Generalized Queries on Probabilistic Context-Free Grammars

David V. Pynadath and Michael P. Wellman

Abstract—Probabilistic context-free grammars (PCFGs) provide a simple way to represent a particular class of distributions over sentences in a context-free language. Efficient parsing algorithms for answering particular queries about a PCFG (i.e., calculating the probability of a given sentence, or finding the most likely parse) have been developed and applied to a variety of pattern-recognition problems. We extend the class of queries that can be answered in several ways: (1) allowing missing tokens in a sentence or sentence fragment, (2) supporting queries about intermediate structure, such as the presence of particular nonterminals, and (3) flexible conditioning on a variety of types of evidence. Our method works by constructing a Bayesian network to represent the distribution of parse trees induced by a given PCFG. The network structure mirrors that of the chart in a standard parser, and is generated using a similar dynamic-programming approach. We present an algorithm for constructing Bayesian networks from PCFGs, and show how queries or patterns of queries on the network correspond to interesting queries on PCFGs. The network formalism also supports extensions to encode various context sensitivities within the probabilistic dependency structure.

Index Terms—Probabilistic context-free grammars, Bayesian networks.

1 Introduction

Most pattern-recognition problems start from observations generated by some structured stochastic process. Probabilistic context-free grammars (PCFGs) [1], [2] have provided a useful method for modeling uncertainty in a wide range of structures, including natural languages [2], programming languages [3], images [4], speech signals [5], and RNA sequences [6]. Domains like plan recognition, where nonprobabilistic grammars have provided useful models [7], may also benefit from an explicit stochastic model.

Once we have created a PCFG model of a process, we can apply existing PCFG parsing algorithms to answer a variety of queries. For instance, standard techniques can efficiently compute the probability of a particular observation sequence or find the most probable parse tree for that sequence. Section 2 provides a brief description of PCFGs and their associated algorithms.

However, these techniques are limited in the types of evidence they can exploit and the types of queries they can answer. In particular, the existing PCFG techniques generally require specification of a complete observation sequence. In many contexts, we may have only a partial sequence available. It is also possible that we may have evidence beyond simple observations. For example, in natural language processing, we may be able to exploit contextual information about a sentence in determining our beliefs about certain unobservable variables in its parse tree. In addition, we may be interested in computing the probabilities of alternate types of events (e.g., future observations or abstract features of the parse) that the extant techniques do not directly support.

The restricted query classes addressed by the existing algorithms limit the applicability of the PCFG model in domains where we may require the answers to more complex queries. A flexible and expressive representation for the distribution of structures generated by the grammar would support broader forms of evidence and queries than supported by the more specialized algorithms that currently exist. We adopt Bayesian networks for this purpose, and define an algorithm to generate a network representing the distribution of possible parse trees (up to a specified string length) generated from a given PCFG. Section 3 describes this algorithm, as well as our algorithms for extending the class of queries to include the conditional probability of a symbol appearing anywhere within any region of the parse tree, conditioned on any evidence about symbols appearing in the parse tree.

The restrictive independence assumptions of the PCFG model also limit its applicability, especially in domains like plan recognition and natural language with complex dependency structures. The flexible framework of our Bayesian-network representation supports further extensions to context-sensitive probabilities, as in the probabilistic parse tables of Briscoe and Carroll [8]. Section 4 explores several possible ways to relax the independence assumptions of the PCFG model within our approach. Modified versions of our PCFG algorithms can support the same class of queries supported in the context-free case.

2 Probabilistic Context-Free Grammars

A probabilistic context-free grammar is a tuple $\langle \Sigma, N, S, P \rangle$, where the disjoint sets Σ and N specify the terminal and nonterminal symbols, respectively, with $S \in N$ being the

• *The authors are with the Artificial Intelligence Laboratory, University of Michigan, 1101 Beal Avenue, Ann Arbor, MI 48109.*
E-mail: {pynadath, wellman}@umich.edu.

Manuscript received 18 Nov. 1996; revised 4 Nov. 1997. Recommended for acceptance by T. Ishida.
For information on obtaining reprints of this article, please send e-mail to: tpami@computer.org, and reference IEEECS Log Number 106036.

S	→	np vp	(0.8)	pp	→	prep np	(1.0)
S	→	vp	(0.2)	prep	→	like	(1.0)
np	→	noun	(0.4)	verb	→	swat	(0.2)
np	→	noun pp	(0.4)	verb	→	flies	(0.4)
np	→	noun np	(0.2)	verb	→	like	(0.4)
vp	→	verb	(0.3)	noun	→	swat	(0.05)
vp	→	verb np	(0.3)	noun	→	flies	(0.45)
vp	→	verb pp	(0.2)	noun	→	ants	(0.5)
vp	→	verb np pp	(0.2)				

Fig. 1. A probabilistic context-free grammar (from Charniak [2]).

start symbol. P is the set of productions, which take the form $E \rightarrow \xi(p)$, with $E \in N$, $\xi \in (\Sigma \cup N)^{+}$, and $p = \Pr(E \rightarrow \xi)$, the probability that E will be expanded into the string ξ. The sum of probabilities p over all expansions of a given nonterminal E must be one. The examples in this paper will use the sample grammar (from Charniak [2]) shown in Fig. 1.

This definition of the PCFG model prohibits rules of the form $E \rightarrow \varepsilon$, where ε represents the empty string. However, we can rewrite any PCFG to eliminate such rules and still represent the original distribution [2], as long as we note the probability $\Pr(S \rightarrow \varepsilon)$. For clarity, the algorithm descriptions in this paper assume $\Pr(S \rightarrow \varepsilon) = 0$, but a negligible amount of additional bookkeeping can correct for any nonzero probability.

The probability of applying a particular production $E \rightarrow \xi$ to an intermediate string is conditionally independent of what productions generated this string, or what productions will be applied to the other symbols in the string, given the presence of E. Therefore, the probability of a given derivation is simply the product of the probabilities of the individual productions involved. We define the *parse tree* representation of each such derivation as for nonprobabilistic context-free grammars [9]. The probability of a string in the language is the sum taken over all its possible derivations.

2.1 Standard PCFG Algorithms

Since the number of possible derivations grows exponentially with the string's length, direct enumeration would not be computationally viable. Instead, the standard dynamic programming approach used for both probabilistic and nonprobabilistic CFGs [10] exploits the common production sequences shared across derivations. The central structure is a table, or *chart*, storing previous results for each subsequence in the input sentence. Each entry in the chart corresponds to a subsequence $x_i \cdots x_{i+j-1}$ of the observation string $x_1 \cdots x_L$. For each symbol E, an entry contains the probability that the corresponding subsequence is derived from that symbol, $\Pr(x_i \cdots x_{i+j-1} \mid E)$. The index i refers to the position of the subsequence within the entire terminal string, with $i = 1$ indicating the start of the sequence. The index j refers to the length of the subsequence.

The bottom row of the table holds the results for subsequences of length one, and the top entry holds the overall result, $\Pr(x_1 \cdots x_L \mid S)$, which is the probability of the observed string. We can compute these probabilities bottom-up, since we know that $\Pr(x_i \mid E) = 1$, if E is the observed symbol x_i. We can define all other probabilities recursively as the sum, over all productions $E \rightarrow \xi(p)$, of the product $p \cdot \Pr(x_i \cdots x_{i+j-1} \mid \xi)$. Altering this procedure to take the maximum rather than the sum yields the most probable parse tree for the observed string. Both algorithms require time $O(L^3)$ for a string of length L, ignoring the dependency on the size of the grammar.

To compute the probability of the sentence `Swat flies like ants`, we would use the algorithm to generate the table shown in Fig. 2, after eliminating any unused intermediate entries. There are also separate entries for each production, though this is not necessary if we are interested only in the final sentence probability. In the top entry, there are two listings for the production $S \rightarrow$ `np vp`, with different subsequence lengths for the right-hand side symbols. The sum of all probabilities for productions with S on the left-hand side in this entry yields the total sentence probability of 0.001011.

This algorithm is capable of computing any *inside* probability, the probability of a particular string appearing inside the subtree rooted by a particular symbol. We can work top-down in an analogous manner to compute any *outside* probability [2], the probability of a subtree rooted by a particular symbol appearing amid a particular string. Given these probabilities, we can compute the probability of any particular nonterminal symbol appearing in the parse tree as the root of a subtree covering some subsequence. For example, in the sentence `Swat flies like ants`, we can compute the probability that `like ants` is a prepositional phrase, using a combination of inside and

$j = 4$	$S \rightarrow$ vp: 0.00072; $S \rightarrow$ np(2) vp(2): 0.000035; $S \rightarrow$ np(1) vp(3): 0.000256; vp → verb np pp: 0.0014; vp → verb np: 0.00216			
3		vp → verb pp: 0.016; np → noun pp: 0.036		
2	np → noun np: 0.0018		vp → verb np: 0.024; pp → prep np: 0.2	
1	np → noun: 0.02; verb → swat: 0.2; noun → swat: 0.05	np → noun: 0.18; verb → flies: 0.4; noun → flies: 0.45	prep → like: 1.0; verb→ like: 0.4	np → noun: 0.2; noun → ants: 0.5
	$i = 1$	2	3	4

Fig. 2. Chart for `Swat flies like ants`.

