Products of weighted logic programs

SHAY B. COHEN, ROBERT J. SIMMONS and NOAH A. SMITH

School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, USA
(*e-mail:* {scohen,rjsimmon,nasmith}@cs.cmu.edu)

Abstract

Weighted logic programming, a generalization of bottom-up logic programming, is a well-suited 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 real-valued score (often interpreted as a probability) that depends on the real weights 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. In addition, we show how the computation of Kullback–Leibler divergence, an information-theoretic measure, can be interpreted using PRODUCT.

KEYWORDS: weighted logic programming, program transformations, natural language processing

1 Introduction

Weighted logic programming is a technique that can be used to declaratively specify dynamic programming algorithms in a number of fields such as natural language processing (Manning and Schütze 1999) and computational biology (Durbin *et al.* 1998). Weighted logic programming is a generalization of bottom-up logic programming where each proof is assigned a score (or weight) that is a function of the scores of the axioms used in the proof. When these scores are interpreted as probabilities, then the solution to a whole weighted logic program can be interpreted in terms of probabilistic reasoning about unknowns, implying that the weighted logic program implements *probabilistic inference*[1].

---

[1] The word *inference* has a distinct meaning in logic programming (e.g., "inference rule," "valid inference"), and so we will attempt to avoid confusion by using the *probabilistic* modifier whenever we are talking about probabilistic reasoning about unknowns.

Even though weighted logic programming is not limited to probabilistic inference, it is worth detailing their relationship. Let I, A, and P be random variables, where the values of I and A are known and the value of P is not known. Often there is a correspondence where

- I corresponds to a conditional "input," encoded as axioms, known to be true;
- A corresponds to a set of axioms known to be true; and
- P corresponds to a deductive proof of the goal theorem using the axioms.

In the setting of weighted logic programming, there may be many different proofs of the goal given the set of axioms. We must therefore distinguish the weighted logic program from the "world" we are reasoning about in which these many different proofs of the goal correspond to different, mutually exclusive events, each of which has some probability of occurring. Weighted logic programming implements probabilistic inference over the value of the proof random variable P given the values of A and I: the weighted logic program implies a probability distribution $p(P \mid A, I)$, and it can be used to compute different useful quantities related to the distribution.

Previous work on weighted logic programming has shown that certain families of probabilistic models lend themselves extremely well to weighted logic programming as an inference mechanism. In general, weighted logic programming deals with probability distributions over objects with combinatorial structure—paths through graphs, grammatical derivations, and sequence alignments—that are quite useful in computer science applications.

In principle, one can think about combining such distributions with each other, creating distributions over even more complex structures that are related. This paper is about a natural extension to weighted logic programming as probabilistic inference over structures: combining weighted logic programs to perform inference over two or more structures. We describe a program transformation, PRODUCT, that implements joint probabilistic inference via weighted logic programming over two structured variables P_1 and P_2, when (a) each of the two separate structures can be independently reasoned about using weighted logic programming and (b) the joint model factors into a product of two distributions $p(P_1 \mid A_1, I_1)$ and $p(P_2 \mid A_2, I_2)$.[2]

As a program transformation on traditional logic programs, PRODUCT is not novel; it has existed as a compiler transformation for over a decade (Pettorossi and Proietti 1994; Pettorossi 1999). As a way of describing joint probabilistic inference in weighted logic programming, the transformation has been intuitively exploited in designing algorithms for specific applications but has not, to our knowledge, been generalized. The contribution of this paper is a general, intuitive, formal setting for dynamic programming algorithms that process two or more conceptually distinct objects. Indeed, we show that many important dynamic programming algorithms can be derived using simpler "factor" programs and the PRODUCT transformation together with side conditions that capture the relationship between the structures.

---

[2] In the language of probability, this means that P_1 and P_2 are conditionally independent given A_1, A_2, I_1, and I_2.

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

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

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

$$\begin{aligned} \text{initial(a)} &= \text{T} & \text{edge(c, d)} &= \text{T} \\ \text{edge(a, c)} &= \text{T} & \text{edge(d, b)} &= \text{T} \\ \text{edge(a, d)} &= \text{T} & \text{edge(d, c)} &= \text{T} \\ \text{edge(b, b)} &= \text{T} & \text{edge(d, d)} &= \text{T} \\ \text{edge(c, a)} &= \text{T} \end{aligned}$$
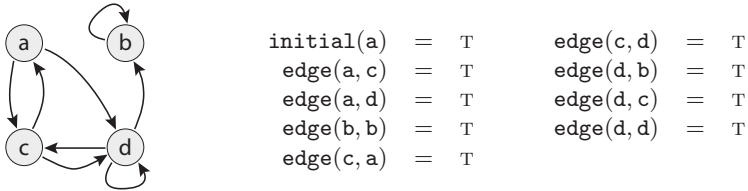
Fig. 2. A directed graph and the corresponding initial database.

The paper is organized as follows. In Section 2, we give an overview of weighted logic programming. In Section 3, we describe products of experts, a concept from machine learning that elucidates the kinds of probabilistic models amenable to our framework. In Section 4, we describe the PRODUCT transformation. In Section 5, we show how several well-known algorithms can be derived using the PRODUCT transformation applied to simpler algorithms. Section 6 presents some variations on the PRODUCT transformation. In Section 7, we show how to use the PRODUCT transformation and a specially designed semiring to calculate important information-theoretic quantities related to probability distributions over proofs.
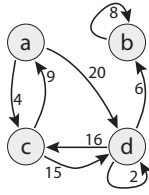
2 Weighted logic programming

To motivate weighted logic programming, we begin with a logic program for single-source connectivity on a directed graph, shown in Figure 1. In the usual bottom-up interpretation of this program, an initial database (i.e., set of axioms) would describe the edge relation and one (or more) starting vertices as axioms of the form `initial(a)` for some a. Repeated forward inference can then be applied on the rules in Figure 1 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.

Weighted logic programming generalizes traditional logic programming. In traditional logic programming, a proof is a tree of valid (deductive) inferences from axioms, and a valid atomic proposition is one that has at least one proof. In weighted logic programming, we generalize this notion: axioms, proofs, and atomic propositions are said to "have values" rather than just "be valid." Traditional logic programs can be understood as weighted logic programs with Boolean values: axioms all have the value "true," as do all valid propositions. The single-source connectivity program would describe the graph in Figure 2 by assigning T as the value of all the existing edges and the proposition `initial(a)`.
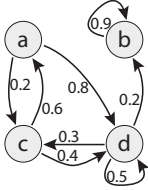
2.1 Non-Boolean programs

With weighted logic programming, the axioms and propositions can be understood as having non-Boolean values. In Figure 3, each axiom of the form `edge(X, Y)` is given

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.

a value corresponding to the cost associated with that 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 the axioms at its leaves and take the value of a proposition to be the *minimum* score over all possible proofs, then the program from Figure 1 gives a declarative specification of the *single-source shortest path* problem. Multiple uses of an axiom in a proof are meaningful: if a proof includes the `edge(d, d)` axiom once, it corresponds to a single traversal of the loop from d to d and adds a cost of 2, and if a proof includes the axiom twice, it corresponds to two distinct traversals and adds a cost of 4.

We replace the connectives :- (disjunction) and , (conjunction) with min = and +, respectively, and interpret the weighted logic program over the nonnegative numbers. With a specific execution model, the result is Dijkstra's single-source shortest path algorithm.

In addition to the cost-minimization interpretation in Figure 3, we can interpret weights on edges as *probabilities* and restate the problem in terms of probability *maximization*. In Figure 4, the outgoing edges from each vertex sum to at most 1. If we assign the missing 0.1 probability from vertex b to a "stopping" event—either implicitly or explicitly by modifying the axioms—then each vertex's outgoing edges sum to exactly 1 and the graph can be seen as a Markov model or probabilistic finite-state network over which random walks are well defined. If we replace the connectives :- (disjunction) and , (conjunction) with max = and ×, then the value of `reachable(X)` for any X is the probability of the most likely path from a to X. For instance, `reachable(a)` ends up with the value 1, and `reachable(b)` ends up with value 0.16, corresponding to the path a → d → b, whose weight is (value of `initial(a)` × value of `edge(a, d)` × value of `edge(d, b)`).

If we keep the initial database from Figure 4 but change our operators from max = and × to += and ×, 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

$$\text{reachable(Q)} \quad \oplus= \quad \text{initial(Q)}. \tag{3}$$
$$\text{reachable(Q)} \quad \oplus= \quad \text{reachable(P)} \otimes \text{edge(P,Q)}. \tag{4}$$

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

as the "path sum" (Tarjan 1981). The path sum for `reachable(b)`, for instance, is 10—this is not a probability, but rather an infinite sum of probabilities of many paths, some of which are prefixes of each other[3].

These three related weighted logic programs are useful generalizations of the reachability logic program in Figure 1. Figure 5 gives a generic representation of all four algorithms in the Dyna language (Eisner *et al.* 2005). The key difference among them is the *semiring* in which we interpret the weights. 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. We require, following Goodman (1999), 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 infinitely many 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.

In our running example, reachability uses the Boolean semiring $\langle \{\text{T}, \text{F}\}, \vee, \wedge, \text{F}, \text{T} \rangle$, single-source shortest path uses $\langle \mathbb{R}_{\geqslant 0} \cup \{\infty\}, \min, +, \infty, 0 \rangle$, the most-probable-path variant uses $\langle [0,1], \max, \times, 0, 1 \rangle$, and the probabilistic path-sum variant uses the so-called "real" semiring $\langle \mathbb{R}_{\geqslant 0} \cup \{\infty\}, +, \times, 0, 1 \rangle$.

Weighted logic programming was developed primarily within the computational linguistics community. Building upon the observations of Shieber *et al.* (1995) and Sikkel (1997) that many parsing algorithms for nondeterministic grammars could be represented as deductive logic programs, Goodman (1999) showed that the structure of the parsing algorithms was amenable to interpretation on a number of semirings. McAllester (2002) additionally showed that this representation facilitates reasoning about asymptotic complexity. Other developments include a connection between weighted logic programs and hypergraphs (Klein and Manning 2004), optimal A* search for maximizing programs (Felzenszwalb and McAllester 2007), semiring-general agenda-based implementations (Eisner *et al.* 2005), improved k-best

---

[3] Clearly, "10" is not a meaningful probability, but that is a result of the loop from `b` to `b` with probability 0.9—in fact, one informal way of looking at the result is simply to observe that $10 = 1 + 0.9 + (0.9)^2 + (0.9)^3 + \ldots$, corresponding to proofs of `reachable(b)` that include `edge(b,b)` zero, one, two, three, ... times. If we added an axiom `edge(b,final)` with weight 0.1 representing the 10% probability of stopping at any step in state `b`, then the path sum for `reachable(final)` would be $10 \times 0.1 = 1$, which is a reasonable probability that corresponds to the fact that a graph traversal can be arbitrarily long but has a 100% chance of eventually reaching `b` and then stopping.

algorithms (Huang and Chiang 2005), and program transformations to improve efficiency (Eisner and Blatz 2007).

2.2 Formal definition

Eisner and Blatz (2007) describe the semantics of weighted logic programs in detail; we summarize their discussion in this section and point the reader to that paper for further detail. A weighted logic program is a set of *Horn equations* describing a set of declarative, usually recursive equations over an abstract semiring. Horn equations, which we will refer to by the shorter and more traditional term *rules*, take the form

$$\mathtt{consequent}(\mathbf{U}) \oplus= \mathtt{antecedent}_1(\mathbf{W_1}) \otimes \cdots \otimes \mathtt{antecedent}_n(\mathbf{W_n}).$$

Here, $\mathbf{U}$ and $\mathbf{W_i}$ are sequences of terms that include free variables. If the variables in $\mathbf{U}$ are a subset of the variables in $\mathbf{W_1}, \ldots, \mathbf{W_n}$ for every rule, then the program is *range restricted* or *fully grounded*.

We can also give rules *side conditions*. Side conditions are additional constraints that are added to a rule to remove certain proofs from consideration. For example, side conditions could allow us to modify rule (4) in Figure 5 to disallow self-loops and only allow traversal of an edge when there was another edge in the opposite direction:

$$\mathtt{reachable}(\mathtt{Q}) \oplus= \mathtt{reachable}(\mathtt{P}) \otimes \mathtt{edge}(\mathtt{P},\mathtt{Q}) \text{ if } \mathtt{edge}(\mathtt{Q},\mathtt{P}) \wedge \mathtt{Q} \neq \mathtt{P}. \qquad (5)$$

Side conditions do not change the value of any individual proof; they only filter out any proof that does not satisfy the side conditions. In this paper, we use mostly side conditions that enforce equality between variables. For a more thorough treatment of side conditions, see Goodman (1999) or Eisner and Blatz (2007).

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. The meaning of a weighted logic program is determined by the rules together with a set of fully grounded axioms (or *facts* in the Prolog setting). Each axiom is assigned a value from the set $\mathbb{K}$ that is interpreted as a weight or score.

A common idiom in weighted logic programming is to specify the query as a distinguished predicate `goal` that takes no arguments. A computationally uninteresting (because there are no intermediate computation steps) but otherwise legitimate way to present a weighted logic program is as a single rule of the form

$$\mathtt{goal} \oplus= \mathtt{axiom}_1(\mathbf{W_1}) \otimes \cdots \otimes \mathtt{axiom}_n(\mathbf{W_n}).$$

In this degenerate case, each distinct way of satisfying the premises using axioms in the database would correspond to a distinct proof of `goal`. The score of each proof would be given by the semiring product of the scores of the axioms, and the value of `goal` would be determined by the semiring sum of the scores of all the proofs.

