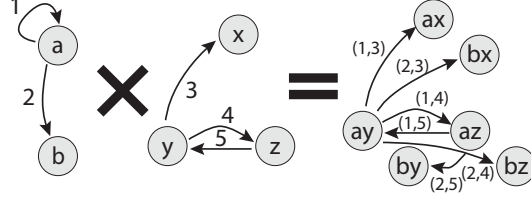


Fig. 23: A finite-state transducer that can be expressed as the PRODUCT of two finite state automata.

between the set of predicates from \mathcal{P}_1 and the set of predicates from \mathcal{P}_2 . Then, as usual, the resulting program can be constrained.

This view can make it more intuitive to deal with calculations over two correlated structures, each represented by a different program, \mathcal{P}_1 or \mathcal{P}_2 , which are very useful in the setting of product of experts models (§3) for *training* or *decoding* with such models. Such useful calculations involve constraining one of the structures (e.g., fixing it to an observed structure) and calculating over the possibilities of the other structures (and their scores) a sum or a maximum. If the scores are probabilities, for example, we may wish to calculate $p(x) = \sum_y p(x, y)$ for an observed structure x when training our model, or to decode $\text{argmax}_y p(y | x)$ (as in equation 5). We now describe how to do that mechanically in our setting, assuming that the structure to be fixed originates in \mathcal{P}_1 and the structure to be maximized or summed originates in \mathcal{P}_2 .

Given a structure to be “fixed,” from \mathcal{P}_2 we first encode it as a single proof (i.e., a set of theorems), which will be passed to the constrained PRODUCT program as a set of axioms. Further constraints are added to the WLP: whenever a theorem from \mathcal{P}_2 appears (as part of a new, coupled theorem), a side condition is imposed on that inference rule. For example, in synchronized CKY (Eq. 52 in Fig. 21) would become:

$$\begin{aligned} c_{1 \circ 2}(\mathbf{X}_1, \mathbf{X}_2, \mathbf{I}, \mathbf{K}) \oplus = & \text{binary}_1(\mathbf{X}_1, \mathbf{Y}_1, \mathbf{Z}_1) \otimes \text{binary}_2(\mathbf{X}_2, \mathbf{Y}_2, \mathbf{Z}_2) \otimes \\ & c_{1 \circ 2}(\mathbf{Y}_1, \mathbf{Y}_2, \mathbf{I}, \mathbf{J}) \otimes c_{1 \circ 2}(\mathbf{Z}_1, \mathbf{Z}_2, \mathbf{J}, \mathbf{K}) \\ & \text{whenever } c_2(\mathbf{Y}_2, \mathbf{I}, \mathbf{J}) \wedge c_2(\mathbf{Z}_2, \mathbf{J}, \mathbf{K}) \end{aligned} \quad (53)$$

Depending on the semiring, this change results in summing or maximizing over all possible parse trees using one grammar, while constraining the other grammar to derive a single parse tree.

6.3 Axiom Generalization

Axiom generalization is another way of manipulating products of weighted logic programs in a way that reveals the simple structures underlying a complex structure. Figure 22 is close to the weighted logic program in Fig. 14 that describes the intersection of two finite state machines, but there are two differences. First, we have *not* forced the two symbols to be the same; instead, we wish to interpret \mathbf{A}_1 from the first expert as the transducer’s input symbol and \mathbf{A}_2 as the transducer’s output symbol. Second, we have merged $\text{initial}_1(\mathcal{Q}_1) \otimes \text{initial}_2(\mathcal{Q}_2)$ to the single product predicate $\text{initial}_{1 \circ 2}(\mathcal{Q}_1, \mathcal{Q}_2)$, and likewise for arc . As a first approximation, we can just define $\text{arc}_{1 \circ 2}$ (and, similarly, $\text{initial}_{1 \circ 2}$) by a single rule of this form:

$$\text{arc}_{1 \circ 2}(\mathcal{P}_1, \mathcal{P}_2, \mathcal{Q}_1, \mathcal{Q}_2, \mathbf{A}_1, \mathbf{A}_2) \oplus = \text{arc}_1(\mathcal{P}_1, \mathcal{Q}_1, \mathbf{A}_1) \otimes \text{arc}_2(\mathcal{P}_2, \mathcal{Q}_2, \mathbf{A}_2) \quad (56)$$