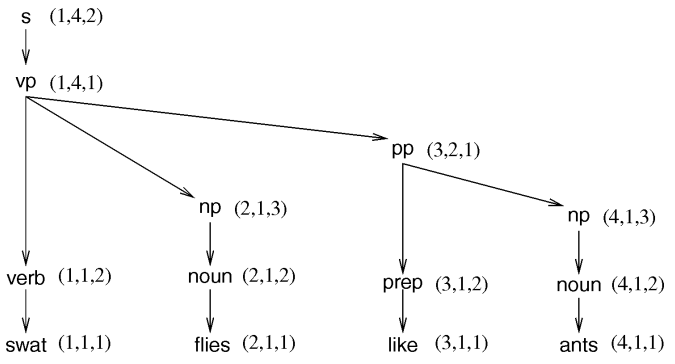

Fig. 3. Parse tree for `Swat flies like ants`, with (*i*, *j*, *k*) indices labeled.

outside probabilities. The Left-to-Right Inside (LRI) algorithm [10] specifies how we can use inside probabilities to obtain the probability of a given initial subsequence, such as the probability of a sentence (of any length) beginning with the words `Swat flies`. Furthermore, we can use such initial subsequence probabilities to compute the conditional probability of the next terminal symbol given a prefix string.

2.2 Indexing Parse Trees

Yet other conceivable queries are not covered by existing algorithms, or answerable via straightforward manipulations of inside and outside probabilities. For example, given observations of arbitrary partial strings, it is unclear how to exploit the standard chart directly. Similarly, we are unaware of methods to handle observation of nonterminals only (e.g., that the last two words form a prepositional phrase). We seek, therefore, a mechanism that would admit observational evidence of any form as part of a query about a PCFG, without requiring us to enumerate all consistent parse trees.

We first require a scheme to specify such events as the appearance of symbols at designated points in the parse tree. We can use the indices i and j to delimit the leaf nodes of the subtree, as in the standard chart parsing algorithms. For example, the `pp` node in the parse tree of Fig. 3 is the root of the subtree whose leaf nodes are `like` and `ants`, so $i = 3$ and $j = 2$.

However, we cannot always uniquely specify a node with these two indices alone. In the branch of the parse tree passing through `np`, `n`, and `flies`, all three nodes have $i = 2$ and $j = 1$. To differentiate them, we introduce the k index, defined recursively. If a node has no child with the same i and j indices, then it has $k = 1$. Otherwise, its k index is one more than the k index of its child. Thus, the `flies` node has $k = 1$, the `noun` node above it has $k = 2$, and its parent `np` has $k = 3$. We have labeled each node in the parse tree of Fig. 3 with its (i, j, k) indices.

We can think of the k index of a node as its level of abstraction, with higher values indicating more abstract symbols. For instance, the `flies` symbol is a specialization of the `noun` concept, which, in turn, is a specialization of the `np` concept. Each possible specialization corresponds to an *abstraction production* of the form $E \rightarrow E'$, that is, with only one symbol on the right-hand side. In a parse tree involving such a production, the nodes for E and E' have identical i and j values, but the k value for E is one more than that of E'. We denote the set of abstraction productions as $P_A \subseteq P$.

All other productions are *decomposition productions*, in the set $P_D = P \backslash P_A$, and have two or more symbols on the right-hand side. If a node E is expanded by a decomposition production, the sum of the j values for its children will equal its own j value, since the length of the original subsequence derived from E must equal the total lengths of the subsequences of its children. In addition, since each child must derive a string of nonzero length, no child has the same j index as E, which must then have $k = 1$. Therefore, abstraction productions connect nodes whose indices match in the i and j components, while decomposition productions connect nodes whose indices differ.

3 Bayesian Networks for PCFGs

A *Bayesian network* [11], [12], [13] is a directed acyclic graph where nodes represent random variables, and associated with each node is a specification of the distribution of its variable conditioned on its predecessors in the graph. Such a network defines a joint probability distribution—the probability of an assignment to the random variables is given by the product of the probabilities of each node conditioned on the values of its predecessors according to the assignment. Edges not included in the graph indicate conditional independence; specifically, each node is conditionally independent of its nondescendants given its immediate predecessors. Algorithms for inference in Bayesian networks exploit this independence to simplify the calculation of arbitrary conditional probability expressions involving the random variables.

By expressing a PCFG in terms of suitable random variables structured as a Bayesian network, we could in principle support a broader class of inferences than the standard PCFG algorithms. As we demonstrate below, by expressing the distribution of parse trees for a given probabilistic grammar, we can incorporate partial observations of a sentence as well as other forms of evidence, and determine the resulting probabilities of various features of the parse trees.

3.1 PCFG Random Variables

We base our Bayesian-network encoding of PCFGs on the scheme for indexing parse trees presented in Section 2.2. The random variable N_{ijk} denotes the symbol in the parse tree at the position indicated by the (i, j, k) indices. Looking back at the example parse tree of Fig. 3, a symbol E labeled (i, j, k) indicates that $N_{ijk} = E$. Index combinations not appearing in the tree correspond to N variables taking on the null value `nil`.

Assignments to the variables N_{ijk} are sufficient to describe a parse tree. However, if we construct a Bayesian network using only these variables, the dependency structure would be quite complicated. For example, in the example PCFG, the fact that N_{213} has the value `np` would influence whether N_{321} takes on the value `pp`, even given that N_{141} (their parent in the parse tree) is `vp`. Thus, we would need an additional link between N_{213} and N_{321}, and, in fact, between all possible sibling nodes whose parents have multiple expansions.

To simplify the dependency structure, we introduce random variables P_{ijk} to represent the productions that expand the corresponding symbols N_{ijk}. For instance, we add the node P_{141}, which would take on the value `vp` $\rightarrow$ `verb np pp` in the example. N_{213} and N_{321} are conditionally independent given P_{141}, so no link between siblings is necessary in this case.

However, even if we know the production P_{ijk}, the corresponding children in the parse tree may not be conditionally independent. For instance, in the chart of Fig. 2, entry (1, 4) has two separate probability values for the production $S \rightarrow$ `np vp`, each corresponding to different subsequence lengths for the symbols on the right-hand side. Given only the production used, there are again multiple possibilities for the connected N variables: N_{113} = `np` and N_{231} = `vp`, or N_{121} = `np` and N_{321} = `vp`. All four of these sibling nodes are conditionally dependent, since knowing any one determines the values of the other three. Therefore, we dictate that each variable P_{ijk} take on different values for each breakdown of the right-hand symbols' subsequence lengths.

The domain of each P_{ijk} variable therefore consists of productions, augmented with the j and k indices of each of the symbols on the right-hand side. In the previous example, the domain of P_{141} would require two possible values, $S \rightarrow$ `np`[1, 3]`vp`[3, 1] and $S \rightarrow$ `np`[2, 1]`vp`[2, 1], where the numbers in brackets correspond to the j and k values, respectively, of the associated symbol. If we know that P_{141} is the former, then N_{113} = `np` and N_{231} = `vp` with probability one. This deterministic relationship renders the child N variables conditionally independent of each other given P_{ijk}. We describe the exact nature of this relationship in Section 3.3.2.

Having identified the random variables and their domains, we complete the definition of the Bayesian network by specifying the conditional probability tables representing their interdependencies. The tables for the N variables represent their deterministic relationship with the parent P variables. However, we also need the conditional probability of each P variable given the value of the corresponding N variable, that is, $\Pr(P_{ijk} = E \rightarrow E_1[j_1, k_1] \cdots E_m[j_m, k_m] \mid N_{ijk} = E)$. The PCFG specifies the relative probabilities of different productions for each nonterminal, but we must compute the probability, $\beta(E, j, k)$ (analogous to the inside probability [2]), that each symbol E_t on the right-hand side is the root node of a subtree, at abstraction level k_t, with a terminal subsequence length j_t.