In the general case, 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
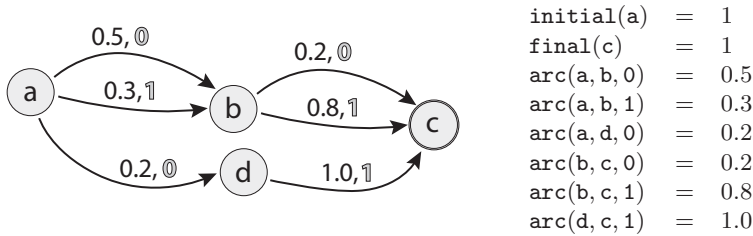
Fig. 6. A probabilistic FSA and the corresponding initial database.

of products, exploiting distributivity and shared substructure for efficiency. This inherent notion of shared substructure means that weighted logic programming can give straightforward declarative specifications for problems that are typically solved by dynamic programming. The Dyna programming language implements a particular dynamic programming strategy for implementing these declarative specifications (Eisner *et al.* 2005), though the agenda algorithm that it implements may potentially have significantly different behavior, in terms of time and space complexity, than other dynamic programming algorithms that meet the same specification.

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 Weighted logic programs and probabilistic reasoning

In this section, we will return focus to the probabilistic interpretation of weighted logic programs that we first described in the Introduction. In Section 3.1, we will describe in more detail how the results of weighted logic programs are interpreted as probabilities—readers with a background in statistics and machine learning can probably skip or skim this section. In Section 3.2, we will introduce the notion of a *product of experts* that motivates the PRODUCT transformation.

Our running example for this section is a probabilistic finite-state automaton (FSA) over the alphabet $\{0, 1\}$, shown in Figure 6. The most probable path through the graph is the one that recognizes the string "01" by going through states a, b, and c, and that the probability of this path is 0.4. Other than the labels on the edges, this is the same setup used in the graph-reachability example from Figure 4. The edge predicate from the previous section is now called arc and has been augmented to carry a third argument representing an output character.

3.1 The probabilistic interpretation of weighted logic programming

Recall from the Introduction that, in the context of weighted logic programming, we have random variables I, A, and P, where

- I corresponds to a set of conditional "input" axioms known to be true,
- A corresponds to a set of axioms known to be true, and
- P corresponds to a deductive proof of the goal theorem using the axioms.

$$\text{goal} \quad \oplus= \quad \text{path(Q)} \otimes \text{final(Q)}. \tag{6}$$

$$\text{path(Q)} \quad \oplus= \quad \text{initial(Q)}. \tag{7}$$

$$\text{path(Q)} \quad \oplus= \quad \text{path(P)} \otimes \text{arc(P,Q,A)}. \tag{8}$$

Fig. 7. The weighted logic program for weighted FSA reachability.

$$\text{goal} \quad \oplus= \quad \text{path(Q,I)} \otimes \text{final(Q)} \otimes \text{length(I)}. \tag{9}$$

$$\text{path(Q,0)} \quad \oplus= \quad \text{initial(Q)}. \tag{10}$$

$$\text{path(Q,I)} \quad \oplus= \quad \text{path(P,I}-1) \otimes \text{arc(P,Q,A)} \otimes \text{string(I,A)}. \tag{11}$$

Fig. 8. The weighted logic program for weighted FSA recognition.

In this case, I corresponds to one of the various possible sentences recognized by the FSA (i.e., 00, 01, 10, and 11); A corresponds to a particular directed graph with weighted edges, encoded by a set of axioms; and P corresponds to an individual proof/path through the graph. In Figure 7, which is the straightforward adaptation of the reachability program in Figure 5 to labeled edges, the value of goal in the most-probable-path semiring is $\max_{proof} p(P = proof, I = sentence \mid A = graph)$— the value of the most probable path emitting *any* possible sentence I.

In order to talk about the input sentences I, we first add a set of axioms that describe I. If we are interested in the sentence "01," we would add axioms string(1, 0), string(2, 1), and length(2), whereas if we are interested in the sentence "hey," we would add axioms string(1, h), string(2, e), string(3, y), and length(3). These axioms are all given the value 1 (the multiplicative unit of the semiring), and so they could equivalently be treated as side conditions. With these new axioms, we modify Figure 7 to obtain Figure 8, a weighted logic program that limits the proofs/paths to the ones which represent recognition of the input string I[4].

Now, Figure 8 interpreted over the most-probable-path semiring *does* allow us to find the proof that, given the edge weights and a specific sentence, maximizes $p(P = proof \mid I = sentence, A = graph)$. It does *not*, however, give us $p(P = proof \mid I = sentence, A = graph)$, but rather $p(P = proof, I = sentence \mid A = graph)$, the *joint* probability of a path and a sentence given the weights on the edges.

Concretely, in our running example, there are five possible proofs of goal in Figure 7 whose probabilities sum to 1, but there are only two parses that also recognize the string "01," which are $\text{a}_0\text{b}_1\text{c}$ with weight 0.4 and $\text{a}_0\text{d}_1\text{c}$ with weight 0.2—the route through b is twice as likely. The value of goal in Figure 8 interpreted in the most-probable-path semiring would be 0.4 (the joint probability of obtaining the proof $\text{a}_0\text{b}_1\text{c}$ and of recognizing the string "01") not $0.\overline{6}$ (the probability of the proof $\text{a}_0\text{b}_1\text{c}$ given the sentence "01"). In other words, we have $p(P = \text{a}_0\text{b}_1\text{c}, I = 01 \mid A = \text{Fig. 6}) = 0.4$, $p(P = \text{a}_0\text{d}_1\text{c}, I = 01 \mid A = \text{Fig. 6}) = 0.2$, $p(P = \text{a}_0\text{b}_1\text{c} \mid I = 01, A = \text{Fig. 6}) = 0.\overline{6}$.

---

[4] Rule (11) in this figure uses "$I - 1$" in a premise: we assume that our formalism includes natural numbers that support increment/decrement operations, and our simple uses can be understood as syntactic shorthand for either structured terms (z, s(z), etc.) or the use of primitive side conditions such as inc(I, I′).

The solution for correctly discovering the conditional probability lies in the fact that the joint and conditional probabilities are related in the following way:

$$p(P \mid A, I) = \frac{p(P, I \mid A)}{p(I \mid A)}.$$

This, combined with the knowledge that the *marginal* probability $p(I \mid A)$ is the result of evaluating Figure 8 over the path-sum semiring (i.e., $\langle \mathbb{R}_{\geqslant 0} \cup \{\infty\}, +, \times, 0, 1 \rangle$), allows us to correctly calculate not only the most probable proof P of a given sentence but also the probability of that proof *given* the sentence. The marginal probability in our running example is 0.6, and $0.4/0.6 = 0.\overline{6}$, which is the desired result.

To restate this in a way that is more notationally consistent with other work in machine learning, we first take the weighted axioms A as implicit. Then, instead of proofs P, we talk about values y for a random variable Y drawn out of a domain $\mathcal{Y}$ (the space of possible structures, which in our setting corresponds to the space of possible proofs), and instead of inputs I, we talk about values x for a random variable X drawn out of a domain $\mathcal{X}$ (the space of all possible inputs).

Then, to predict the most likely observed value for y, denoted $\hat{y}$, we have the following formula:

$$\hat{y} = \underset{y \in \mathcal{Y}}{\operatorname{argmax}}\, p(Y = y \mid X = x) = \underset{y \in \mathcal{Y}}{\operatorname{argmax}}\, \frac{p(Y = y, X = x)}{p(X = x)}. \tag{12}$$

Because $p(X = x)$ does not depend on y, if we only want to know $\hat{y}$ it suffices to find y that maximizes $p(Y = y, X = x)$ (which was written as $p(P = proof, I = sentence \mid A = axioms)$ above). One way to do this is to execute a weighted logic program in the most-probable-path semiring.

3.2 Products of experts

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

$$p(Y = y \mid X = x) \propto p_1(Y = y \mid X = x) \times \cdots \times p_n(Y = y \mid X = x), \tag{13}$$

where $\propto$ signifies "proportional to" and suppresses the means by which the probability distribution is renormalized to sum to 1. This kind of model is called a *product of experts* (Hinton 2002). Intuitively, the probability of an event under p can only be relatively large if "all the experts concur," i.e., if the probability is large under each of p_i. Any single expert can make an event arbitrarily unlikely (even impossible) by giving it very low probability, and the solution to equation (12) for a product of experts model will be $y \in \mathcal{Y}$ (here, a proof) least objectionable to all experts.

The attraction of such probability distributions is that they modularize complex systems (Klein and Manning 2003; Liang *et al.* 2008). They can also offer computational advantages when solving equation (12) (Chiang 2007). Further, the expert factors can often be trained (i.e., estimated from data) separately, speeding up expensive but powerful machine learning methods (Smith and Smith 2004; Smith *et al.* 2005; Sutton and McCallum 2005; Cohen and Smith 2007).

To the best of our knowledge, there has been no attempt to formalize the following intuitive idea about products of experts: algorithms for reasoning about mutually constrained product proof values should resemble the individual algorithms for each of the two separate "factor" proofs' values. Our formalization is intended to aid in algorithm development as new kinds of complex 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," analogous to the product of expert probability models that are derived from factor expert probability models.

To relate this observation to the running example from this section, imagine we created two copies of Figure 8 which operated over the same sentence (as described by string and length predicates) but which had different predicates and axioms $goal_1$, $path_1$, $final_1$, $initial_1$, and arc_1 (and likewise $goal_2$, $path_2$, etc.). Consider a combined goal predicate $goal_{1 \bullet 2}$ defined by the rule

$$goal_{1 \bullet 2} \oplus = goal_1 \otimes goal_2. \tag{14}$$

Now we have two experts ($goal_1$ and $goal_2$), and we literally take the (semiring) product of them, but this is still not quite the "product of experts," because the proofs of the goals are allowed to be independent. In other words, what we have is the following:

$$p(Y_1 = y_1, Y_2 = y_2 \mid X = x) \propto p_1(Y_1 = y_1 \mid X = x) \times p_2(Y_2 = y_2 \mid X = x).$$

The PRODUCT transformation is a meaning-preserving transformation on weighted logic programs that exposes the joint structure in such a way that—depending on our domain-specific understanding of what it means for the two proofs y_1 and y_2 to match—allows us to add constraints that result in a weighted logic program that forces the structures to match, as required by the specification in equation (13).

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.

The PRODUCT transformation can be seen as an instance of the *tupling* program transformation combined with an *unfold/fold* transformation (Pettorossi and Proietti 1994; Pettorossi 1999) that preserves the meaning of programs. However, we are interested in this transformation not for reasons of efficiency, but because it has the effect of exposing the shared structure of the two individual programs in such a way that, by the manual addition of constraints, we can force the two original programs to optimize over the *same* structures, thereby implementing optimization over the product of experts as described in the previous section. The addition of constraints requires an understanding of the problem at hand, as shown in Section 5 by presenting a number of examples.

$$\text{reachable}_1(\mathtt{Q}_1) \quad \oplus= \quad \text{initial}_1(\mathtt{Q}_1). \tag{15}$$

$$\text{reachable}_1(\mathtt{Q}_1) \quad \oplus= \quad \text{reachable}_1(\mathtt{P}_1) \otimes \text{edge}_1(\mathtt{P}_1, \mathtt{Q}_1). \tag{16}$$

$$\text{reachable}_2(\mathtt{Q}_2) \quad \oplus= \quad \text{initial}_2(\mathtt{Q}_2). \tag{17}$$

$$\text{reachable}_2(\mathtt{Q}_2) \quad \oplus= \quad \text{reachable}_2(\mathtt{P}_2) \otimes \text{edge}_2(\mathtt{P}_2, \mathtt{Q}_2). \tag{18}$$

Fig. 9. Two identical experts for generalized graph reachability, duplicates of the program in Figure 5.

$$\text{reachable}_{1\bullet2}(\mathtt{Q}_1, \mathtt{Q}_2) \quad \oplus= \quad \text{initial}_1(\mathtt{Q}_1) \otimes \text{initial}_2(\mathtt{Q}_2). \tag{19}$$

$$\text{reachable}_{1\bullet2}(\mathtt{Q}_1, \mathtt{Q}_2) \quad \oplus= \quad \text{reachable}_2(\mathtt{P}_2) \otimes \text{edge}_2(\mathtt{P}_2, \mathtt{Q}_2) \otimes \text{initial}_1(\mathtt{Q}_1). \tag{20}$$

$$\text{reachable}_{1\bullet2}(\mathtt{Q}_1, \mathtt{Q}_2) \quad \oplus= \quad \text{reachable}_1(\mathtt{P}_1) \otimes \text{edge}_1(\mathtt{P}_1, \mathtt{Q}_1) \otimes \text{initial}_2(\mathtt{Q}_2). \tag{21}$$

$$\text{reachable}_{1\bullet2}(\mathtt{Q}_1, \mathtt{Q}_2) \quad \oplus= \quad \text{reachable}_{1\bullet2}(\mathtt{P}_1, \mathtt{P}_2) \otimes \text{edge}_1(\mathtt{P}_1, \mathtt{Q}_1) \otimes \text{edge}_2(\mathtt{P}_2, \mathtt{Q}_2). \tag{22}$$

Fig. 10. Four rules that, in addition to the rules in Figure 9, give the product of the two experts defined by the reachable_1 and reachable_2 predicates.