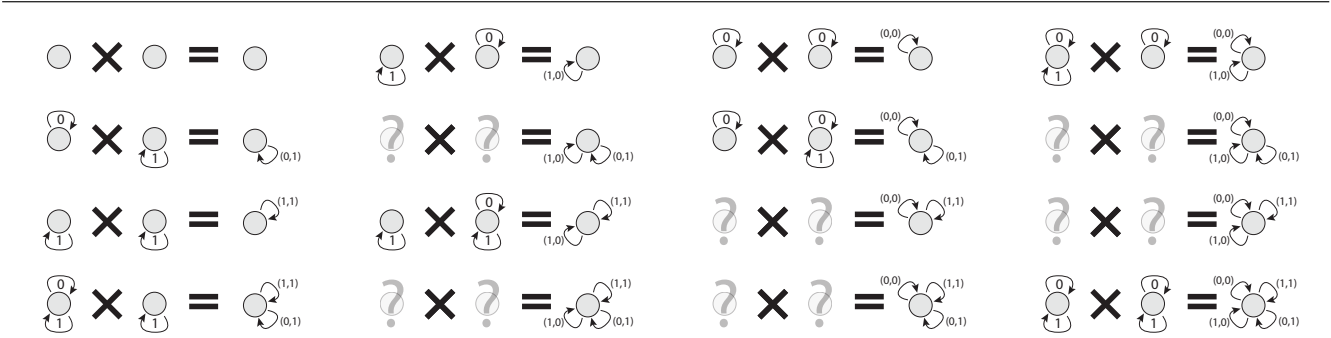


Fig. 24: The sixteen possible single-state transducers with two symbols, and possible factorings for each one—six transducers cannot be factored, and the transducer in the upper-left corner with no transitions can be factored seven different ways.

An example is given on the left-hand-side of Fig. 23. Two finite state *machines*, one with two states (a and b) and one with three states (x, y, and z), are shown - we are interpreting over the Boolean semiring, so each present arc in the figure corresponds to a true-valued *arc* axiom. The **PRODUCT** of these two experts in the manner of Fig. 22 is a single finite state *transducer* with six states.

However, we can only describe a certain subset of finite state transducers as the direct product of finite state machines in this way. If we consider all possible Boolean-valued finite state transducers with two symbols and one state, we have 16 possible transducers, but only 10 that can be “factored” as two independent finite-state machines, as shown in Fig. 24. More generally, one can show that in order for a set of axioms, of the same predicate, to be factored into smaller components, we need to be able to represent its range (meaning, all of the variables settings for that set of axioms) as a Cartesian product of sets, one per each range of a single variable for the axiom.

How, then, can we capture all possible transducers? The obvious solution is what could be called *axiom generalization*, which amounts to removing the requirement of equation 56 that the value of atomic propositions of the form $\text{arc}_{1 \circ 2}$ be the product of an atomic proposition of the form arc_1 and an atomic proposition of the form arc_2 . Instead, if we directly define axioms of the form $\text{arc}_{1 \circ 2}$, we can describe transducers in their full generality. A similar issue arises in synchronous and lexicalized parsing; all possible synchronous and lexicalized parsers cannot be described as the product of two CKY parsers unless we consider the generalized form of axioms.

7 Conclusion

We have described a general framework for dynamic programming algorithms whose solutions correspond to proof values in two mutually constrained weighted logic programs. Our framework includes a program transformation, **PRODUCT**, which combines the two weighted logic programs that compute over two structures into a single weighted logic program for a joint proof. Appropriate constraints, encoded intuitively as variable unification or side conditions in the WLP, are then added manually. The framework naturally captures many existing algorithms.

Acknowledgments

The authors acknowledge helpful comments from three anonymous ICLP reviewers, Jason Eisner, Frank Pfenning, David Smith, and Sylvia Rebolz. This research was supported by an NSF graduate fellowship to the second author and NSF grant IIS-0713265 and an IBM faculty award to the third author.