3.2 Calculating β

3.2.1 Algorithm

We can calculate the values for β with a modified version of the dynamic programming algorithm sketched in Section 2.1. As in the standard chart-based PCFG algorithms, we can define this function recursively and use dynamic programming to compute its values. Since terminal symbols always appear as leaves of the parse tree, we have, for any terminal symbol $x \in \Sigma$, $\beta(x, 1, 1) = 1$, and for any $j > 1$ or $k > 1$, $\beta(x, j, k) = 0$. For any nonterminal symbol $E \in N$, $\beta(E, 1, 1) = 0$, since nonterminals can never be leaf nodes. For $j > 1$ or $k > 1$, $\beta(E, j, k)$ is the sum, over all productions expanding E, of the probability of that production expanding E and producing a subtree constrained by the parameters j and k.

For $k > 1$, only abstraction productions are possible. For an abstraction production $E \rightarrow E'$, we need the probabilities that E is expanded into E' and that E' derives a string of length j from the abstraction level immediately below E. The former is given by the probability associated with the production, while the latter is simply $\beta(E', j, k - 1)$. According to the independence assumptions of the PCFG model, the expansion of E' is independent of its derivation, so the joint probability is simply the product. We can compute these probabilities for every abstraction production expanding E. Since the different expansions are mutually exclusive events, the value for $\beta(E, j, k)$ is merely the sum of all the separate probabilities.

We assume that there are no abstraction cycles in the grammar. That is, there is no sequence of productions $E_1 \rightarrow E_2, \ldots, E_{t-1} \rightarrow E_t, E_t \rightarrow E_1$, since, if such a cycle existed, the above recursive calculation would never halt. The same assumption is necessary for termination of the standard parsing algorithm. The assumption does restrict the classes of grammars for which such algorithms are applicable, but it will not be restrictive in domains where we interpret productions as specializations, since cycles would render an abstraction hierarchy impossible.

For $k = 1$, only decomposition productions are possible. For a decomposition production $E \rightarrow E_1E_2 \cdots E_m(p)$, we need the probability that E is thus expanded, and that each E_t derives a subsequence of appropriate length. Again, the former is given by p, and the latter can be computed from values of the β function. We must consider every possible subsequence length j_t for each E_t, such that $\sum_{t=1}^{m} j_t = j$. In addition, the E_t could appear at any level of abstraction k_t, so we must consider all possible values for a given subsequence length. We can obtain the joint probability of any combination of $\left\{(j_t, k_t)\right\}_{t=1}^{m}$ values by computing $\prod_{t=1}^{m} \beta(E_t, j_t, k_t)$, since the derivation from each E_t is independent of the others. The sum of these joint probabilities over all possible $\left\{(j_t, k_t)\right\}_{t=1}^{m}$ yields the probability of the expansion specified by the production's right-hand side. The product of the resulting probability and p yields the probability of that particular expansion, since the two events are independent. Again, we can sum over all relevant decomposition productions to find the value of $\beta(E, j, 1)$.

The algorithm in Fig. 4 takes advantage of the division between abstraction and decomposition productions to compute the values $\beta(E, j, k)$ for strings bounded by *length*. The array *kmax* keeps track of the depth of the abstraction hierarchy for each subsequence length.

3.2.2 Example Calculations

To illustrate the computation of β values, consider the result of using Charniak's grammar from Fig. 1 as its input. We initialize the entries for $j = 1$ and $k = 1$ to have probability one for each terminal symbol, as in Fig. 1. To fill in the entries for $j = 1$ and $k = 2$, we look at all of the abstraction productions. The symbols `noun`, `verb`, and `prep` can all be

```
COMPUTE-BETA(grammar,length)
  for each symbol $x$ ∈TERMINALS(grammar)
    $\beta[x,1,1] \leftarrow 1.0$
  for $j \leftarrow 1$ to length
    $kmax[j] \leftarrow 0$
    for each symbol $E$ ∈NONTERMINALS(grammar)
      $\beta[E,j,1] \leftarrow 0.0$
    if $j > 1$
      then
        /* Decomposition phase */
        for each production
            $E \rightarrow E_1 \cdots E_m$ $(p)$ ∈DECOMP-PRODS(grammar)
          for each sequence $\{j_t\}_{t=1}^{m}$ such that $\sum_t j_t = j$
            for each sequence $\{k_t\}_{t=1}^{m}$ such that $1 \le k_t \le kmax[j_t]$
              result ← $p$
              for $t \leftarrow 1$ to $m$
                result ← result·$\beta[E_t, j_t, k_t]$
              $\beta[E,j,1] \leftarrow \beta[E,j,1]$+result
    /* Abstraction Phase */
    while $\beta[E',j,kmax[j]+1] > 0$ for some $E'$
      $kmax[j] \leftarrow kmax[j]+1$
      for each production $E \rightarrow E'$ $(p)$ ∈ABSTRACT-PRODS(grammar)
        if $\beta[E',j,kmax[j]] > 0$
          then $\beta[E,j,kmax[j]+1] \leftarrow \beta[E,j,kmax[j]+1] + p \cdot \beta[E',j,kmax[j]]$
  return $\beta$, kmax
```

Fig. 4. Algorithm for computing β values.

k	E	$\beta(E, 4, k)$	k	E	$\beta(E, 3, k)$	k	E	$\beta(E, 2, k)$	k	E	$\beta(E, 1, k)$
2	S	0.02016	2	S	0.0208	2	S	0.024	4	S	0.06
1	S	0.0832	1	S	0.0576	1	S	0.096	3	np	0.4
	np	0.0672		np	0.176		np	0.08		vp	0.3
	vp	0.1008		vp	0.104		vp	0.12	2	prep	1.0
	pp	0.176		pp	0.08		pp	0.4		noun	1.0
										verb	1.0
									1	like	1.0
										swat	1.0
										flies	1.0
										ants	1.0

Fig. 5. Final table for sample grammar.

expanded into one or more terminal symbols, which have nonzero β values at $k = 1$. We enter these three nonterminals at $k = 2$, with β values equal to the sum, over all relevant abstraction productions, of the product of the probability of the given production and the value for the right-hand symbol at $k = 1$. For instance, we compute the value for noun by adding the product of the probability of noun $\rightarrow$ swat and the value for swat, that of noun $\rightarrow$ flies and flies, and that of noun $\rightarrow$ ants and ants. This yields the value one, since a noun will always derive a string of length one, at a single level abstraction above the terminal string, given this grammar. The abstraction phase continues until we find S at $k = 4$, for which there are no further abstractions, so we go on to $j = 2$ and begin the decomposition phase.

To illustrate the decomposition phase, consider the value for $\beta(S, 3, 1)$. There is only one possible decomposition production, s$\rightarrow$ np vp. However, we must consider two separate cases: when the noun phrase covers two symbols and the verb phrase one, and when the noun phrase covers one and the verb phrase two. At a subsequence length of two, both np and vp have nonzero probability only at the bottom level of abstraction, while, at a length of one, only at the third. So, to compute the probability of the first subsequence length combination, we multiply the probability of the production by β(np, 2, 1) and β(vp, 1, 3). The probability of the second combination is a similar product, and the sum of the two values provides the value to enter for S.

The other abstractions and decompositions proceed along similar lines, with additional summation required when multiple productions or multiple levels of abstraction are possible. The final table is shown in Fig. 5, which lists only the nonzero values.

3.2.3 Complexity

For analysis of the complexity of computing the β values for a given PCFG, it is useful to define d to be the maximum length of possible chains of abstraction productions (i.e., the maximum k value), and m to be the maximum

```
CREATE-NETWORK(grammar,length,$\beta$,kmax)
  if $\beta[S, length, kmax[length]] > 0.0$
    then INSERT-STATE($N_{1(length)kmax[length]}$,$S$)
  if START-PROB($\beta$,kmax,length,kmax[length]$-1$)$> 0.0$
    then INSERT-STATE($N_{1(length)kmax[length]}$,nil*)
  for $j \leftarrow length$ down-to 1
    for $k \leftarrow kmax[j]$ down-to 1
      for $i \leftarrow 1$ to $length-j+1$
        for each symbol $E$ $\in$DOMAIN($N_{ijk}$)
          if $E$ $\in$NONTERMINALS(grammar)
            then if $k > 1$
              then ABSTRACTION-PHASE(grammar,E,i,j,k)
              else DECOMPOSITION-PHASE(grammar,E,i,j,k)
          else START-TREE($\beta$,i,j,k)
```

Fig. 6. Procedure for generating the network.

```
ABSTRACTION-PHASE(grammar,E,i,j,k)
  for each production $E \rightarrow E'$ $(p)$ $\in$ABSTRACT-PRODS(grammar)
    INSERT-STATE($P_{ijk}$,$E \rightarrow E'[j, k-1]$ $(p)$)
    ADD-PARENT($N_{ij(k-1)}$,$P_{ijk}$)
    INSERT-STATE($N_{ij(k-1)}$,$E'$)
```