4.1 The product of graph-reachability experts

Figure 9 defines two experts, copies of the graph-reachability program from Figure 5. We are interested in a new predicate $\text{reachable}_{1\bullet2}(\mathtt{Q}_1, \mathtt{Q}_2)$, which for any particular $\mathtt{Q}_1$ and $\mathtt{Q}_2$ should be equal to the product of $\text{reachable}_1(\mathtt{Q}_1)$ and $\text{reachable}_2(\mathtt{Q}_2)$. Just as we did in our thought experiment with $\text{goal}_{1\bullet2}$ in the previous section, we could define the predicate by adding the following rule to the program in Figure 9:

$$\text{reachable}_{1\bullet2}(\mathtt{Q}_1, \mathtt{Q}_2) \oplus= \text{reachable}_1(\mathtt{Q}_1) \otimes \text{reachable}_2(\mathtt{Q}_2).$$

This program is a bit simplistic; however, it merely describes calculating the experts independently and then combining them at the end.

The predicate $\text{reachable}_{1\bullet2}$ can alternatively be calculated by adding the following four rules to Figure 9:

$$\text{reachable}_{1\bullet2}(\mathtt{Q}_1, \mathtt{Q}_2) \oplus= \text{initial}_1(\mathtt{Q}_1) \otimes \text{initial}_2(\mathtt{Q}_2),$$
$$\text{reachable}_{1\bullet2}(\mathtt{Q}_1, \mathtt{Q}_2) \oplus= \text{initial}_1(\mathtt{Q}_1) \otimes \text{reachable}_2(\mathtt{P}_2) \otimes \text{edge}_2(\mathtt{P}_2, \mathtt{Q}_2),$$
$$\text{reachable}_{1\bullet2}(\mathtt{Q}_1, \mathtt{Q}_2) \oplus= \text{reachable}_1(\mathtt{P}_1) \otimes \text{edge}_1(\mathtt{P}_1, \mathtt{Q}_1) \otimes \text{initial}_2(\mathtt{Q}_2),$$
$$\text{reachable}_{1\bullet2}(\mathtt{Q}_1, \mathtt{Q}_2) \oplus= \text{reachable}_1(\mathtt{P}_1) \otimes \text{edge}_1(\mathtt{P}_1, \mathtt{Q}_1) \otimes$$
$$\text{reachable}_2(\mathtt{P}_2) \otimes \text{edge}_2(\mathtt{P}_2, \mathtt{Q}_2).$$

This step is described as an *unfold* by Pettorossi (1999). This unfold can then be followed by a *fold*: because $\text{reachable}_{1\bullet2}(\mathtt{Q}_1, \mathtt{Q}_2)$ was defined above to be the product of $\text{reachable}_1(\mathtt{Q}_1)$ and $\text{reachable}_2(\mathtt{Q}_2)$, we can replace each instance of the two premises $\text{reachable}_1(\mathtt{Q}_1)$ and $\text{reachable}_2(\mathtt{Q}_2)$ with the single premise $\text{reachable}_{1\bullet2}(\mathtt{Q}_1, \mathtt{Q}_2)$.

The new rules that result from this replacement can be seen in Figure 10.

4.2 The PRODUCT transformation

The PRODUCT program transformation is shown in Figure 11. For each desired product of experts, where one expert, the predicate p, is defined by n rules and the

Input: A logic program $\mathcal{P}$ and a set $\mathcal{S}$ of pairs of predicates $(\mathtt{p}, \mathtt{q})$.
Output: A program $\mathcal{P}'$ that extends $\mathcal{P}$, additionally computing the product predicate
$\quad$ $\mathtt{p}{\bullet}\mathtt{q}$ for every pair $(\mathtt{p}, \mathtt{q}) \in \mathcal{S}$ in the input.
$\quad$ 1. $\mathcal{P}' \leftarrow \mathcal{P}$
$\quad$ 2. **for all** pairs $(\mathtt{p}^{c}\mathtt{q})$ in $\mathcal{S}$ **do**
$\quad$ 3. $\quad$ **for all** rules in $\mathcal{P}$, of the form $\mathtt{p}(\mathbf{W}) \oplus{=} A_1 \otimes \cdots \otimes A_n$ **do**
$\quad$ 4. $\quad\quad$ **for all** rules in $\mathcal{P}$, of the form $\mathtt{q}(\mathbf{X}) \oplus{=} B_1 \otimes \cdots \otimes B_m$ **do**
$\quad$ 5. $\quad\quad\quad$ **let** $r \leftarrow [\mathtt{p}{\bullet}\mathtt{q}(\mathbf{W}, \mathbf{X}) \oplus{=} A_1 \otimes \cdots \otimes A_n \otimes B_1 \otimes \cdots \otimes B_m]$
$\quad$ 6. $\quad\quad\quad$ **for all** pairs $(\mathtt{s}(\mathbf{Y}), \mathtt{t}(\mathbf{Z}))$ of antecedents in r such that $(\mathtt{s}, \mathtt{t}) \in \mathcal{S}$ **do**
$\quad$ 7. $\quad\quad\quad\quad$ remove the antecedents $\mathtt{s}(\mathbf{Y})$ and $\mathtt{t}(\mathbf{Z})$ from r
$\quad$ 8. $\quad\quad\quad\quad$ insert the antecedent $\mathtt{s}{\bullet}\mathtt{t}(\mathbf{Y}, \mathbf{Z})$ into r
$\quad$ 9. $\quad\quad\quad$ **end for**
$\quad$ 10. $\quad\quad\quad$ add r to $\mathcal{P}'$
$\quad$ 11. $\quad\quad$ **end for**
$\quad$ 12. $\quad$ **end for**
$\quad$ 13. **end for**
$\quad$ 14. **return** $\mathcal{P}'$

Fig. 11. Algorithmic specification of the PRODUCT transformation.

other expert $\mathtt{q}$ by m rules, the transformation defines the product of experts for $\mathtt{p}{\bullet}\mathtt{q}$ with $n \times m$ new rules, the cross product of inference rules from the first and second experts. The value of a coupled proposition $\mathtt{p}{\bullet}\mathtt{q}$ in $\mathcal{P}'$ will be equal to the semiring product of $\mathtt{p}$'s value and $\mathtt{q}$'s value in $\mathcal{P}$ (or, equivalently, in $\mathcal{P}'$).

Note that lines 6–8 are nondeterministic under certain circumstances, because if the antecedent of the combined program is $\mathtt{a}(\mathtt{X}) \otimes \mathtt{a}(\mathtt{Y}) \otimes \mathtt{b}(\mathtt{Z})$ and the algorithm is computing the product of $\mathtt{a}$ and $\mathtt{b}$, then the resulting antecedent could be either $\mathtt{a}{\bullet}\mathtt{b}(\mathtt{X}, \mathtt{Z}) \otimes \mathtt{a}(\mathtt{Y})$ or $\mathtt{a}{\bullet}\mathtt{b}(\mathtt{Y}, \mathtt{Z}) \otimes \mathtt{a}(\mathtt{X})$. This nondeterminism usually does not arise, and when it does, as in Section 5.2, there is usually an obvious preference.

The PRODUCT transformation is essentially *meaning preserving*: if the program $\mathcal{P}'$ is the result of the PRODUCT transformation on $\mathcal{P}$, then the following is true:

- Any ground instance $\mathtt{p}(\mathbf{X})$ that is given a value in $\mathcal{P}$ is given the same value in $\mathcal{P}'$. This is immediately apparent because the program $\mathcal{P}'$ is stratified: none of the new rules are ever used to compute values of the form $\mathtt{p}(\mathbf{X})$, so their value is identical to their value in $\mathcal{P}$.
- Any ground instance $\mathtt{p}{\bullet}\mathtt{q}(\mathbf{X}, \mathbf{Y})$ in $\mathcal{P}'$ has the same value as $\mathtt{p}(\mathbf{X}) \otimes \mathtt{q}(\mathbf{Y})$. This is the result of the following theorem:

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 $\mathcal{P}'$, for every $(\mathtt{p}, \mathtt{q}) \in \mathcal{S}$, the value $\mathtt{p}{\bullet}\mathtt{q}(\mathbf{X}, \mathbf{Y})$ in $\mathcal{P}'$ is $\mathtt{p}(\mathbf{X}) \otimes \mathtt{q}(\mathbf{Y})$.

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

of proofs 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

$$p(\mathbf{X}) \oplus= a_1(\mathbf{X}_1) \otimes \cdots \otimes a_n(\mathbf{X}_n) \tag{23}$$

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

$$q(\mathbf{Y}) \oplus= b_1(\mathbf{Y}_1) \otimes \cdots \otimes b_m(\mathbf{Y}_m). \tag{24}$$

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

$$p{\bullet}q(\mathbf{X}, \mathbf{Y}) \oplus= c_1(\mathbf{W}_1) \otimes \cdots \otimes c_p(\mathbf{W}_p), \tag{25}$$

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

We resolve each case as follows:

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

We have shown that there is a proof for each antecedent $c_i(\mathbf{W}_i)$ in rule (25), and therefore there is a proof of $p{\bullet}q(\mathbf{X}, \mathbf{Y})$. Furthermore, the value of $p{\bullet}q(\mathbf{X}, \mathbf{Y})$ is indeed $p(\mathbf{X}) \otimes q(\mathbf{Y})$, as concluded trivially from the induction steps.

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

4.3 *From* PRODUCT *to a product of experts*

The output of the PRODUCT transformation is a starting point for describing dynamic programming algorithms that perform two 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 Figure 9 and four from Figure 10. Two examples of constrained product programs are given in Figures 12–14. In the first example in Figure 12, the only change is that all but two rules have been removed from the program in Figures 9 and 10. Whereas in the original product program reachable$_{1\bullet2}(\mathtt{Q}_1, \mathtt{Q}_2)$ corresponded to the product of the weight of the best path

$$\texttt{reachable}_{1\bullet2}(\texttt{Q}_1,\texttt{Q}_2) \quad \oplus= \quad \texttt{initial}_1(\texttt{Q}_1) \otimes \texttt{initial}_2(\texttt{Q}_2). \tag{26}$$

$$\texttt{reachable}_{1\bullet2}(\texttt{Q}_1,\texttt{Q}_2) \quad \oplus= \quad \texttt{reachable}_{1\bullet2}(\texttt{P}_1,\texttt{P}_2) \otimes \texttt{edge}_1(\texttt{P}_1,\texttt{Q}_1) \otimes \texttt{edge}_2(\texttt{P}_2,\texttt{Q}_2). \tag{27}$$

Fig. 12. By removing all but these two rules from the product of experts in Figure 10, we constrain both paths to have the same number of steps.

$$\texttt{reachable}_{1\bullet2}(\texttt{Q}_1,\texttt{Q}_2) \oplus= \texttt{initial}_1(\texttt{Q}_1) \otimes \texttt{initial}_2(\texttt{Q}_2) \boxed{\text{ if } \texttt{Q}_1 = \texttt{Q}_2.} \tag{28}$$

$$\texttt{reachable}_{1\bullet2}(\texttt{Q}_1,\texttt{Q}_2) \oplus= \texttt{reachable}_{1\bullet2}(\texttt{P}_1,\texttt{P}_2) \otimes \texttt{edge}_1(\texttt{P}_1,\texttt{Q}_1) \otimes \texttt{edge}_2(\texttt{P}_2,\texttt{Q}_2) \boxed{\text{ if } \texttt{Q}_1 = \texttt{Q}_2.}$$

Fig. 13. By further constraining the program in Figure 12 to demand that $\texttt{Q}_1 = \texttt{Q}_2$ at all points, we constrain both paths to be identical.

$$\texttt{reachable}_{1\bullet2}(\texttt{Q}) \quad \oplus= \quad \texttt{initial}_1(\texttt{Q}) \otimes \texttt{initial}_2(\texttt{Q}). \tag{29}$$

$$\texttt{reachable}_{1\bullet2}(\texttt{Q}) \quad \oplus= \quad \texttt{reachable}_{1\bullet2}(\texttt{P}) \otimes \texttt{edge}_1(\texttt{P},\texttt{Q}) \otimes \texttt{edge}_2(\texttt{P},\texttt{Q}). \tag{30}$$

Fig. 14. We can simplify Figure 13 by internalizing the side condition and giving $\texttt{reachable}_{1\bullet2}$ only one argument.

from the initial state of graph one to $\texttt{Q}_1$ and the weight of the best path from the initial state of graph two to $\texttt{Q}_2$, the new program computes the best paths from the two origins to the two destinations with the additional requirement that the paths be of 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 visit exactly the same sequence of vertices, then we can further constrain the program to only define $\texttt{reachable}_{1\bullet2}(\texttt{Q}_1,\texttt{Q}_2)$ where $\texttt{Q}_1 = \texttt{Q}_2$, as shown in Figure 13. After adding this side condition, it is no longer necessary for $\texttt{reachable}_{1\bullet2}$ to have two arguments that are always the same, so we can simply further as shown in Figure 14. For simplicity's sake, we will usually collapse arguments that have been forced by equality constraints to agree.

The choice of paired predicates $\mathscr{S}$ is important for the final weighted logic program that PRODUCT returns, and it also limits the way we can add constraints to derive a new weighted logic program. Future research might consider a machine learning setting for automatically deriving $\mathscr{S}$ from data, to minimize some cost (e.g., observed runtime). When PRODUCT is applied on two copies of the same weighted logic program (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.

5 Examples

In this section, we give several examples of constructing weighted logic programs as constrained products of simpler weighted logic programs.