References

1. D. Chiang. Hierarchical phrase-based translation. *Computational Linguistics*, 32(2):201–228, 2007.

2. J. Cocke and J. T. Schwartz. Programming languages and their compilers: Preliminary notes. Technical report, Courant Institute of Mathematical Sciences, New York University, 1970.
3. S. B. Cohen, R. J. Simmons, and N. A. Smith. Dynamic programming algorithms as products of weighted logic programs. In *Proc. of ICLP*, 2008.
4. S. B. Cohen and N. A. Smith. Joint morphological and syntactic disambiguation. In *Proc. of EMNLP-CoNLL*, 2007.
5. J. Eisner and J. Blatz. Program transformations for optimization of parsing algorithms and other weighted logic programs. In *Proc. of Formal Grammar*, 2007.
6. J. Eisner, E. Goldlust, and N. A. Smith. Dyna: A declarative language for implementing dynamic programs. In *Proc. of ACL* (companion volume), 2004.
7. J. Eisner, E. Goldlust, and N. A. Smith. Compiling Comp Ling: Practical weighted dynamic programming and the Dyna language. In *Proc. of HLT-EMNLP*, 2005.
8. J. Eisner and G. Satta. Efficient parsing for bilexical context-free grammars and head automaton grammars. In *Proc. of ACL*, 1999.
9. P. F. Felzenszwalb and D. McAllester. The generalized A* architecture. *Journal of Artificial Intelligence Research*, 29:153–190, 2007.
10. H. Gaifman. Dependency systems and phrase-structure systems. *Information and Control*, 8, 1965.
11. H. Ganzinger and D. A. McAllester. Logical algorithms. In *Proc. of ICLP*, 2002.
12. J. Goodman. Semiring parsing. *Computational Linguistics*, 25(4):573–605, 1999.
13. S. Greco and C. Zaniolo. Greedy algorithms in Datalog. *Theory Pract. Log. Program.*, 1(4):381–407, 2001.
14. G. E. Hinton. Training products of experts by minimizing contrastive divergence. *Neural Computation*, 14:1771–1800, 2002.
15. J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 1979.
16. L. Huang and D. Chiang. Better k -best parsing. In *Proc. of IWPT*, 2005.
17. D. Klein and C. D. Manning. Fast exact inference with a factored model for natural language parsing. In *Advances in NIPS 15*, 2003.
18. D. Klein and C. D. Manning. Parsing and hypergraphs. *New developments in parsing technology*, pages 351–372, 2004.
19. V. Levenshtein. Binary codes capable of correcting spurious insertions and deletions of ones. *Problems of Information Transmission*, 1:8–17, 1965.
20. Percy Liang, Dan Klein, and Michael Jordan. Agreement-based learning. In J.C. Platt, D. Koller, Y. Singer, and S. Roweis, editors, *Advances in Neural Information Processing Systems 20*, pages 913–920, Cambridge, MA, 2008. MIT Press.
21. C.D. Manning and H. Schütze. *Foundations of Statistical Natural Language Processing*. MIT Press, 1999.
22. D. McAllester. On the complexity analysis of static analyses. In *Proc. of Static Analysis Symposium*, 1999.
23. I. D. Melamed. Multitext grammars and synchronous parsers. In *Proc. of HLT-NAACL*, 2003.
24. M. Mohri. Finite-state transducers in language and speech processing. *Computational Linguistics*, 23(2):269–311, 1997.
25. S. M. Shieber, Y. Schabes, and F. C. N. Pereira. Principles and implementation of deductive parsing. *Journal of Logic Programming*, 24(1–2):3–36, 1995.
26. K. Sikkel. *Parsing Schemata*. Springer-Verlag, 1997.
27. D. A. Smith and N. A. Smith. Bilingual parsing with factored estimation: Using English to parse Korean. In *Proc. of EMNLP*, 2004.
28. C. Sutton and A. McCallum. Piecewise training of undirected models. In *Proc. of UAI*, 2005.
29. R. E. Tarjan. A unified approach to path problems. *Journal of the ACM*, 28(3):577–93, 1981.
30. D. Wu. Stochastic inversion transduction grammars and bilingual parsing of parallel corpora. *Computational Linguistics*, 23(3):377–404, 1997.
31. H. Zhang and D. Gildea. Stochastic lexicalized inversion transduction grammar for alignment. In *Proc. of ACL*, 2005.
32. H. Zhang and D. Gildea. Inducing word alignments with bilexical synchronous trees. In *Proc. of COLING-ACL*, 2006.