Fig. 7. Procedure for finding all possible abstraction productions.

```
DECOMPOSITION-PHASE(grammar,E,i,j,k)
  for each production $E \rightarrow E_1E_2\cdots E_m$ $(p)$ $\in$DECOMP-PRODS(grammar)
    for each sequence $\{j_t\}_{t=1}^{m}$ such that $\sum_t j_t = j$
      for each sequence $\{k_t\}_{t=1}^{m}$ such that $1 \le k_t \le kmax[j_t]$
        if $p \cdot \prod_t \beta[E_t, j_t, k_t] > 0$
          then INSERT-STATE($P_{ijk}$,$E \rightarrow E_1[j_1,k_1]\cdots E_m[j_m,k_m]$ $(p)$)
            $start \leftarrow i$
            for $t \leftarrow 1$ to $m$
              ADD-PARENT($N_{(start)j_tk_t}$,$P_{ijk}$)
              INSERT-STATE($N_{(start)j_tk_t}$,$E_t$)
              $start \leftarrow start + j_t$
```

Fig. 8. Procedure for finding all possible decomposition productions.

```
START-TREE($\beta$,i,j,k)
  if $k > 1$
    then if $\beta[S, j, k-1] > 0.0$
      then INSERT-STATE($P_{ijk}$,nil* $\rightarrow S[j, k-1]$)
        ADD-PARENT($N_{ij(k-1)}$,$P_{ijk}$)
        INSERT-STATE($N_{ij(k-1)}$,$S$)
    else if $\beta[S, j-1, kmax[j-1]] > 0.0$
      then INSERT-STATE($P_{ijk}$,nil* $\rightarrow S[j-1, kmax[j-1]]$)
        ADD-PARENT($N_{i(j-1)kmax[j-1]}$,$P_{ijk}$)
        INSERT-STATE($N_{i(j-1)kmax[j-1]}$,$S$)
  if START-PROB($\beta$,kmax,j,k)$> 0.0$
    then INSERT-STATE($P_{ijk}$,nil* $\rightarrow$nil*)
      if $k > 1$
        then ADD-PARENT($N_{ij(k-1)}$,$P_{ijk}$)
          INSERT-STATE($N_{ij(k-1)}$,nil*)
        else ADD-PARENT($N_{i(j-1)kmax[j-1]}$,$P_{ijk}$)
          INSERT-STATE($N_{i(j-1)kmax[j-1]}$,nil*)
```

Fig. 9. Procedure for handling start of parse tree at next level.

```
START-PROB($\beta$,kmax,j,k)
  if $j$=0
    then return 0.0
    else if $k$=0
      then return START-PROB($\beta$,kmax,$j-1$,kmax[$j-1$])
      else return $\beta[S, j, k]$+START-PROB($\beta$,kmax,j,$k-1$)
```

Fig. 10. Procedure for computing the probability of the start of the tree occurring for a particular string length and abstraction level.

production length (number of symbols on the right-hand side). A single run through the abstraction phase requires time $O(|P_A|)$, and for each subsequence length, there are $O(d)$ runs. For a specific value of j, the decomposition phase requires time $O(|P_D|j^{m-1}d^m)$, since, for each decomposition production, we must consider all possible combinations of subsequence lengths and levels of abstractions for each symbol on the right-hand side. Therefore, the whole algorithm would take time $O(n[d|P_A| + |P_D|n^{m-1}d^m]) = O(|P|n^md^m)$.

3.3 Network Generation Phase

We can use the β function calculated as described above to compute the domains of random variables N_{ijk} and P_{ijk} and the required conditional probabilities.

3.3.1 Specification of Random Variables

The procedure CREATE-NETWORK, described in Fig. 6, begins at the top of the abstraction hierarchy for strings of length n starting at position 1. The root symbol variable, $N_{1n(kmax[n])}$, can be either the start symbol, indicating the parse tree begins here, or `nil`*, indicating that the parse tree begins below. We must allow the parse tree to start at any j and k where $\beta(S, j, k) > 0$, because these can all possibly derive strings (of any length bounded by n) within the language.

CREATE-NETWORK then proceeds downward through the N_{ijk} random variables and specifies the domain of their corresponding production variables, P_{ijk}. Each such production variable takes on values from the set of possible expansions for the possible nonterminal symbols in the domain of N_{ijk}. If $k > 1$, only abstraction productions are possible, so the procedure ABSTRACTION-PHASE, described in Fig. 7, inserts all possible expansions and draws links from P_{ijk} to the random variable $N_{ij(k-1)}$, which takes on the value of the right-hand side symbol. If $k = 1$, the procedure DECOMPOSITION-PHASE, described in Fig. 8, performs the analogous task for decomposition productions, except that it must also consider all possible length breakdowns and abstraction levels for the symbols on the right-hand side.

CREATE-NETWORK calls the procedure START-TREE, described in Fig. 9, to handle the possible expansions of `nil`*: either `nil`$^* \rightarrow S$, indicating that the tree starts immediately below, or `nil`$^* \rightarrow$ `nil`*, indicating that the tree starts further below. START-TREE uses the procedure START-PROB, described in Fig. 10, to determine the probability of the parse tree starting anywhere below the current point of expansion.

When we insert a possible value into the domain of a production node, we add it as a parent of each of the nodes corresponding to a symbol on the right-hand side. We also insert each symbol from the right-hand side into the domain of the corresponding symbol variable. The algorithm descriptions assume the existence of procedures INSERT-STATE and ADD-PARENT. The procedure INSERT-STATE(*node*, *label*) inserts a new state with name *label* into the domain of variable *node*. The procedure ADD-PARENT(*child*, *parent*) draws a link from node *parent* to node *child*.

3.3.2 Specification of Conditional Probability Table

After CREATE-NETWORK has specified the domains of all of the random variables, we can specify the conditional probability tables. We introduce the lexicographic order $\prec$ over the set $\{(j, k) \mid 1 \le j \le n, 1 \le k \le kmax[j]\}$, where if $j_1 < j_2$ then $(j_1, k_1) \prec (j_2, k_2)$ and if $k_1 < k_2$ then $(j, k_1) \prec (j, k_2)$. For the purposes of simplicity, we do not specify an exact value for each probability $\Pr(X = x \mid Y)$, but instead specify a weight, $\Pr(X = x_t \mid Y) \propto \alpha_t$. We compute the exact probabilities through normalization, where we divide each weight by the sum $\sum_t \alpha_t$. The prior probability table for the top node, which has no parents, can be defined as follows:

$$\Pr(N_{1n(kmax[n])} = S) \propto \beta(S, n, kmax[n])$$

$$\Pr(N_{1n(kmax[n])} = \texttt{nil}^*) \propto \textstyle\sum_{(j,k) \prec (n, kmax[n])} \beta(S, j, k).$$

For a given state ρ in the domain of any P_{ijk} node, where ρ represents a production and corresponding assignment of j and k values to the symbols on the right-hand side, of the form $E \rightarrow E_1[j_1, k_1] \cdots E_m[j_m, k_m](p)$, we can define the conditional probability of that state as:

$$\Pr\left(P_{ijk} = \rho \middle| N_{ijk} = E\right) \propto p \prod_{t=1}^{m} \beta\left[E_t, j_t, k_t\right].$$

For any symbol $E' \neq E$ in the domain of N_{ijk}, $\Pr(P_{ijk} = \rho \mid N_{ijk} = E') = 0$. For the productions for starting or delaying the tree, the probabilities are:

$$\Pr(P_{1jk} = \texttt{nil}^* \rightarrow S[j', k'] \mid N_{ijk} = \texttt{nil}^*) \propto \beta[S, j', k']$$

$$\Pr(P_{1jk} = \texttt{nil}^* \rightarrow \texttt{nil}^* \mid N_{ijk} = \texttt{nil}^*) \propto \textstyle\sum_{(j',k') \prec (j,k)} \beta[S, j', k'].$$

The probability tables for the N_{ijk} nodes are much simpler, since once the productions are specified, the symbols are completely determined. Therefore, the entries are either one or zero. For example, consider the nodes $N_{i_\ell j_\ell k_\ell}$ with the parent node $P_{i'j'k'}$ (among others). For the rule ρ representing $E \rightarrow E_1[j_1, k_1] \cdots E_m[j_m, k_m]$, $\Pr(N_{i_\ell j_\ell k_\ell} = E_\ell \mid P_{i'j'k'} = \rho, \ldots) = 1$ when $i_\ell = i' + \sum_{t=1}^{\ell-1} j_t$, $j = j_\ell$. For all symbols other than E_ℓ in the domain of $N_{i_\ell j_\ell k_\ell}$, this conditional probability is zero. We can fill in this entry for all configurations of the other parent nodes (represented by the ellipsis in the condition part of the probability), though we know that any conflicting configurations (i.e., two productions both trying to specify the symbol $N_{i_\ell j_\ell k_\ell}$) are impossible. Any configuration of the parent nodes that does not specify a certain symbol indicates that the $N_{i_\ell j_\ell k_\ell}$ node takes on the value `nil` with probability one.