5.1 Finite-state algorithms

We have already encountered weighted finite-state automata (WFSAs) in Section 3.1. Like WFSAs, weighted finite-state transducers (WFSTs) are a generalization of the graph-reachability problem: in WFSAs, the edges are augmented with a symbol and represented as `arc(P,Q,A)`, whereas in WFSTs, the edges are augmented with a pair of input-output symbols and represented as `arc(P,Q,A,B)`. Weighted finite-state machines are widely used in speech and language processing (Mohri 1997; Pereira and Riley 1997). They are used to compactly encode many competing string hypotheses, for example, in speech recognition, translation, and morphological (word-structure) disambiguation. Many sequence labeling and segmentation methods can also be seen as weighted finite-state models.

5.1.1 Weighted finite-state automata

Our starting point for WFSAs will be the weighted logic program for WFSAs described in Figure 7, which is usually interpreted as a probabilistic automaton in the most-probable-path semiring (i.e., $\langle [0,1], \max, \times, 0, 1 \rangle$). If the PRODUCT of that algorithm with itself is taken, we can follow a series of steps similar to the ones described in Section 4.3. First, we remove rules that would allow the two WFSAs to consider different prefixes, and then we add a constraint to rule (33) that requires the two paths' symbols to be identical. The result is a WFSA describing the *(weighted) intersection* of the two WFSAs. The intersection of two WFSAs is itself a WFSA, though it is a WFSA where states are described by two terms—Q_1 and Q_2 in $\texttt{path}_{1 \bullet 2}(Q_1, Q_2)$—instead of a single term.

Weighted intersection generalizes intersection and has a number of uses. For instance, consider an FSA that is "probabilistic" but only accepts the single string "01" because the transitions are all deterministic and have probability 1:
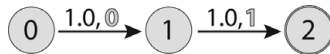
If we consider the program in Figure 15 with axioms describing the above FSA and the probabilistic FSA given in Figure 6, then the resulting program is functionally equivalent to the weighted logic program in Figure 8 describing a WFSA specialized to a particular string. Alternatively, if we consider the program in Figure 15 with axioms describing the probabilistic FSA in Figure 6 and the following single-state probabilistic FSA, the result will be a probabilistic FSA biased toward edges with the "1" symbol and against edges with the "0" symbol.
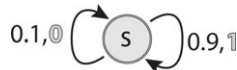
Both of the above examples can be understood as instances of the product of experts pattern discussed in Section 3.2. In the first case, the additional expert eliminates certain possibilities by assigning zero probability to them, and in the second case, the additional expert merely modifies probabilities by preferring the symbol "1" to the symbol "0."

$$\text{goal}_{1 \bullet 2} \quad \oplus= \quad \text{path}_{1 \bullet 2}(Q_1, Q_1) \otimes \text{final}_1(Q_2) \otimes \text{final}_2(Q_2). \tag{31}$$

$$\text{path}_{1 \bullet 2}(Q_1, Q_2) \quad \oplus= \quad \text{initial}_1(Q_1) \otimes \text{initial}_2(Q_2). \tag{32}$$

$$\text{path}_{1 \bullet 2}(Q_1, Q_2) \quad \oplus= \quad \text{path}_{1 \bullet 2}(P_1, P_2) \otimes \text{arc}_1(P_1, Q_1, A_1) \otimes \text{arc}_2(P_2, Q_2, A_2) \boxed{\text{if } A_1 = A_2.} \tag{33}$$

Fig. 15. The constrained product of two of the WFSA experts described in Figure 7.

$$\text{goal} \quad \oplus= \quad \text{path}(Q) \otimes \text{final}(Q). \tag{34}$$

$$\text{path}(Q) \quad \oplus= \quad \text{initial}(Q). \tag{35}$$

$$\text{path}(Q) \quad \oplus= \quad \text{path}(P) \otimes \text{arc}(P, Q, A, B). \tag{36}$$

Fig. 16. The weighted logic program describing WFSTs.

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

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

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

Fig. 17. The *composition* of two WFSTs can be derived by constraining the product of two WFSTs.

5.1.2 *Weighted finite-state transducers*

Suppose we take the PRODUCT transformation of the WFST recognition algorithm in Figure 16 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 Figure 17; 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 of string transduction (Pereira and Riley 1997).

5.2 *Context-free parsing*

Parsing natural languages is a difficult, central problem in computational linguistics (Manning and Schütze 1999). Consider the sentence "Alice saw Bob with binoculars." One analysis (the most likely in the real world) is that Alice had the binoculars and saw Bob through them. Another is that *Bob* had the binoculars and Alice saw the binocular-endowed Bob. Figure 18 shows syntactic parses into noun phrases (**NP**), verb phrases (**VP**), etc., corresponding to these two meanings. It also shows some of the axioms that could be used to specify a context-free grammar (CFG) describing English sentences in Chomsky normal form (Hopcroft and Ullman 1979)[5]. A proof

---

[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.
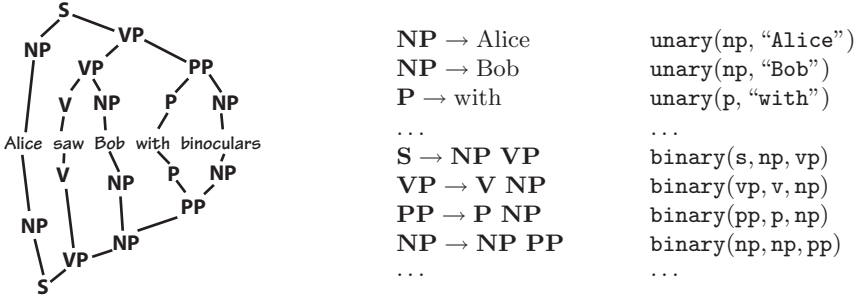
Fig. 18. An ambiguous sentence that can be parsed two ways in English (left), some of the Chomsky normal form rules for English grammar (center), and the corresponding axioms (right). There would also need to be five axioms of the form string(1, "Alice"), string(2, "saw"), etc.

$$\text{goal}_1 \quad \oplus= \quad \text{start}_1(\text{S}) \otimes \text{length}(\text{N}) \otimes c_1(\text{S}, 0, \text{N}). \tag{39}$$

$$c_1(\text{X}, \text{I} - 1, \text{I}) \quad \oplus= \quad \text{unary}_1(\text{X}, \text{W}) \otimes \text{string}(\text{I}, \text{W}). \tag{40}$$

$$c_1(\text{X}, \text{I}, \text{K}) \quad \oplus= \quad \text{binary}_1(\text{X}, \text{Y}, \text{Z}) \otimes c_1(\text{Y}, \text{I}, \text{J}) \otimes c_1(\text{Z}, \text{J}, \text{K}). \tag{41}$$

Fig. 19. A weighted logic program for parsing weighted context-free grammars.

corresponds to a derivation of the given sentence in a context-free grammar, i.e., a parse tree.

Shieber *et al.* (1995) show that parsing with CFGs can be formalized as a logic program, and in Goodman (1999), this framework is extended to the weighted case. If weights are interpreted as probabilities, then the $\langle [0, 1], \max, \times, 0, 1 \rangle$ semiring interpretation finds the probability of the parse with maximum probability and the $\langle \mathbb{R}_{\geqslant 0} \cup \{\infty\}, +, \times, 0, 1 \rangle$ semiring interpretation finds the total weight of all parse trees (a measure of the "total grammatically" of a sentence). In Figure 19, we give the specification of the weighted version of the Cocke-Kasami-Younger (CKY) algorithm (Kasami 1965; Younger 1967; Cocke and Schwartz 1970), which is a dynamic programming algorithm for parsing using a context-free grammar in Chomsky normal form[6].

Figure 19 suggestively has a subscript attached to all but the length and string inputs. In our description of the product of experts framework in Section 3.2, the axioms length and string correspond to the conditional input sentence I. The unconstrained result of the PRODUCT transformation on the combination of the rules in Figure 19 and a second copy that has "2" subscripts is given in Figure 20. Under the most-probable-path probabilistic interpretation, the value of $\text{goal}_{1 \bullet 2}$ is the probability of the given string being generated twice, once by each of the two probabilistic grammars, in each case by the most probable tree in that grammar. By constraining Figure 20, we get the more interesting program in Figure 21 that adds the additional requirement that the two parse trees in the two different grammars have the same *structure*. In particular, in all cases, the constraints $I_1 = I_2$, $J_1 = J_2$,

[6] Strictly speaking, the CKY parsing algorithm corresponds to a naïve bottom-up evaluation strategy for this program.

$$\text{goal}_{1\bullet2} \quad \oplus= \quad \texttt{length}(N_1) \otimes \texttt{length}(N_2) \otimes \qquad (42)$$
$$\texttt{start}_1(S_1) \otimes \texttt{start}_2(S_2) \otimes c_{1\bullet2}(S_1, 0, N_1, S_2, 0, N_2).$$
$$c_{1\bullet2}(X_1, I_1 - 1, I_1, X_2, I_2 - 1, I_2) \quad \oplus= \quad \texttt{unary}_1(X_1, W_1) \otimes \texttt{string}(I_1, W_1) \otimes \qquad (43)$$
$$\texttt{unary}_2(X_2, W_2) \otimes \texttt{string}(I_2, W_2).$$
$$c_{1\bullet2}(X_1, I_1 - 1, I_1, X_2, I_2, K_2) \quad \oplus= \quad \texttt{unary}_1(X_1, W_1) \otimes \texttt{string}(I_1, W_1) \otimes \qquad (44)$$
$$\texttt{binary}_2(X_2, Y_2, Z_2) \otimes c_2(Y_2, I_2, J_2) \otimes c_2(Z_2, J_2, K_2).$$
$$c_{1\bullet2}(X_1, I_1, K_1, X_2, I_2 - 1, I_2) \quad \oplus= \quad \texttt{unary}_2(X_2, W_2) \otimes \texttt{string}(I_2, W_2) \otimes \qquad (45)$$
$$\texttt{binary}_1(X_1, Y_1, Z_1) \otimes c_1(Y_1, I_1, J_1) \otimes c_1(Z_1, J_1, K_1).$$
$$c_{1\bullet2}(X_1, I_1, K_1, X_2, I_2, K_2) \quad \oplus= \quad \texttt{binary}_1(X_1, Y_1, Z_1) \otimes \texttt{binary}_2(X_2, Y_2, Z_2) \otimes \qquad (46)$$
$$c_{1\bullet2}(Y_1, I_1, J_1, Y_2, I_2, J_2) \otimes c_{1\bullet2}(Z_1, J_1, K_1, Z_2, K_2, J_2).$$

Fig. 20. The result of the PRODUCT transformation on two copies of Figure 19.

$$\text{goal}_{1\bullet2} \quad \oplus= \quad \texttt{length}(N) \otimes \texttt{start}_1(S_1) \otimes \texttt{start}_2(S_2) \otimes c_{1\bullet2}(S_1, S_2, 0, N) \quad (47)$$
$$c_{1\bullet2}(X_1, X_2, I - 1, I) \quad \oplus= \quad \texttt{unary}_1(X_1, W) \otimes \texttt{unary}_2(X_2, W) \otimes \texttt{string}(I, W). \qquad (48)$$
$$c_{1\bullet2}(X_1, X_2, I, K) \quad \oplus= \quad \texttt{binary}_1(X_1, Y_1, Z_1) \otimes \texttt{binary}_2(X_2, Y_2, Z_2) \otimes \qquad (49)$$
$$c_{1\bullet2}(Y_1, Y_2, I, J) \otimes c_{1\bullet2}(Z_1, Z_2, J, K).$$

Fig. 21. The program in Figure 20 constrained to require internally identical trees.
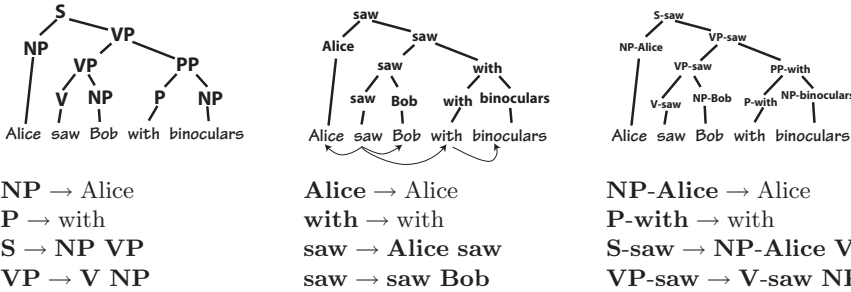
Fig. 22. On the left, the grammar previously shown. In the middle, a context-free *dependency grammar*, whose derivations can be seen as parse trees (above) or a set of dependencies (below). On the right, a lexicalized grammar. Sample rules are given for each grammar.

$K_1 = K_2$, and $N_1 = N_2$ are added, so that instead of writing $c_{1\bullet2}(X_1, I_1, J_1, X_2, I_2, J_2)$, we just write $c_{1\bullet2}(X_1, X_2, I, J)$.

5.2.1 *Lexicalized CFG parsing*

An interesting variant of the previous rule involves *lexicalized grammars*, which are motivated in Figure 22. Instead of describing a grammar using nonterminals denoting phrases (e.g., **NP** and **VP**), which is called a *constituent-structure grammar,* we can define a (context-free) *dependency grammar* (Gaifman 1965) that encodes the syntax of a sentence in terms of parent–child relationships between words. In the case of the example of Figure 22, 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."

A simple kind of dependency grammar is a Chomsky normal form CFG 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 $\mathbf{P} \rightarrow \mathbf{P\ C}$, $\mathbf{P} \rightarrow \mathbf{C\ P}$, and $\mathbf{W} \rightarrow w$ (where $\mathbf{P}$ is the "parent" word, $\mathbf{C}$ is the "child" word that is dependent on the parent, and $\mathbf{W}$ is the nonterminal version of terminal word w).

If we encode the constituent-structure grammar in the $\texttt{unary}_1$ and $\texttt{binary}_1$ relations and encode a dependency grammar in the $\texttt{unary}_2$ and $\texttt{binary}_2$ relations, then the product is a *lexicalized grammar*, like the third example in Figure 22. In particular, it describes a lexicalized context-free grammar with a product of experts probability model (Klein and Manning 2003), because the weight given to any production $\mathbf{A\text{-}X} \rightarrow \mathbf{B\text{-}X\ C\text{-}Y}$ is the semiring product of the weight given to the production $\mathbf{A} \rightarrow \mathbf{B\ C}$ and the weight given to the dependency-based production $\mathbf{X} \rightarrow \mathbf{X\ Y}$. This was an important distinction for Klein and Manning—they were interested in *factored* lexicalized grammars that Figure 21 can describe. These are only a small (but interesting) subset of all possible lexicalized grammars. Standard lexicalized CFGs assign weights directly to grammar productions of the form $\mathbf{A\text{-}X} \rightarrow \mathbf{B\text{-}X\ C\text{-}Y}$, not indirectly (as we do) by assigning weights to a constituent structure and a dependency grammar. We will return to this point in Section 6.2 when we consider the "axiom generalization" pattern that allows us to describe general lexicalized CKY parsing (Eisner 2000; Eisner and Satta 1999).

5.2.2 Nondeterminism and rule binarization

The result of the PRODUCT transformation shown in Figure 20 was the first time the nondeterminism inherent in lines 6–8 of the description of the PRODUCT transformation (Figure 11) has come into play. Because there were two $\texttt{c}_1$ premises and two $\texttt{c}_2$ premises, they could have been merged in more than one way. For example, the following would have been a potential alternative to rule (46):

$$\texttt{c}_{1\bullet2}(\texttt{X}_1, \texttt{I}_1, \texttt{K}_1, \texttt{X}_2, \texttt{I}_2, \texttt{K}_2) \oplus= \texttt{binary}_1(\texttt{X}_1, \texttt{Y}_1, \texttt{Z}_1) \otimes \texttt{binary}_2(\texttt{X}_2, \texttt{Y}_2, \texttt{Z}_2) \otimes \qquad (50)$$
$$\texttt{c}_{1\bullet2}(\texttt{Y}_1, \texttt{I}_1, \texttt{J}_1, \texttt{Z}_2, \texttt{K}_2, \texttt{J}_2) \otimes \texttt{c}_{1\bullet2}(\texttt{Z}_1, \texttt{J}_1, \texttt{K}_1, \texttt{Y}_2, \texttt{I}_2, \texttt{J}_2).$$

However, this would have broken the correspondence between $\texttt{I}_1$ and $\texttt{I}_2$ and made it impossible to constrain the resulting program as we did. An alternative to CKY is the *binarized* variant of CKY where rule (41) is split into two rules by introducing a new, temporary predicate (rules (51) and (52)):

$$\texttt{temp}_1(\texttt{X}, \texttt{Y}, \texttt{J}, \texttt{K}) \oplus= \texttt{binary}_1(\texttt{X}, \texttt{Y}, \texttt{Z}) \otimes \texttt{c}_1(\texttt{Z}, \texttt{J}, \texttt{K}), \qquad (51)$$
$$\texttt{c}_1(\texttt{X}, \texttt{I}, \texttt{K}) \oplus= \texttt{c}_1(\texttt{Y}, \texttt{I}, \texttt{J}) \otimes \texttt{temp}_1(\texttt{X}, \texttt{Y}, \texttt{J}, \texttt{K}). \qquad (52)$$

In this variant, the nondeterministic choice in the PRODUCT transformation disappears. The choice that we made in pairing was consistent with the choice that is forced in the binarized CKY program.

$$\text{goal} \quad \oplus= \quad \text{targetlength}(M) \otimes \text{predict}(E_{M-1}, E_M, M+1). \qquad (53)$$

$$\text{predict}(E_{J-1}, E_J, J+1) \quad \oplus= \quad \text{predict}(E_{J-2}, E_{J-1}, J) \otimes \text{trigram}(E_{J-2}, E_{J-1}, E_J). \quad (54)$$

Fig. 23. A weighted logic program giving a trigram prediction model for a language, which can be generalized to an *n*-gram model for any *n*.

5.3 *Translation algorithms*

Another example of two probabilistic models that play the role of experts arises in translation of sentences from one natural language to another. We will summarize how the PRODUCT transformation was applied to a simple form of phrase-to-phrase translation (Koehn *et al.* 2003) by Lopez (2009).

Lopez (2009) suggested a deductive view of algorithms for machine translation, similar to the view of parsing given by Shieber *et al.* (1995). Lopez used the PRODUCT transformation to derive an algorithm for phrase translation from two different factor programs, one which attempts to enforce *fluency* (a measure of the grammaticality of a sentence) in the translated sentence and one which attempts to enforce *adequacy* (a measure of how much of the meaning of an original sentence is preserved in the translation.)

If fluency is a measure of the grammaticality of a sentence, then it would seem that the CKY algorithm for parsing context-free grammars would be a candidate. While such models have been used in translation (Charniak *et al.* 2003), Lopez's example uses a simpler notion of fluency based on an *n*-gram language model (Manning and Schütze 1999, Chapter 6). An *n*-gram model assigns the probability of a sentence to be the product of probabilities of each word following the $(n-1)$-word sequence immediately preceding it. As a concrete example, let us say that $n = 3$ (called a "trigram" model) and work with the program in Figure 23. If we were estimating our trigram model on the basis of the relative frequencies of sequences in Shakespeare's *Othello*, we would note that the phrase "if it" appears eight times in the text. Three of these are from the sequence "if it be" and one is from the sequence "if it prove," so the axiom trigram("if", "it", "be") should have a probability that is three times the probability given to trigram("if", "it", "prove"). If we then stared the program with the initial sentence fragment predict("if", "it", 3), we could derive predict("it", "be", 4) with the aforementioned axiom and then predict("be", "demanded", 5) with the axiom trigram("it", "be", "demanded"), a sequence occurring once in the text. The result so far is a sequence "if it be demanded" that does not appear in *Othello*, but which perhaps sounds like it could (which is an informal way of describing the criterion for fluency).

The weighted logic program that Lopez uses to enforce adequacy is the "monotone decoding" logic program presented in Figure 24. The program is slightly contrived in order to interact with the PRODUCT transformation correctly. The atomic proposition trans(I, Es) refers to a particular point, I, in the source-language string and a list Es of unprocessed words in the target language[7]. Each deduction consumes

---

[7] We use a standard syntactic shorthand for lists; "[]" can be read as the constant nil, and "E :: Es" can be read as cons(E, Es).

$$\text{goal} \quad \oplus\!= \quad \text{sourcelength}(\text{N}) \otimes \text{trans}(\text{N}, []). \tag{55}$$

$$\text{trans}(\text{I}', \text{Es}) \quad \oplus\!= \quad \text{trans}(\text{I}, []) \otimes \text{phrase}(\text{I}, \text{I}', \text{E}_\text{J} :: \text{Es}). \tag{56}$$

$$\text{trans}(\text{I}', \text{Es}) \quad \oplus\!= \quad \text{trans}(\text{I}', \text{E}_\text{J} :: \text{Es}). \tag{57}$$

Fig. 24. A weighted logic program that describes monotone decoding—translating a phrase at a time of the input language into the output language without reordering.

$$\text{goal} \quad \oplus\!= \quad \text{sourcelength}(\text{N}) \otimes \text{targetlength}(\text{M}) \otimes \tag{58}$$
$$\text{pr}\bullet\text{tr}(\text{N}, \text{M}, \text{E}_{\text{M}-1}, \text{E}_\text{M}, \text{M}+1).$$

$$\text{pr}\bullet\text{tr}(\text{I}', \text{J}+1, \text{E}_{\text{J}-1}, \text{E}_\text{J}, \text{Es}) \quad \oplus\!= \quad \text{pr}\bullet\text{tr}(\text{I}, \text{J}, \text{E}_{\text{J}-2}, \text{E}_{\text{J}-1}, []) \otimes \tag{59}$$
$$\text{trigram}(\text{E}_{\text{J}-2}, \text{E}_{\text{J}-1}, \text{E}_\text{J}) \otimes \text{phrase}(\text{I}, \text{I}', \text{E}_\text{J} :: \text{Es}).$$

$$\text{pr}\bullet\text{tr}(\text{I}', \text{J}+1, \text{E}_{\text{J}-1}, \text{E}_\text{J}, \text{Es}) \quad \oplus\!= \quad \text{pr}\bullet\text{tr}(\text{I}', \text{J}, \text{E}_{\text{J}-2}, \text{E}_{\text{J}-1}, \text{E}_\text{J} :: \text{Es}) \otimes \tag{60}$$
$$\text{trigram}(\text{E}_{\text{J}-2}, \text{E}_{\text{J}-1}, \text{E}_\text{J}).$$

Fig. 25. Phrase translation as the constrained product of Figures 23 and 24.

a single word (E_J) in the target language—indeed, this is the only function of rule (57). When there are no words to remove, then either the entire source-language string has been translated (rule (55)), or else progress can continue by translating some chunk of the source-language sentence starting from position I and ending at position I' as the nonempty list of target-language words $\text{E}_\text{J} :: \text{Es}$ and applying rule (56). This translation of a sequence of the source-language words is captured by the premise $\text{phrase}(\text{I}, \text{I}', \text{Es})$, corresponding to the source subsequence from position I to position I' being translated as Es (a target-language phrase). The meaning of phrase could be defined by a set of axioms or by a rule. In the latter case, if we enumerate all the substrings Ds in the source-language sentence as axioms $\text{substr}(\text{I}, \text{I}', \text{Ds})$ and provide axioms $\text{ptranslate}(\text{Ds}, \text{Es})$ describing source-language to target-language phrase translation, then $\text{phrase}(\text{I}, \text{I}', \text{Es})$ may be defined by the following rule:

$$\text{phrase}(\text{I}, \text{I}', \text{Es}) \oplus\!= \text{substr}(\text{I}, \text{I}', \text{Ds}) \otimes \text{ptranslate}(\text{Ds}, \text{Es}). \tag{61}$$

Note that substr might be provided as an axiom or derived from axioms encoding the source sentence through another inference rule.

Figure 25 displays Lopez's phrase translation program by constraining the product of the *n*-gram model and monotone decoding programs. Lopez describes this for any *n*, but for simplicity, we continue using a trigram model ($n = 3$). The combined predicate simultaneously tracks a position in the source-language sentence I and the target-language sentence J. The word E_J that was discarded at each step in Figure 24 is given relevance by the trigram model. The combination of these two programs uses the monotone decoding program's capabilities to make sure that the phrase-by-phrase meaning of the source-language string $D_1, \ldots, D_N$ is preserved in the destination language string $E_1, \ldots, E_M$ (adequacy) while simultaneously using the trigram model's capabilities to ensure that the result is a plausible sentence in the destination language (fluency).

Our presentation of machine translation algorithms through the PRODUCT transformation is simplistic. Lopez (2009) discusses more powerful translation algorithms that permit, for example, reordering of phrases.

6 Variations on PRODUCT

Up to this point, we have viewed our use of the PRODUCT transformation as the one that solves a problem of *joint optimization*: we take two logic programs that describe structures (such as strings, paths, or trees), relate them to one another by adding constraints, and then optimize over the two original structures simultaneously (one instance of this is when we use weighted logic programming to describe a product of experts.) This is a useful pattern, but it is not the only interesting use of the fold/unfold transformation underlying the PRODUCT transformation. In this section, we consider two other variants: in the first, we only optimize over one of the two structures and fix the other one, and in the second, we take the output of PRODUCT as describing not joint optimization over two simple structures but over one complex structure.

6.1 Fixing one of the factor structures

The usual use of the PRODUCT transformation is to perform joint optimization on two structures, but general side conditions can be used to take the additional step of *fixing* one of the two structures and having the weighted logic program perform optimization on the *other* structure, subject to constraints imposed through the pairing.

In the setting where we consider weights to be probabilities, this is useful for solving certain probabilistic inference problems. Using the path-sum semiring (i.e., $\langle \mathbb{R}_{\geqslant 0} \cup \{\infty\}, +, \times, 0, 1 \rangle$), the result is a program calculating the *marginalized quantity* $p(x) = \sum_y p(x, y)$ (where x corresponds to one program's proof and y to the other program's proof). This is a useful quantity in learning; for example, the expectation-maximization algorithm (Dempster *et al.* 1977) for optimizing the marginalized log likelihood of observed structures requires calculating *sufficient statistics* which are based on marginal quantities. Using the most-probable-path semiring (i.e., $\langle [0, 1], \max, \times, 0, 1 \rangle$), the result is a program for solving $\text{argmax}_y\, p(y \mid x)$—that is, for finding the most probable y given the fixed x.

The transformation of the constrained result of the PRODUCT transformation to a program with one proof fixed is essentially mechanical. We consider the example of lexicalized parsing from Figure 22. We take the constituent-structure parse as the structure we want to fix in order to optimize over the possible matching parses from the dependency grammar. The shape of the constituent-structure parse tree can be represented by a series of new axioms that mirror the structure of the $c_1(X, I, J)$ predicate defining the constituent-structure grammar: $\text{proof}_1(s, 0, 5)$, $\text{proof}_1(np, 0, 1)$, $\text{proof}_1(vp, 1, 5)$, $\text{proof}_1(vp, 1, 3)$, $\text{proof}_1(pp, 3, 5)$, and so on.