3.3.3 Network Generation Example

As an illustration, consider the execution of this algorithm using the β values from Fig. 5. We start with the root variable N_{142}. The start symbol S has a β value greater than zero here, as well as at points below, so the domain must include both S and $\texttt{nil}^*$. To obtain $\Pr(N_{142} = S)$, we simply divide $\beta(S, 4, 2)$ by the sum of all β values for S, yielding 0.055728.

The domain of P_{142} is partially specified by the abstraction phase for the symbol S in the domain of N_{142}. There is only one relevant production, $S \rightarrow \texttt{vp}$, which is a possible expansion since $\beta(\texttt{vp}, 4, 1) > 0$. Therefore, we insert the production into the domain of P_{142}, with conditional probability one given that $N_{142} = S$, since there are no other possible expansions. We also draw a link from P_{142} to N_{141}, whose domain now includes `vp` with conditional probability one given that $P_{142} = S \rightarrow \texttt{vp}$.

To complete the specification of P_{142}, we must consider the possible start of the tree, since the domain of N_{142} includes $\texttt{nil}^*$. The conditional probability of $P_{142} = \texttt{nil}^* \rightarrow S$ is 0.24356, the ratio of $\beta(S, 4, 1)$ and the sum of $\beta(S, j, k)$ for $(j, k) \preccurlyeq (4, 1)$. The link from P_{142} to N_{141} has already been made during the abstraction phase, but we must also insert S and $\texttt{nil}^*$ into the domain of N_{141}, each with conditional probability one given the appropriate value of P_{142}.

We then proceed to N_{141}, which is at the bottom level of abstraction, so we must perform a decomposition phase. For the production $S \rightarrow \texttt{np}\ \texttt{vp}$, there are three possible combinations of subsequence lengths which add to the total length of four. If `np` derives a string of length one and `vp` a string of length three, then the only possible levels of abstraction for each are three and one, respectively, since all others will have zero β values. Therefore, we insert the production $\texttt{s} \rightarrow \texttt{np}[1, 3]\texttt{vp}[3, 1]$ into the domain of P_{141}, where the numbers in brackets correspond to the subsequence length and level of abstraction, respectively. The conditional probability of this value, given that $N_{141} = S$, is the product of the probability of the production, $\beta(\texttt{np}, 1, 3)$, and $\beta(\texttt{vp}, 3, 1)$, normalized over the probabilities of all possible expansions.

We then draw links from P_{141} to N_{113} and N_{231}, into whose domains we insert `np` and `vp`, respectively. The i values are obtained by noting that the subsequence for `np` begins at the same point as the original string while that for `vp` begins at a point shifted by the length of the subsequence for `np`. Each occurs with probability one, given that the value of P_{141} is the appropriate production. Similar actions are taken for the other possible subsequence length combinations. The operations for the other random variables are performed in a similar fashion, leading to the network structure shown in Fig. 11.

3.3.4 Complexity of Network Generation

The resulting network has $O(n^2 d)$ nodes. The domain of each N_{i11} variable has $O(|\Sigma|)$ states to represent the possible terminal symbols, while all other N_{ijk} variables have $O(|N|)$ possible states. There are n variables of the former, and $O(n^2 d)$ of the latter. For $k > 1$, the P_{ijk} variables (of which there are $O(n^2 d)$) have a domain of $O(|P_A|)$ states. For P_{ij1} variables, there are states for each possible decomposition production, for each possible combination of subsequence lengths, and for each possible level of abstraction of the symbols on the right-hand side. Therefore, the P_{ij1} variables (of which there are $O(n^2)$) have a domain of $O(|P_D| j^{m-1} d^m)$ states, where we have again defined d to be the maximum value of k, and m to be the maximum production length.

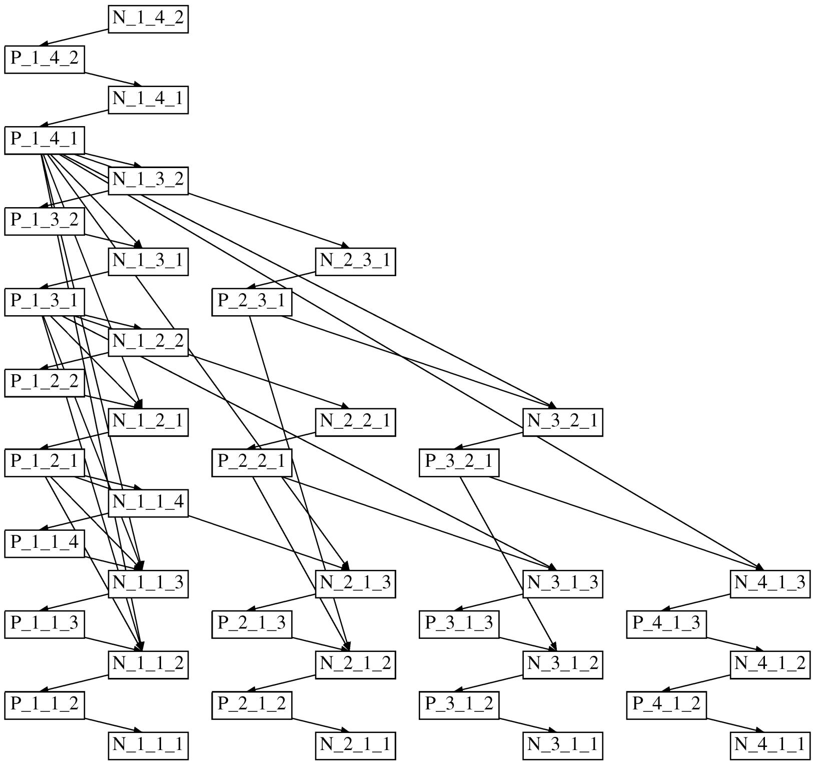

Fig. 11. Network from example grammar at maximum length 4.

Unfortunately, even though each particular P variable has only the corresponding N variable as its parent, a given N variable could have potentially $O(n)$ P variables as parents. The size of the conditional probability table for a node is exponential in the number of parents, although given that each N can be determined by at most one P (i.e., no interactions are possible), we can specify the table in a linear number of parameters.

If we define T to be the maximum number of entries of any conditional probability table in the network, then the abstraction phase of the algorithm requires time $O(|P_A|T)$, while the decomposition phase requires time $O(|P_D|n^{m-1}d^mT^m)$. Handling the start of the parse tree and the potential space holders requires time $O(T)$. The total time complexity of the algorithm is then $O(n^2|P_D|n^{m-1}d^mT^m + ndT + n^2d|P_A|T + n^2dT) = O(|P|n^{m+1}d^mT^m)$, which dwarfs the time complexity of the dynamic programming algorithm for the β function. However, this network is created only once for a particular grammar and length bound.

3.4 PCFG Queries

We can use the Bayesian network to compute any joint probability that we can express in terms of the N and P random variables included in the network. The standard Bayesian network algorithms [11], [12], [14] can return joint probabilities of the form $\Pr\left(X_{i_1j_1k_1} = x_1, \ldots, X_{i_mj_mk_m} = x_m\right)$ or conditional probabilities of the form $\Pr\left(X_{ijk} = x \mid X_{i_1j_1k_1} = x_1, \ldots, X_{i_mj_mk_m} = x_m\right)$, where each X is either N or P. Obviously, if we are interested only in whether a symbol E appeared at a particular i, j, k location in the parse tree, we need only examine the marginal probability distribution of the corresponding N variable. Thus, a single network query will yield the probability $\Pr(N_{ijk} = E)$.

The results of the network query are implicitly conditional on the event that the length of the terminal string does not exceed n. We can obtain the joint probability by multiplying the result by the probability that a string in the language has a length not exceeding n. For any j, the probability that we expand the start symbol S into a terminal string of length j is $\sum_{k=1}^{kmax[j]} \beta(S, j, k)$, which we can then sum for $1 \le j \le n$. To obtain the appropriate unconditional probability for any query, all network queries reported in this section must be multiplied by $\sum_{j=1}^{n} \sum_{k=1}^{kmax[j]} \beta(S, j, k)$.