Then, we take the constrained PRODUCT of CKY that we used to describe lexicalized parsing (Figure 21), and wherever there was a conclusion derived from c_1, we add

$$\text{path}_{1\bullet 2}(Q_1, Q_2) \quad \oplus= \quad \text{initial}_{1\bullet 2}(Q_1, Q_2). \tag{63}$$

$$\text{path}_{1\bullet 2}(Q_1, Q_2) \quad \oplus= \quad \text{path}_{1\bullet 2}(P_1, P_2) \otimes \text{arc}_{1\bullet 2}(P_1, P_2, Q_1, Q_2, A_1, A_2). \tag{64}$$

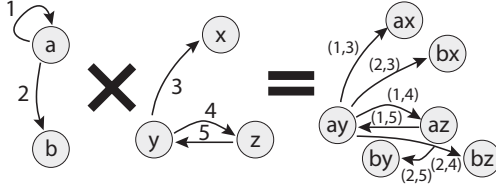Fig. 26. WFSTs as the product of two weighted finite-state machines.

Fig. 27. A finite-state transducer that can be expressed as the PRODUCT of two finite-state automata.

a matching side condition that references proof_1. The critical rule (49) ends up looking like this:

$$c_{1\bullet 2}(X_1, X_2, I, K) \oplus= \text{binary}_1(X_1, Y_1, Z_1) \otimes \text{binary}_2(X_2, Y_2, Z_2) \otimes \tag{62}$$
$$c_{1\bullet 2}(Y_1, Y_2, I, J) \otimes c_{1\bullet 2}(Z_1, Z_2, J, K) \text{ if } \text{proof}_1(X_1, I, K).$$

The effect of this additional constraint is to disqualify any proof that does not match the constituent-structure grammar, which we have fixed and encoded as proof_1 axioms. The idea of partially constraining CFG derivations with some bracketing structure was explored by Pereira and Schabes (1992).

6.2 Axiom generalization

Axiom generalization is another way of manipulating products of weighted logic programs in such a way that reveals the simple structures underlying a complex structure. Figure 26, which is intended to describe a WFST, is close to the weighted logic program in Figure 15 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 A_1 from the first expert as the transducer's input symbol and A_2 as the transducer's output symbol. Second, we have merged $\text{initial}_1(Q_1) \otimes \text{initial}_2(Q_2)$ with the single product predicate $\text{initial}_{1\bullet 2}(Q_1, Q_2)$, and likewise for arc. As a first approximation, we can just define $\text{arc}_{1\bullet 2}$ (and, similarly, $\text{initial}_{1\bullet 2}$) by a single rule of this form:

$$\text{arc}_{1\bullet 2}(P_1, P_2, Q_1, Q_2, A_1, A_2) \oplus= \text{arc}_1(P_1, Q_1, A_1) \otimes \text{arc}_2(P_2, Q_2, A_2). \tag{65}$$

An example is given in Figure 27. Two finite-state *machines*, one with two states (a and b) and one with three states (x, y and z), are shown—we are working over the Boolean semiring, so each arc in the figure corresponds to a true-valued arc axiom. The PRODUCT of these two experts in the manner of Figure 26 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
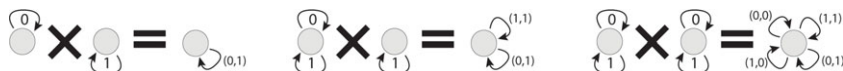
$$\text{goal}_{1 \bullet 2} \oplus= \text{length}(N) \otimes \text{start}(S) \otimes c_{1 \bullet 2}(S, W, 0, N). \tag{66}$$

$$c_{1 \bullet 2}(X, W, I - 1, I) \oplus= \text{unary}_{1 \bullet 2}(X, W) \otimes \text{string}(I, W). \tag{67}$$

$$c_{1 \bullet 2}(X, W, I, K) \oplus= \text{binary}_{1 \bullet 2}(X, W, Y, W_1, Z, W_2) \otimes c_{1 \bullet 2}(Y, W_1, I, J) \otimes c_{1 \bullet 2}(Z, W_2, J, K). \tag{68}$$

Fig. 28. A algorithm for CKY over a general lexicalized grammar derived from Figure 21 by axiom generalization.

possible transducers, but only 10 can be "factored" as two independent finite-state machines, such as these three:

Six others, like the NOT transducer that outputs 1 given the input 0 and outputs 0 given the input 1, cannot be represented as the product of two finite-state machines.

In many settings, limiting ourselves to the "factorable" finite-state transducers (or lexicalized grammars) can have conceptual or computational advantages. When this does not suffice, we can perform *axiom generalization*, which amounts to removing the requirement of equation (65) that the value of atomic propositions of the form $\text{arc}_{1 \bullet 2}$ be the product of an atomic proposition of the form arc_1 and an atomic proposition of the form arc_2. If we directly define axioms of the form $\text{arc}_{1 \bullet 2}$, we can describe transducers in their full generality.

This represents a new way of thinking about the PRODUCT transformation. Thus far, we have considered the result of the PRODUCT transformation as a way of describing programs that work over two different structures. Axiom generalization suggests that we can consider the PRODUCT transformation as a way of taking two programs that work over individual structures and deriving a new program that works over a *single more complicated structure* that, in special cases, can be factored into two different structures. This is particularly relevant in the area of lexicalized grammars and parsing, where the general, more complicated structure is what came first, and the factored models that we have considered thus far arose later as special cases.

6.2.1 *Parsing algorithms and the* PRODUCT *transformation*

Many parsing algorithms can be derived by using the PRODUCT transformation as a way of deriving programs that do not neatly factor into two parts. Lexicalized parsing is a simple example; Figure 28 derives a lexicalized parser by performing axiom generalization on Figure 21. The grammar production "**P-with** → with" can be represented by including the axiom $\text{unary}_{1 \bullet 2}(p, \text{"with"})$, and the binary production **S-saw** → **NP-Alice VP-saw** can be represented by the axiom $\text{binary}_{1 \bullet 2}(s, \text{"saw"}, np, \text{"alice"}, vp, \text{"saw"})$.

Synchronous grammars are another instance in which the axiom generalization view is interesting. A synchronous grammar can be thought of as parsing two different sentences in two different languages with two different grammars, using a *single* parse tree. For example, if **X** → **YZ** is a grammar production in one language and **A** → **BC** is a grammar production in another language, then **X-A** → **Y-B Z-C** is a possible grammar production in the synchronous grammar.

$$\text{goal} \quad \oplus= \quad \text{length}(N) \otimes \text{start}(S) \otimes c(S, 0, N). \tag{69}$$

$$c(X, I, I) \quad \oplus= \quad \text{unary}(X, \epsilon) \otimes \text{pos}(I). \tag{70}$$

$$c(X, I-1, I) \quad \oplus= \quad \text{unary}(X, W) \otimes \text{string}(I, W). \tag{71}$$

$$c(X, I, K) \quad \oplus= \quad \text{binary}(X, Y, Z) \otimes c(Y, I, J) \otimes c(Z, J, K). \tag{72}$$

Fig. 29. A variant of CKY that handles grammar productions of the form $\mathbf{X} \to \epsilon$.

$$\text{goal}_{1\bullet 2} \quad \oplus= \quad \text{length}_1(M) \otimes \text{length}_2(N) \otimes \text{start}_{1\bullet 2}(S) \otimes \tag{73}$$
$$c_{1\bullet 2}(S, 0, N, 0, M).$$

$$c_{1\bullet 2}(X, I-1, I, J, J) \quad \oplus= \quad \text{unary}_{1\bullet 2}(X, W_1, \epsilon) \otimes \text{string}_1(I, W_1) \otimes \text{pos}_2(J). \tag{74}$$

$$c_{1\bullet 2}(X, I, I, J-1, J) \quad \oplus= \quad \text{unary}_{1\bullet 2}(X, \epsilon, W_2) \otimes \text{pos}_1(I) \otimes \text{string}_2(J, W_2). \tag{75}$$

$$c_{1\bullet 2}(X, I-1, I, J-1, J) \quad \oplus= \quad \text{unary}_{1\bullet 2}(X, W_1, W_2) \otimes \text{string}_1(I, W_1) \otimes \text{string}_2(J, W_2). \tag{76}$$

$$c_{1\bullet 2}(X_1, I_1, K_1, I_2, K_2) \quad \oplus= \quad \text{binary}_{1\bullet 2}(X, Y, Z) \otimes \tag{77}$$
$$c_{1\bullet 2}(Y, I_1, J_1, I_2, J_2) \otimes c_{1\bullet 2}(Z, J_1, K_1, J_2, K_2).$$

Fig. 30. A simple transduction grammar derived from Figure 29.

A *transduction grammar* (Wu 1997) is a synchronous grammar that generates two isomorphic derivations with a trivial alignment between the nodes of those two derivations. We can describe a parser for a transduction grammar with the program in Figure 30. Synchronous grammars need to be able to deal with situations in which a word in one language does not appear in the matching sentence in the other language; this is done by starting from the enriched CKY program in Figure 29 that can handle grammar productions of the form $\mathbf{X} \to \epsilon$.

In practice, transduction grammars do a bad job of aligning two sentences in different natural languages that are translations of each other, because it is often the case that two parts of a pair of sentences need to be in opposite positions relative to one another—in language one, the verb phrase might precede a prepositional phrase, and in language two, the corresponding verb phrase might *follow* the corresponding prepositional phrase. An *inversion transduction grammar* describes an alternate form of grammar production, which Wu (1997) writes as $\mathbf{X} \to \langle \mathbf{YZ} \rangle$. This grammar production declares that if A_1 and A_2 simultaneously parse as $\mathbf{Y}$ in languages one and two (respectively) and B_1 and B_2 simultaneously parse as $\mathbf{Z}$ in languages one and two (respectively), then $A_1 B_1$ and $B_2 A_2$ simultaneously parse as $\mathbf{Z}$.

Somewhat surprisingly, this inversion production rule can be described using the *alternate* allowable way of merging the premises when the PRODUCT transformation is performed on two copies of the CKY algorithm, as discussed in Section 5.2 (see rule (50)). By adding this alternate form as given in Figure 31, we can describe the algorithm for parsing with inversion transduction grammars described by Wu (1997).

7 The entropy semiring and Kullback–Leibler divergence

An important construct in information theory and machine learning is the Kullback–Leibler (KL) divergence (Kullback and Leibler 1951). KL divergence is a function

$$\text{c}_{1 \bullet 2}(\text{X}_1, \text{I}_1, \text{K}_1, \text{I}_2, \text{K}_2) \quad \oplus = \quad \text{inversion}_{1 \bullet 2}(\text{X}, \text{Y}, \text{Z}) \otimes \tag{78}$$
$$\text{c}_{1 \bullet 2}(\text{Y}, \text{I}_1, \text{J}_1, \text{J}_2, \text{K}_2) \otimes \text{c}_{1 \bullet 2}(\text{Z}, \text{J}_1, \text{K}_1, \text{I}_2, \text{J}_2).$$

Fig. 31. By adding to Figure 30 the rule corresponding to the *other* way that the c_1 and c_2 antecedents may be merged in the PRODUCT transformation, we can describe an inversion transduction grammar.

of two probability distributions over the same event space. It measures their dissimilarity, though it is not, strictly speaking, a distance (it is not symmetric). For two distributions p and q for random variable X ranging over events $x \in \mathscr{X}$, KL divergence is defined as

$$\text{KL}(p \| q) = \sum_{x \in \mathscr{X}} p(X = x) \log \frac{p(X = x)}{q(X = x)}, \tag{79}$$

$$= \underbrace{\sum_{x \in \mathscr{X}} p(X = x) \log p(X = x)}_{-H(p)} - \underbrace{\sum_{x \in \mathscr{X}} p(X = x) \log q(X = x)}_{\text{CE}(p \| q)}, \tag{80}$$

where $H(p)$ denotes the Shannon entropy of the distribution p (Shannon 1948), a measure of uncertainty, and $\text{CE}(p \| q)$ denotes the cross entropy between p and q[8]. A full discussion of these information-theoretic quantities is out of scope for this paper; we note that they are widely used in statistical machine learning (Koller and Friedman 2009). In this section, we first show how the entropy of $p(P)$, with P ranging over proofs of goal (the axioms corresponding to random variables A and I are suppressed here, for clarity), can be calculated using a weighted logic program, following Hwa (2004). We then describe a generalization of a result of Cortes *et al.* (2006) to show how to use PRODUCT to produce a weighted logic program for calculating the KL divergence between the two distributions induced by the weighted logic programs.

7.1 Generalized entropy semiring

The domain of the generalized entropy semiring is $(\mathbb{R} \cup \{+\infty, -\infty\})^3$. The multiplication and addition operations are defined as follows:

$$\langle x_1, y_1, z_1 \rangle \oplus \langle x_2, y_2, z_2 \rangle = \langle x_1 + x_2, y_1 + y_2, z_1 + z_2 \rangle, \tag{81}$$

$$\langle x_1, y_1, z_1 \rangle \otimes \langle x_2, y_2, z_2 \rangle = \langle x_1 x_2, x_1 y_2 + x_2 y_1, z_1 z_2 \rangle. \tag{82}$$

These operations have the required closure, associativity, and commutativity properties previously discussed for semirings. See Cortes *et al.* (2006) for a proof that can be extended trivially to our generalized semiring.

---

[8] In brief, the Shannon entropy of distribution p is the expected number of bits required to send a message drawn according to p under an optimal coding scheme. Cross-entropy is the average number of bits required to encode a message in the optimal coding scheme for q when messages are actually distributed according to p. Hence, $\text{KL}(p \| q) = \text{CE}(p \| q) - H(p)$ is the average number of *extra* bits required when the true distribution of messages is p, but the coding scheme is based on q. Note that $\text{KL}(p \| p) = 0$. If there is an event $x \in \mathscr{X}$ such that $p(x) > 0$ and $q(x) = 0$, then $\text{KL}(p \| q) = +\infty$.

Suppose we have a weighted logic program such that the path sum (in the $\langle \mathbb{R}_{\geqslant 0} \cup \{\infty\}, +, \times, 0, 1 \rangle$ semiring) is 1 (i.e., the value of the `goal` theorem is 1). If we map the weights of all axioms in the original program to new values in the generalized entropy semiring, we can use the new semiring to calculate the Shannon entropy of the distribution over proofs of `goal`:

$$-\sum_{proof} p(P = proof) \log p(P = proof), \tag{83}$$

where x ranges over proofs of `goal`. The mapping is simply $w \mapsto \langle w, -w \log w, 0 \rangle$. (The third element of the semiring value is not needed here.) If we solve the new weighted logic program and achieve value $\langle w', h', 0 \rangle$ for the `goal` theorem, then under the assumption that $w' = 1$ (the value of `goal` in the original program in the real semiring), h' is the entropy of the distribution over the proof random variable (given the axioms and `goal`). The formal result is given as a corollary in Section 7.2.

This semiring can be used, for example, with the CKY algorithm from Figure 19. It makes the derivation of the tree entropy for context-free grammars (i.e., the entropy over the context-free derivations for an ambiguous string) automatic and obviates the design of a specific algorithm for computing the tree entropy for probabilistic context-free grammars, as described in Hwa (2004). With the CKY algorithm, a proof *proof* in equation (83) represents a derivation in the grammar. Similarly, a weighted logic program describing a finite-state transducer (Figure 16) can be used to compute the entropy of hidden sequences for hidden Markov models as described by Hernando *et al.* (2005).

We now relax the assumption that the sum of all proof scores is 1. Suppose that the value of the `goal` theorem in the generalized entropy semiring is $(w', h', 0)$, with $w' \neq 1$. In this case, h' is not the entropy of a proper probability distribution. We can renormalize the scores of the proofs, $u(proof)$, dividing by w', treating them as a proper conditional distribution (conditioning on the truth of the `goal` theorem); then the entropy of this conditional distribution, $\frac{u(proof)}{w'}$, is

$$-\sum_{proof} \frac{u(proof)}{w'} \log \frac{u(proof)}{w'}$$

$$= \frac{1}{w'} \left(-\sum_{proof} u(proof)(\log u(proof) - \log w') \right)$$

$$= \frac{1}{w'} \left(h' + (\log w') \sum_{proof} u(proof) \right) = \frac{1}{w'}(h' + w' \log w') = \frac{h'}{w'} + \log w'. \tag{84}$$

Therefore, whenever we can use weighted logic programming (in the real semiring) to calculate sums of proof scores, we can use the generalized entropy semiring to find the Shannon entropy of the (possibly renormalized) distribution over proofs. The renormalization uses w' and h', two quantities that are calculated directly when we use the generalized entropy semiring.

7.2 *KL divergence between proof distributions and* PRODUCT

Cortes *et al.* (2006) showed how to compute the KL divergence (also called relative entropy) between two distributions over strings defined by probabilistic FSA, using a construct similar to our generalized entropy semiring. We generalize that result to KL divergence over two *proof* distributions $p(P)$ and $q(P)$ given by a weighted logic program $\mathscr{P}$. We assume in this discussion that the set of axioms with nonzero weights is identical under p and q; the general setting where this does not hold is correctly handled, using $a \log 0 = -\infty$ and $0 \log a = 0$ for all $a > 0$.

We abuse notation slightly and use p and q to denote the values of axioms, theorems, and proofs in the real semiring weighted logic programs used to calculate the sum of proof scores for goal under axioms weighted according to p and q. Let $\text{Proofs}(t)$ denote the set of logical proofs of a theorem t, and for $x \in \text{Proofs}(t)$, let $p(t)$—respectively, $q(t)$—denote the score of the proof x:

$$p(t) = \sum_{x \in \text{Proofs}(t)} p(x), \tag{85}$$

$$q(t) = \sum_{x \in \text{Proofs}(t)} q(x). \tag{86}$$

We seek the KL divergence

$$\text{KL}(p \| q) = \sum_{x \in \text{Proofs}(\texttt{goal})} p(x) \log \frac{p(x)}{q(x)}. \tag{87}$$

In order to accomplish this calculation, we will first map the weights of axioms under p and q into the generalized entropy semiring as follows, for any axiom a:

$$\langle p(a), q(a) \rangle \mapsto \langle p(a), p(a) \log q(a), q(a) \rangle. \tag{88}$$

For a theorem t, let

$$R(t) = \sum_{x \in \text{Proofs}(t)} p(x) \log q(x). \tag{89}$$

Theorem 2
Solving $\mathscr{P}$ in the generalized entropy semiring with weights defined as above results in goal having value $\langle p(\texttt{goal}), R(\texttt{goal}), q(\texttt{goal}) \rangle$.

Proof: We will treat the weighted logic program as a set of equations with all left-hand-side variables grounded. We will use uppercase to refer to free variables (e.g., $\mathbf{Z} = \langle Z_1, \ldots \rangle$) and lowercase to refer to grounded values (e.g., $\mathbf{z} = \langle z_1, \ldots \rangle$). The range of values that variables $\mathbf{Z}$ can get is denoted by $\text{Rng}(\mathbf{Z})$. The weighted logic program can be seen as a set of equations:

$$\texttt{c}(\mathbf{w}) = \bigoplus_{[\texttt{c}(\mathbf{w}) \oplus = \texttt{a}_\texttt{i}(\mathbf{w}',\mathbf{Z}) \otimes \texttt{b}_\texttt{i}(\mathbf{w}'',\mathbf{Z})] \in \mathscr{P}, \mathbf{w}' \subseteq \mathbf{w}, \mathbf{w}'' \subseteq \mathbf{w}} \bigoplus_{\mathbf{z} \in \text{Rng}(\mathbf{Z})} \texttt{a}_\texttt{i}(\mathbf{w}', \mathbf{z}) \otimes \texttt{b}_\texttt{i}(\mathbf{w}'', \mathbf{z}). \tag{90}$$

(Note that any of $\mathbf{w}$, $\mathbf{w}'$, $\mathbf{w}''$, and $\mathbf{z}$ may be empty.)

We now show that the value achieved for $c(\mathbf{w})$ when solving in the semiring is

$$\langle p(c(\mathbf{w})), \sum_{x\in\text{Proofs}(c(\mathbf{w}))} p(x)\log q(x), q(c(\mathbf{w}))\rangle, \tag{91}$$

where $\text{Proofs}(c(\mathbf{w}))$ denotes the set of proofs for $c(\mathbf{w})$. We will show that the solution of equations (90) is the value in equation (91) for $c(\mathbf{w})$.

For the first and third coordinates, this equality follows naturally because of the definition of the generalized entropy semiring: the first and third coordinates are equivalent to the nonnegative real semiring used for summing over proof scores under the two value assignments p and q, respectively.

Consider a particular $\oplus$-addend to the value of $c(\mathbf{w})$,

$$a_i(\mathbf{w}', \mathbf{z}) \otimes b_i(\mathbf{w}'', \mathbf{z}) \tag{92}$$

$$= \langle p(a_i(\mathbf{w}', \mathbf{z})), R(a_i(\mathbf{w}', \mathbf{z})), q(a_i(\mathbf{w}', \mathbf{z}))\rangle$$
$$\otimes \langle p(b_i(\mathbf{w}'', \mathbf{z})), R(b_i(\mathbf{w}'', \mathbf{z})), q(b_i(\mathbf{w}'', \mathbf{z}))\rangle \tag{93}$$

$$= \left\langle \begin{array}{l} p(a_i(\mathbf{w}', \mathbf{z}))p(b_i(\mathbf{w}'', \mathbf{z})), \\ p(a_i(\mathbf{w}', \mathbf{z}))R(b_i(\mathbf{w}'', \mathbf{z})) + p(b_i(\mathbf{w}'', \mathbf{z}))R(a_i(\mathbf{w}', \mathbf{z})), \\ q(a_i(\mathbf{w}', \mathbf{z})))q(b_i(\mathbf{w}'', \mathbf{z})), \end{array} \right\rangle \tag{94}$$

Consider the second coordinate.

$$p(a_i(\mathbf{w}', \mathbf{z}))R(b_i(\mathbf{w}'', \mathbf{z})) + p(b_i(\mathbf{w}'', \mathbf{z}))R(a_i(\mathbf{w}', \mathbf{z})) \tag{95}$$

$$= \left(p(a_i(\mathbf{w}', \mathbf{z})) \sum_{x\in\text{Proofs}(b_i(\mathbf{w}'', \mathbf{z}))} p(x)\log q(x) \right),$$

$$+ \left(p(b_i(\mathbf{w}'', \mathbf{z})) \sum_{x'\in\text{Proofs}(a_i(\mathbf{w}', \mathbf{z}))} p(x')\log q(x') \right), \tag{96}$$

$$= \left(\sum_{x'\in\text{Proofs}(a_i(\mathbf{w}', \mathbf{z}))} p(x') \sum_{x\in\text{Proofs}(b_i(\mathbf{w}'', \mathbf{z}))} p(x)\log q(x) \right)$$

$$+ \left(\sum_{x\in\text{Proofs}(b_i(\mathbf{w}'', \mathbf{z}))} p(x) \sum_{x'\in\text{Proofs}(a_i(\mathbf{w}', \mathbf{z}))} p(x')\log q(x') \right), \tag{97}$$

$$= \sum_{x\in\text{Proofs}(b_i(\mathbf{w}'', \mathbf{z}))} \sum_{x'\in\text{Proofs}(a_i(\mathbf{w}', \mathbf{z}))} p(x)p(x')\log(q(x)q(x')), \tag{98}$$

Embedding the above in a $\oplus$-summation over $\mathbf{z}$ and a $\oplus$-summation over inference rule instantiations gives a $\oplus$-summation over proofs of $c(\mathbf{w})$,

$$\sum_{x\in\text{Proofs}(c(\mathbf{w}))} p(x)\log q(x), \tag{99}$$

which is $R(c(\mathbf{w}))$ as desired. $\qquad\square$

Denote by $(\bar{p}, \bar{R}, \bar{q})$ the value for goal in the generalized entropy semiring as discussed above, i.e., $\bar{p} = p(\text{goal})$, $\bar{R} = R(\text{goal})$, and $\bar{q} = q(\text{goal})$. If we wish to renormalize p by $\bar{p}$ and q by $\bar{q}$ to give proper distributions over proofs of goal

$$\text{goal} \quad \oplus= \quad \text{path}(\text{P},\text{Q}) \otimes \text{final}(\text{Q}). \tag{103}$$

$$\text{path}(\text{null},\text{Q}) \quad \oplus= \quad \text{initial}(\text{Q}). \tag{104}$$

$$\text{path}(\text{P}',\text{Q}) \quad \oplus= \quad \text{path}(\text{P},\text{P}') \otimes \text{biarc}(\text{P},\text{P}',\text{Q},\text{A},\text{B}). \tag{105}$$

Fig. 32. WFST where arriving at a certain state depends on the last two states. `null` serves as a placeholder for the nonstate prior to the initial state.

(given axioms and `goal`), then

$$\text{CE}\left(\frac{1}{\bar{p}}p \,\middle\|\, \frac{1}{\bar{q}}q\right) = \frac{\bar{R}}{\bar{p}} - \log \bar{q}. \tag{100}$$

Noting that $-H(p) = \text{CE}(p\|p)$,

$$\text{KL}\left(\frac{1}{\bar{p}}p \,\middle\|\, \frac{1}{\bar{q}}q\right) = \text{CE}(p\|p) - \text{CE}(p\|q), \tag{101}$$

we can solve for the KL divergence of two (possibly renormalized) distributions p and q by using the above results. Alternatively, if the generalized KL divergence between *unnormalized* distributions is preferred (O'Sullivan 1998), note that (in the notation of the above)

$$\sum_{x\in\text{Proofs}(\text{goal})} \left(p(x)\log\frac{p(x)}{q(x)} - p(x) + q(x)\right) = \bar{R} - \bar{p} + \bar{q}. \tag{102}$$

Cortes *et al.* (2006) describe how to compute KL divergence between two probabilistic finite-state automata with a single path per string ("unambiguous" automata). The authors make use of finite-state intersection (discussed above in Section 5.1). This suggests an analogous interpretation of the PRODUCT transformation for computing KL divergence between two weighted logic programs.

Let $\mathscr{P}$ and $\mathscr{Q}$ be two instances of a weighted logic program, with possibly different axiom weights. Assume we set the values of the axioms of $\mathscr{P}$ (ranging over a) to be $\langle p(a), 0, 1\rangle$, and for $\mathscr{Q}$, we set them to $\langle 1, \log q(a), q(a)\rangle$. If we take a PRODUCT of $\mathscr{P}$ and $\mathscr{Q}$, using the "natural" pairing, then we end up with a program that computes $\langle p(\text{goal}), R(\text{goal}), q(\text{goal})\rangle$ in the generalized entropy semiring, where $R(\cdot)$ is specified in equation (89). These quantities can be used to compute KL divergence as specified in equation (101). This is a direct result of Theorem 2.