3.4.1 Probability of Conjunctive Events

The Bayesian network also supports the computation of joint probabilities analogous to those computed by the standard PCFG algorithms. For instance, the probability of a particular terminal string such as `Swat flies like ants` corresponds to the probability $\Pr(N_{111} = \texttt{swat}, N_{211} = \texttt{flies}, N_{311} = \texttt{like}, N_{411} = \texttt{ants})$. The probability of an initial subsequence like `Swat flies...`, as computed by the LRI algorithm [10], corresponds to the probability $\Pr(N_{111} = \texttt{swat}, N_{211} = \texttt{like})$. Since the Bayesian network represents the distribution over strings of bounded length, we can find initial subsequence probabilities only over completions of length bounded by $n - L$.

Although, in this case, our Bayesian network approach requires some modification to answer the same query as the standard PCFG algorithm, it needs no modification to handle more complex types of evidence. The chart parsing and LRI algorithms require complete sequences as input, so any gaps or other uncertainty about particular symbols would require direct modification of the dynamic programming algorithms to compute the desired probabilities. The Bayesian network, on the other hand, supports the computation of the probability of any evidence, regardless of its structure. For instance, if we have a sentence `Swat flies ... ants` where we do not know the third word, a single network query will provide the conditional probability of possible completions $\Pr(N_{311} \mid N_{111} = \texttt{swat}, N_{211} = \texttt{flies}, N_{411} = \texttt{ants})$, as well as the probability of the specified evidence $\Pr(N_{111} = \texttt{swat}, N_{211} = \texttt{flies}, N_{411} = \texttt{ants})$.

This approach can handle multiple gaps, as well as partial information. For example, if we again do not know the exact identity of the third word in the sentence `Swat flies ... ants`, but we do know that it is either `swat` or `like`, we can use the Bayesian network to fully exploit this partial information by augmenting our query to specify that any domain values for N_{311} other than `swat` or `like` have zero probability. Although these types of queries are rare in natural language, domains like speech recognition often require this ability to reason when presented with noisy observations.

We can answer queries about nonterminal symbols as well. For instance, if we have the sentence `Swat flies like ants`, we can query the network to obtain the conditional probability that `like ants` is a prepositional phrase, $\Pr(N_{321} = \texttt{pp} \mid N_{111} = \texttt{swat}, N_{211} = \texttt{flies}, N_{311} = \texttt{like}, N_{411} = \texttt{ants})$. We can answer queries where we specify evidence about nonterminals within the parse tree. For instance, if we know that `like ants` is a prepositional phrase, the input to the network query will specify that $N_{321} = \texttt{pp}$, as well as specifying the terminal symbols.

Alternate network algorithms can compute the most probable state of the random variables given the evidence, instead of a conditional probability [11], [15], [14]. For example, consider the case of possible four-word sentences beginning with the phrase `Swat flies ...`. The probability maximization network algorithms can determine that the most probable state of terminal symbol variables N_{311} and N_{411} is `like flies`, given that $N_{111} = \texttt{swat}$, $N_{211} = \texttt{flies}$, and $N_{511} = \texttt{nil}$.

3.4.2 Probability of Disjunctive Events

We can also compute the probability of disjunctive events through multiple network queries. If we can express an event as the union of mutually exclusive events, each of the form $X_{i_1 j_1 k_1} = x_1 \wedge \cdots \wedge X_{i_m j_m k_m} = x_m$, then we can query the network to compute the probability of each, and sum the results to obtain the probability of the union. For instance, if we want to compute the probability that the sentence `Swat flies like ants` contains any prepositions, we would query the network for the probabilities $\Pr(N_{i12} = \texttt{prep} \mid N_{111} = \texttt{swat}, N_{211} = \texttt{like}, N_{311} = \texttt{like}, N_{411} = \texttt{ants})$, for $1 \le i \le 4$. In a domain like plan recognition, such a query could correspond to the probability that an agent performed some complex action within a specified time span.

In this example, the individual events are already mutually exclusive, so we can sum the results to produce the overall probability. In general, we ensure mutual exclusivity of the individual events by computing the conditional probability of the conjunction of the original query event and the negation of those events summed previously. For our example, the overall probability would be $\Pr(N_{112} = \texttt{prep} \mid \varepsilon) + \Pr(N_{212} = \texttt{prep}, N_{112} \neq \texttt{prep} \mid \varepsilon) + \Pr(N_{312} = \texttt{prep}, N_{112} \neq \texttt{prep}, N_{212} \neq \texttt{prep} \mid \varepsilon) + \Pr(N_{412} = \texttt{prep}, N_{112} \neq \texttt{prep}, N_{212} \neq \texttt{prep}, N_{312} \neq \texttt{prep} \mid \varepsilon)$, where ε corresponds to the event that the sentence is `Swat flies like ants`.

The Bayesian network provides a unified framework that supports the computation of all of the probabilities described here. We can compute the probability of any event ε, where ε is a set of mutually exclusive events $\left\{X_{i_{t1} j_{t1} k_{t1}} \in \mathcal{X}_{t_1} \wedge \cdots \wedge X_{i_{tm_t} j_{tm_t} k_{tm_t}} \in \mathcal{X}_{t_{m_t}}\right\}_{t=1}^{h}$ with each X being either N or P. We can also compute probabilities of events where we specify relative likelihoods instead of strict subset restrictions. In addition, given any such event, we can determine the most probable configuration of the uninstantiated random variables. Instead of designing a new algorithm for each such query, we have only to express the query in terms of the network's random variables, and use any Bayesian network algorithm to compute the desired result.

3.4.3 Complexity of Network Queries

Unfortunately, the time required by the standard network algorithms in answering these queries is potentially exponential in the maximum string length n, though the exact complexity will depend on the connectedness of the network and the particular network algorithm chosen. The algorithm in our current implementation uses a great deal of preprocessing in compiling the networks, in the hope of reducing the complexity of answering queries. Such an algorithm can exploit the regularities of our networks (e.g., the conditional probability tables of each N_{ijk} consist of only zeroes and ones) to provide reasonable response time in answering queries. Unfortunately, such compilation can itself be prohibitive and will often produce networks of exponential size. There exist Bayesian network algorithms [16], [17] that offer greater flexibility in compilation, possibly allowing us to limit the size of the resulting networks, while still providing acceptable query response times.

Determining the optimal tradeoff will require future research, as will determining the class of domains where our Bayesian network approach is preferable to existing PCFG algorithms. It is clear that the standard dynamic programming algorithms are more efficient for the PCFG queries they address. For domains requiring more general queries of the types described here, the flexibility of the Bayesian network approach may justify the greater complexity.

4 Context Sensitivity

For many domains, the independence assumptions of the PCFG model are overly restrictive. By definition, the probability of applying a particular PCFG production to expand a given nonterminal is independent of what symbols have come before and of what expansions are to occur after. Even this paper's simplified example illustrates some of the weaknesses of this assumption. Consider the intermediate string `Swat ants like noun`. It is implausible that the probability that we expand `noun` into `flies` instead of `ants` is independent of the choice of `swat` as the verb or the choice of `ants` as the object.

Of course, we may be able to correct the model by expanding the set of nonterminals to encode contextual information, adding productions for each such expansion, and thus preserving the structure of the PCFG model. However, this can obviously lead to an unsatisfactory increase in complexity for both the design and use of the model. Instead, we could use an alternate model which relaxes the PCFG independence assumptions. Such a model would need a more complex production and/or probability structure to allow complete specification of the distribution, as well as modified inference algorithms for manipulating this distribution.

4.1 Direct Extensions to Network Structure

The Bayesian network representation of the probability distribution provides a basis for exploring such context sensitivities. The networks generated by the algorithms of this paper implicitly encode the PCFG assumptions through assignment of a single nonterminal node as the parent of each production node. This single link indicates that the expansion is conditionally independent of all other nondescendant nodes, once we know the value of this nonterminal. We could extend the context-sensitivity of these expansions within our network formalism by altering the links associated with these production nodes.

We can introduce some context sensitivity even without adding any links. Since each production node has its own conditional probability table, we can define the production probabilities to be a function of the (i, j, k) index values. For instance, the number of words in a group strongly influences the likelihood of that group forming a noun phrase. We could model such a belief by varying the probability of a `np` appearing over different string lengths, as encoded by the j index. In such cases, we can modify the standard PCFG representation so that the probability information associated with each production is a function of i, j, and k, instead of a constant. The dynamic programming algorithm of Fig. 4 can be easily modified to handle production probabilities that depend on j and k. However, a dependency on the i index as well would require adding it as a parameter of β and introducing an additional loop over its possible values. Then, we would have to replace any reference to the production probability, in either the dynamic programming or network generation algorithm, with the appropriate function of i, j, and k.

Alternatively, we may introduce additional dependencies on other nodes in the network. A PCFG extension that conditions the production probabilities on the parent of the left-hand side symbol has already proved useful in modeling natural language [18]. In this case, each production has a set of associated probabilities, one for each nonterminal symbol that is a possible parent of the symbol on the left-hand side. This new probability structure requires modifications to both the dynamic programming and the network generation algorithms. We must first extend the probability information of the β function to include the parent nonterminal as an additional parameter. It is then straightforward

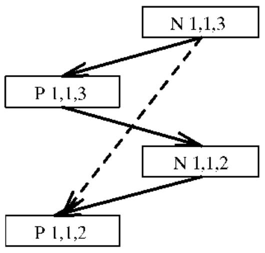

Fig. 12. Subnetwork incorporating parent symbol dependency.

to alter the dynamic programming algorithm of Fig. 4 to correctly compute the probabilities in a bottom-up fashion.

The modifications for the network generation algorithm are more complicated. Whenever we add P_{ijk} as a parent for some symbol node $N_{\hat{i}\hat{j}\hat{k}}$, we also have to add N_{ijk} as a parent of $P_{\hat{i}\hat{j}\hat{k}}$. For example, the dotted arrow in the subnetwork of Fig. 12 represents the additional dependency of P_{112} on N_{113}. We must add this link because N_{112} is a possible child nonterminal, as indicated by the link from P_{113}. The conditional probability tables for each P node must now specify probabilities given the current nonterminal and the parent nonterminal symbols. We can compute these by combining the modified β values with the conditional production probabilities.

Returning to the example from the beginning of this section, we may want to condition the production probabilities on the terminal string expanded so far. As a first approximation to such context sensitivity, we can imagine a model where each production has an associated set of probabilities, one for each terminal symbol in the language. Each represents the conditional probability of the particular expansion given that the corresponding terminal symbol occurs immediately previous to the subsequence derived from the nonterminal symbol on the left-hand side. Again, our β function requires an additional parameter, and we need a modified version of the dynamic programming algorithm to compute its values. However, the network generation algorithm needs to introduce only one additional link, from N_{i11} for each $P_{(i+1)jk}$ node. The dashed arrows in the subnetwork of Fig. 13 reflect the additional dependencies introduced by this context sensitivity, using the network example from Fig. 11. The P_{1jk} nodes are a special case, with no preceding terminal, so the steps from the original algorithm are sufficient.

We can extend this conditioning to cover preceding terminal *sequences* rather than individual symbols. Each production could have an associated set of probabilities, one for each possible terminal sequence of length bounded by some parameter h. The β function now requires an additional parameter specifying the preceding sequence. The network generation algorithms must then add links to P_{ijk} from nodes $N_{(i-h)11}, \ldots, N_{(i-1)11}$, if $i \geq h$, or from $N_{111}, \ldots, N_{(i-1)11}$, if $i < h$. The conditional probability tables then specify the probability of a particular expansion given the symbol on the left-hand side and the preceding terminal sequence.

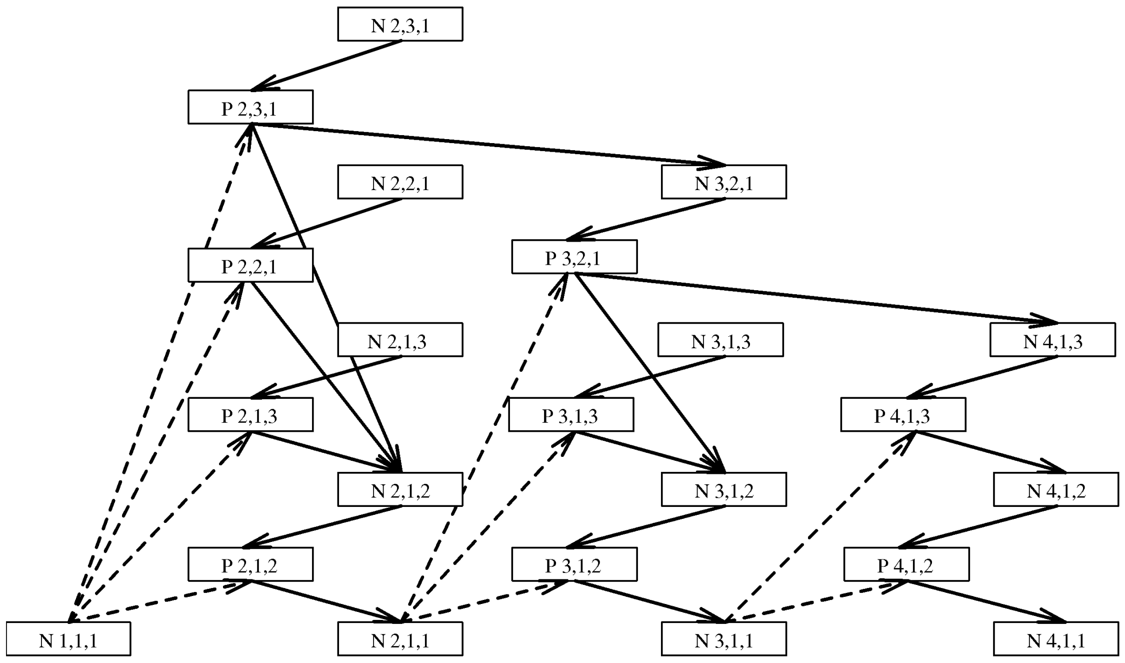

Fig. 13. Subnetwork capturing dependency on previous terminal symbol.

In many cases, we may wish to account for external influences, such as explicit context representation in natural language problems or influences of the current world state in planning, as required by many plan recognition problems [19]. For instance, if we are processing multiple sentences, we may want to draw links from the symbol nodes of one sentence to the production nodes of another, to reflect thematic connections. As long as our network can include random variables to represent the external context, then we can represent the dependency by adding links from the corresponding nodes to the appropriate production nodes and altering the conditional probability tables to reflect the effect of the context.

In general, the Bayesian networks currently generated contain a set of random variables sufficient for expressing arbitrary parse tree events, so we can introduce context sensitivity by adding the appropriate links to the production nodes from the events on which we wish to condition expansion probabilities. Once we have the correct network, we can use any of the query algorithms from Section 3.4 to produce the corresponding conditional probability.

4.2 Extensions to the Grammar Model

Context sensitivities expressed as incremental changes to the network dependency structure represent only a minor relaxation of the conditional independence assumptions of the PCFG model. More global models of context sensitivity will likely require a radically different grammatical form and probabilistic interpretation framework. The History-Based Grammar (HBG) [20] provides a rich model of context sensitivity by conditioning the production probabilities on (potentially) the entire parse tree available at the current expansion point. Since our Bayesian networks represent all positions of the parse tree, it is theoretically possible to represent these conditional probabilities by introducing the appropriate links. However, since the HBG model uses decision tree methods to identify equivalence classes of the partial trees and thus produce simple event structures to condition on, it is unclear exactly how to replicate this behavior in a systematic generation algorithm.

If we restrict the types of context sensitivity, then we are more likely to find such a network generation algorithm. In the nonstochastic case, *context-sensitive grammars* [9] provide a more structured model than the general unrestricted grammar by allowing only productions of the form $\alpha_1 A \alpha_2 \to \alpha_1 B \alpha_2$, where the αs are arbitrary sequences of terminal and/or nonterminal symbols. This restriction eliminates productions where the right-hand side is shorter then the left-hand side. Such a production indicates that A can be expanded into B only when it appears in the surrounding context of α_1 immediately precedent and α_2 immediately subsequent. Therefore, perhaps an extension to a *probabilistic* context-sensitive grammar (PCSG), similar to that for PCFGs, could provide an even richer model for the types of conditional probabilities briefly explored here.

The intuitive extension involves associating a likelihood weighting with each context-sensitive production and computing the probability of a particular derivation based on these weights. These weights cannot correspond to probabilities, because we do not know, a priori, which expansions may be applicable at a given point in the parse (due to the different possible contexts). Therefore, a set of fixed production values may not produce weights that sum to one in a particular context. We can instead use these weights to determine probabilities after we know which productions are applicable. The probability of a particular derivation sequence is then uniquely determined, though it could be sensitive to the order in which we apply the productions. We could then define a probability distribution over all strings in the context-sensitive language so that the probability of a particular string is the sum of the probabilities over all possible derivation sequences for that string.

This definition appears theoretically sound, though it is unclear whether any real-world domains exist for which such a model would be useful. If we create such a model, we should be able to generate a Bayesian network with the proper conditional dependency structure to represent the distribution. We would have to draw links to each production node from its potential context nodes, and the conditional probability tables would reflect the production weights in each particular context possibility. It is an open question whether we could create a systematic generation algorithm similar to that defined for PCFGs.