7.3 KL divergence and projections

We can use PRODUCT to calculate KL divergence between proof distributions even when $\mathscr{P}$ and $\mathscr{Q}$ are not two instances of the same program. We consider cases where the proofs of $\mathscr{P}$ and $\mathscr{Q}$ have a shared semantics, that is, each proof of either $\mathscr{P}$ or $\mathscr{Q}$ maps to an event in some "interpretation space."

As an example, consider the weighted logic program in Figure 16 describing a WFST. In a more general formulation, each state depends on the previous N states visited rather than just the single most recent state. This modification is reflected in Figure 32 for $N = 2$. The axiom `biarc(P',Q,P,A,B)` is to be interpreted as "if the last two states were `Q` and `P'`, transfer to state `P` while reading symbol `A` and emitting

$$\mathtt{goal_{1\bullet2}} \;\oplus\!\!= \; \mathtt{path_{1\bullet2}(P,Q)} \otimes \mathtt{final_1(Q)} \otimes \mathtt{final_2(Q)}. \tag{106}$$

$$\mathtt{path_{1\bullet2}(null,Q)} \;\oplus\!\!= \; \mathtt{initial_1(Q)} \otimes \mathtt{initial_2(Q)}. \tag{107}$$

$$\mathtt{path_{1\bullet2}(P',Q)} \;\oplus\!\!= \; \mathtt{path_{1\bullet2}(P,P')} \otimes \mathtt{arc_1(P',Q,A,B)} \otimes \mathtt{biarc_2(P,P',Q,A,B)} \tag{108}$$

Fig. 33. The PRODUCT program of Figure 16 with Figure 32, with constraints that match proofs according to states and emissions sequences.

the symbol B." Since the two programs have different axioms, the spaces of their respective proofs are different. However, both programs have identical semantics to a proof: a proof (in either program) corresponds to a sequence of states that the transducers go through together with the reading of a symbol and the emission of another symbol.

Running PRODUCT on the WFST in Figure 16 (we call it $\mathscr{P}$) and the WFST in Figure 32 (we call it $\mathscr{Q}$) with a particular pairing and constraints (such that the paths are identical) yields the program in Figure 33. If we let the axioms a in $\mathscr{P}$ have the values $\langle p(a), 0, 1 \rangle$ and the axioms a in $\mathscr{Q}$ have the values $\langle 1, \log q(a), q(a) \rangle$, then the resulting PRODUCT program in Figure 33, as implied by Theorem 2, calculates the KL divergence between two distributions over the set of state paths: one which is defined using a finite-state transducer with $N = 1$ and the other with $N = 2$.

We now generalize this idea for two different programs $\mathscr{P}$ and $\mathscr{Q}$. We assume that PRODUCT is applied in such a way that axioms from $\mathscr{P}$ are paired only with axioms from $\mathscr{Q}$, and vice versa. Further, each proof in the PRODUCT program must decompose into exactly one proof in $\mathscr{P}$ and one proof in $\mathscr{Q}$[9]. For a proof in the PRODUCT program, y, we define $\pi_{\mathscr{P}}(y)$ ($\pi_{\mathscr{Q}}(y)$) to be the projection of y to a proof in $\mathscr{P}$ ($\mathscr{Q}$). The "projection" of a proof is a separation of the proof that uses coupled theorems and axioms into theorems and axioms of only one of the programs. For example, projecting a proof y in the product program in Figure 33 yields two proofs: $\pi_{\mathscr{P}}(y)$ describes a sequence of transitions through the transducer with $N = 1$, and $\pi_{\mathscr{Q}}(y)$ describes a sequence of transitions through the transducer with $N = 2$; yet both proofs correspond to the same sequence of states.

In the generalized entropy semiring, we set the values of the axioms of $\mathscr{P}$ to be $\langle p(a), 0, 1 \rangle$, and for $\mathscr{Q}$, we set them to $\langle 1, \log q(a), q(a) \rangle$. The PRODUCT program computes $\langle p(\mathtt{goal_1}), R(\mathtt{goal_{1\cdot2}}), q(\mathtt{goal_2}) \rangle$. This time, the summation in $R(\mathtt{goal_{1\cdot2}})$ is over proofs that are implicitly paired:

$$R(\mathtt{goal_{1\cdot2}}) = \sum_{y \in \mathrm{Proofs}(\mathtt{goal_{1\cdot2}})} p(\pi_{\mathscr{P}}(y)) \log q(\pi_{\mathscr{Q}}(y)). \tag{109}$$

The quantities $p(\mathtt{goal_1})$, $R(\mathtt{goal_{1\cdot2}})$, and $q(\mathtt{goal_2})$ can be used as before to compute the KL divergence between the distributions over the shared "interpretation space" of the proofs in the two programs. This technique is only correct when interpretations are in a one-to-one correspondence with the proofs in $\mathscr{P}$ and $\mathscr{Q}$, and PRODUCT is applied so that equivalently interpretable proofs in the two programs are paired.

---

[9] Note that these constraints are satisfied in the case of two identical programs with the "natural" pairing, as in Section 7.2.

We note that in the general case, the problem of computing KL divergence between two arbitrary distributions is hard. For example, with Markov networks, there are restrictions, which resemble the restrictions we pose, of clique decomposition (Koller and Friedman 2009).

8 Conclusion

We have described a framework for dynamic programming algorithms whose solutions correspond to proof values in two 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 weighted logic program, are then added manually. The framework naturally captures and permits generalization of many existing algorithms. We have shown how variations on the program transformation enable to include a larger set of algorithms as the result of the program transformation. We have concluded by showing how the program transformation can be used to interpret the computation of KL divergence for two weighted logic programs, which are defined over an identical interpretation space.

Acknowledgements

References

CHARNIAK, E., KNIGHT, K. AND YAMADA, K. 2003. Syntax-based language models for statistical machine translation. In *Proc. of the MT Summit IX*, Association for Machine Translation in the Americas, Washington, DC.

CHIANG, D. 2007. Hierarchical phrase-based translation. *Computational Linguistics 33*, 2, 201–228.

COCKE, J. AND SCHWARTZ, J. T. 1970. *Programming Languages and their Compilers: Preliminary Notes*. Courant Institute of Mathematical Sciences, New York University.

COHEN, S. B. AND SMITH, N. A. 2007. Joint morphological and syntactic disambiguation. In *Proc. of the 2007 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning*, Association for Computational Linguistics, 208–217.

CORTES, C., MOHRI, M., RASTOGI, A. AND RILEY, M. D. 2006. Efficient computation of the relative entropy of probabilistic automata. In *Proc. of LATIN 2006: Theoretical Informatics: 7th Latin American Symposium*, José R. Correa, Alejandro Hevia and Marcos Kiwi, Eds. Lecture Notes in Computer Science, vol. 3887. Springer, 323–336.

DEMPSTER, A., LAIRD, N. AND RUBIN, D. 1977. Maximum likelihood estimation from incomplete data via the EM algorithm. *Journal of the Royal Statistical Society B 39*, 1–38.

DURBIN, R., EDDY, S., KROGH, A. AND MITCHISON, G. 1998. *Biological Sequence Analysis: Probabilistic Models of Proteins and Nucleic Acids*. Cambridge University Press.

EISNER, J. 2000. Bilexical grammars and their cubic-time parsing algorithms. In Harry Bunt and Anton Nijholt Eds. Advances in Probabilistic and Other Parsing Technologies, Kluwer Academic Publishers, 29–62.

EISNER, J. AND BLATZ, J. 2007. Program transformations for optimization of parsing algorithms and other weighted logic programs. In *Proc. of Formal Grammar*, CSLI Publications, 45–85.

EISNER, J., GOLDLUST, E. AND SMITH, N. A. 2005. Compiling Comp Ling: Practical weighted dynamic programming and the Dyna language. In *Proc. of the Human Language Technology Conference and Conference on Empirical Methods in Natural Language Processing*, 281–290.

EISNER, J. AND SATTA, G. 1999. Efficient parsing for bilexical context-free grammars and head automaton grammars. In *Proc. of the 37th Annual Meeting of the Association of Computational Linguistics*, Association of Computational Linguistics, 457–464.

FELZENSZWALB, P. F. AND MCALLESTER, D. 2007. The generalized A* architecture. *Journal of Artificial Intelligence Research 29*, 153–190.

GAIFMAN, H. 1965. Dependency systems and phrase-structure systems. *Information and Control 8*, 3, 304–337.

GOODMAN, J. 1999. Semiring parsing. *Computational Linguistics 25*, 4, 573–605.

HERNANDO, D., CRESPI, V. AND CYBENKO, G. 2005. Efficient computation of the hidden Markov model entropy for a given observation sequence. *IEEE Transactions on Information Theory 51*, 7, 2681–2685.

HINTON, G. E. 2002. Training products of experts by minimizing contrastive divergence. *Neural Computation 14*, 1771–1800.

HOPCROFT, J. E. AND ULLMAN, J. D. 1979. *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley.

HUANG, L. AND CHIANG, D. 2005. Better *k*-best parsing. In *Proc. of the 9th International Workshop on Parsing Technology*, Association for Computational Linguistics, 53–64.

HWA, R. 2004. Sample selection for statistical parsing. *Computational Linguistics 30*, 3, 253–276.

KASAMI, T. 1965. *An Efficient Recognition and Syntax-Analysis Algorithm for Context-Free Languages*. Tech. Rep. AFCRL-65-758, Air Force Cambridge Research Lab.

KLEIN, D. AND MANNING, C. D. 2003. Fast exact inference with a factored model for natural language parsing. In *Advances in Neural Information Processing Systems*, vol. 15. MIT Press, 3–10.

KLEIN, D. AND MANNING, C. D. 2004. Parsing and hypergraphs. *New Developments in Parsing Technology*, Harry Bunt, John Carroll and Giorgio Satta Eds. Kluwer Academic Publishers, 351–372.

KOEHN, P., OCH, F. J. AND MARCU, D. 2003. Statistical phrase-based translation. In *Proc. of the Human Language Technology Conference and Conference of the North American Chapter of the Association for Computational Linguistics*. Association for Computational Linguistics. 48–54.

KOLLER, D. AND FRIEDMAN, N. 2009. *Probabilistic Graphical Models: Principles and Techniques*. MIT Press.

KULLBACK, S. AND LEIBLER, R. A. 1951. On information and sufficiency. *Annals of Mathematical Statistics 22*, 79–86.

LIANG, P., KLEIN, D. AND JORDAN, M. 2008. Agreement-based learning. In *Advances in Neural Information Processing Systems 20*, J. Platt, D. Koller, Y. Singer and S. Roweis, Eds. MIT Press, Cambridge, MA, 913–920.

LOPEZ, A. 2009. Translation as weighted deduction. In *Proc. of the 12th Conference of the European Chapter of the Association for Computational Linguistics*, Association for Computational Linguistics, 532–540.

MANNING, C. AND SCHÜTZE, H. 1999. *Foundations of Statistical Natural Language Processing*. MIT Press.

MCALLESTER, D. 2002. On the complexity analysis of static analyses. *Journal of the ACM 49*, 4, 512–537.

MOHRI, M. 1997. Finite-state transducers in language and speech processing. *Computational Linguistics 23*, 2, 269–311.

O'SULLIVAN, J. A. 1998. Alternating minimization algorithms: From Blahut-Armijo to expectation-maximization. In *Codes, Curves and Signals: Common Threads in Communications*, A. Vardy, Ed. Kluwer, 173–192.

PEREIRA, F. C. N. AND RILEY, M. D. 1997. Speech recognition by composition of weighted finite automata. In *Finite-State Language Processing*, E. Roche and Y. Schabes, Eds. MIT Press, 431–453.

PEREIRA, F. C. N. AND SCHABES, Y. 1992. Inside–outside reestimation from partially bracketed corpora. In *Proc. of the 30th Annual Meeting of the Association for Computational Linguistics*, Association for Computational Linguistics, 128–135.

PETTOROSSI, A. 1999. Synthesis and transformation of logic programs using unfold/fold proofs. *Journal of Logic Programming 41*, 2–3 (December), 197–230.

PETTOROSSI, A. AND PROIETTI, M. 1994. Transformation of logic programs: Foundations and techniques. *Journal of Logic Programming 19*, 261–320.

SHANNON, C. 1948. A mathematical theory of communication. *Bell System Technical Journal 27*, 379–423.

SHIEBER, S. M., SCHABES, Y. AND PEREIRA, F. C. N. 1995. Principles and implementation of deductive parsing. *Journal of Logic Programming 24*, 1–2, 3–36.

SIKKEL, K. 1997. *Parsing Schemata*. Springer-Verlag.

SMITH, A., COHN, T. AND OSBORNE, M. 2005. Logarithmic opinion pools for conditional random fields. In *Proc. of the 43rd Annual Meeting of the Association for Computational Linguistics*, Association for Computational Linguistics, 18–25.

SMITH, D. A. AND SMITH, N. A. 2004. Bilingual parsing with factored estimation: Using English to parse Korean. In *Proc. of the Conference on Empirical Methods in Natural Language Processing*, 49–56.

SUTTON, C. AND MCCALLUM, A. 2005. Piecewise training of undirected models. In *Proc. of the 21st Conference on Uncertainty in Artificial Intelligence*, 568–575.

TARJAN, R. E. 1981. A unified approach to path problems. *Journal of the ACM 28*, 3, 577–593.

WU, D. 1997. Stochastic inversion transduction grammars and bilingual parsing of parallel corpora. *Computational Linguistics 23*, 3, 377–404.

YOUNGER, D. H. 1967. Recognition and parsing of context-free languages in time n^3. *Information and Control 10*, 2, 189–208.