Although the proposed PCSG model cannot account for dependence on position or parent symbol, described earlier in this section, we could make similar extensions to account for these types of dependencies. The result would be similar to the context-sensitive probabilities of PEARL [21]. However, PEARL conditions the probabilities on a part-of-speech trigram, as well as on the sibling and parent nonterminal symbols. If we allow our model to specify conjunctions of contexts, then it may be able to represent these same types of probabilities, as well as more general contexts beyond siblings and trigrams.

It is clearly difficult to select a model powerful enough to encompass a significant set of useful dependencies, but restricted enough to allow easy specification of the productions and probabilities for a particular language. Once we have chosen a grammatical formalism capable of representing the context sensitivities we wish to model, we must define a network generation algorithm to correctly specify the conditional probabilities for each production node. However, once we have the network, we can again use any of the query algorithms from Section 3.4. Thus, we have a unified framework for performing inference, regardless of the form of the language model used to generate the networks.

Probabilistic parse tables [8] and stochastic programs [22] provide alternate frameworks for introducing context sensitivity. The former approach uses the finite-state machine of the chart parser as the underlying structure and introduces context sensitivity into the transition probabilities. Stochastic programs can represent very general stochastic processes, including PCFGs, and their ability to maintain arbitrary state information could support general context sensitivity as well. It is unclear whether any of these approaches have advantages of generality or efficiency over the others.

5 CONCLUSION

The algorithms presented here automatically generate a Bayesian network representing the distribution over all parses of strings (bounded in length by some parameter) in the language of a PCFG. The first stage uses a dynamic programming approach similar to that of standard parsing algorithms, while the second stage generates the network, using the results of the first stage to specify the probabilities. This network is generated only once for a particular PCFG and length bound. Once created, we can use this network to answer a variety of queries about possible strings and parse trees. Using the standard Bayesian network inference algorithms, we can compute the conditional probability or most probable configuration of any collection of our basic random variables, given any other event which can be expressed in terms of these variables.

These algorithms have been implemented and tested on several grammars, with the results verified against those of existing dynamic programming algorithms when applicable, and against enumeration algorithms when given nonstandard queries. When answering standard queries, the time requirements for network inference were comparable to those for the dynamic programming techniques. Our network inference methods achieved similar response times for some other types of queries, providing a vast improvement over the much slower brute force algorithms. However, in our current implementation, the memory requirements of network compilation limit the complexity of the grammars and queries, so it is unclear whether these results will hold for larger grammars and string lengths.

Preliminary investigation has also demonstrated the usefulness of the network formalism in exploring various forms of context-sensitive extensions to the PCFG model. Relatively minor modifications to the PCFG algorithms can generate networks capable of representing the more general dependency structures required for certain context sensitivities, without sacrificing the class of queries that we can answer. Future research will need to provide a more general model of context sensitivity with sufficient structure to support a corresponding network generation algorithm.

Although answering queries in Bayesian networks is exponential in the worst case, our method incurs this cost in the service of greatly increased generality. Our hope is that the enhanced scope will make PCFGs a useful model for plan recognition and other domains that require more flexibility in query forms and in probabilistic structure. In addition, these algorithms may extend the usefulness of PCFGs in natural language processing and other pattern recognition domains where they have already been successful.

ACKNOWLEDGMENTS

We are grateful to the anonymous reviewers for careful reading and helpful suggestions. This work was supported in part by Grant F49620-94-1-0027 from the Air Force Office of Scientific Research.

REFERENCES

[1] R.C. Gonzalez and M.S. Thomason, *Syntactic Pattern Recognition: An Introduction*. Reading, Mass.: Addison-Wesley, 1978.

[2] E. Charniak, *Statistical Language Learning*. Cambridge, Mass.: MIT Press, 1993.

[3] C.S. Wetherell, “Probabilistic Languages: A Review and Some Open Questions,” *Computing Surveys*, vol. 12, no. 4, pp. 361-379, 1980.

[4] P.A. Chou, “Recognition of Equations Using a Two-Dimensional Stochastic Context-Free Grammar,” *Proc. SPIE: Visual Communications and Image Processing IV, Int’l Soc. Optical Eng.*, pp. 852-863, Bellingham, Wash., 1989.

[5] H. Ney, “Stochastic Grammars and Pattern Recognition,” *Speech Recognition and Understanding*, P. Laface and R. DeMori, eds., pp. 319-344. Berlin: Springer, 1992.

[6] Y. Sakakibara, M. Brown, R.C. Underwood, I.S. Mian, and D. Haussler, “Stochastic Context-Free Grammars for Modeling RNA,” *Proc. 27th Hawaii Int’l Conf. System Sciences*, pp. 284-293, 1995.

[7] M. Vilain, “Getting Serious About Parsing Plans: A Grammatical Analysis of Plan Recognition,” *Proc. Eighth Nat’l Conf. Artificial Intelligence*, pp. 190-197, 1990.

[8] T. Briscoe and J. Carroll, “Generalized Probabilistic LR Parsing of Natural Language (Corpora) With Unification-Based Grammars,” *Computational Linguistics*, vol. 19, no. 1, pp. 25-59, Mar. 1993.

[9] J.E. Hopcroft and J.D. Ullman, *Introduction to Automata Theory, Languages, and Computation*. Reading, Mass.: Addison-Wesley, 1979.

[10] F. Jelinek, J.D. Lafferty, and R.L. Mercer, “Basic Methods of Probabilistic Context Free Grammars,” *Speech Recognition and Understanding*, P. Laface and R. DeMori, eds., pp. 345-360. Berlin: Springer, 1992.

[11] J. Pearl, *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. San Mateo, Calif.: Morgan Kaufmann, 1987.

[12] R.E. Neapolitan, *Probabilistic Reasoning in Expert Systems: Theory and Algorithms*. New York: John Wiley and Sons, 1990.

[13] F.V. Jensen, *An Introduction to Bayesian Networks*. New York: Springer, 1996.

[14] R. Dechter, “Bucket Elimination: A Unifying Framework for Probabilistic Inference,” *Proc. 12th Conf. Uncertainty in Artificial Intelligence*, pp. 211-219, San Francisco, 1996.

[15] E. Charniak and S.E. Shimony, “Cost-Based Abduction and MAP Explanation,” *Artificial Intelligence*, vol. 66, pp. 345-374, 1994.

[16] R. Dechter, “Topological Parameters for Time-Space Tradeoff,” *Proc. 12th Conf. Uncertainty in Artificial Intelligence*, pp. 220-227, San Francisco, 1996.

[17] A. Darwiche and G. Provan, “Query DAGs: A Practical Paradigm for Implementing Belief-Network Inference,” *J. Artificial Intelligence Research*, vol. 6, pp. 147-176, 1997.

[18] E. Charniak and G. Carroll, “Context-Sensitive Statistics for Improved Grammatical Language Models,” *Proc. 12th Nat’l Conf. Artificial Intelligence*, pp. 728-733, Menlo Park, Calif., 1994.

[19] D.V. Pynadath and M.P. Wellman, “Accounting for Context in Plan Recognition, With Application to Traffic Monitoring,” *Proc. 11th Conf. Uncertainty in Artificial Intelligence*, pp. 472-481, San Francisco, 1995.

[20] E. Black, F. Jelinek, J. Lafferty, D.M. Magerman, R. Mercer, and S. Roukos, “Towards History-Based Grammars: Using Richer Models for Probabilistic Parsing,” *Proc. Fifth DARPA Speech and Natural Language Workshop*, M. Marcus, ed., pp. 31-37, Feb. 1992.

[21] D.M. Magerman and M.P. Marcus, “Pearl: A Probabilistic Chart Parser,” *Proc. Second Int’l Workshop on Parsing Technologies*, pp. 193-199, 1991.

[22] D. Koller, D. McAllester, and A. Pfeffer, “Effective Bayesian Inference for Stochastic Programs,” *Proc. 14th Nat’l Conf. Artificial Intelligence*, pp. 740-747, Menlo Park, Calif., 1997.

David V. Pynadath received BS degrees in electrical engineering and computer science from the Massachusetts Institute of Technology in 1992. He received the MS degree in computer science from the University of Michigan in 1994. He is currently a doctoral student in computer science at the University of Michigan. His current research involves the use of probabilistic grammars and Bayesian networks for plan recognition.

Michael P. Wellman received a PhD in computer science from the Massachusetts Institute of Technology in 1988 for his work in qualitative probabilistic reasoning and decision-theoretic planning. From 1988 to 1992, Dr. Wellman conducted research in these areas at the USAF’s Wright Laboratory. He is currently an associate professor in the Department of Electrical Engineering and Computer Science at the University of Michigan. Current research also includes investigation of computational market mechanisms for distributed decision making. In 1994, he received a U.S. National Science Foundation National Young Investigator award.