8-25-1993

PARSEC: A Constraint-Based Parser for Spoken Language Processing

Mary P. Harper
Purdue University School of Electrical Engineering

Randall A. Helzerman
Purdue University School of Electrical Engineering

PARSEC: A CONSTRAINT-BASED PARSER FOR SPOKEN LANGUAGE PROCESSING

MARY P. HARPER
RANDALL A. HELZERMAN

TR-EE 93-28
AUGUST 1993

SCHOOL OF ELECTRICAL ENGINEERING
PURDUE UNIVERSITY
WEST LAFAYETTE, INDIANA 47907-1285

PARSEC: A Constraint-Based Parser for Spoken Language Processing

Mary P. Harper and Randall A. Helzerman
School of Electrical Engineering
Purdue University
West Lafayette, IN 47907

August 25, 1993

Abstract

PARSEC[1], a text-based and spoken language processing framework based on the Constraint Dependency Grammar (CDG) developed by Maruyama [26, 27], is discussed. The scope of CDG is expanded to allow for the analysis of sentences containing lexically ambiguous words, to allow feature analysis in constraints, and to efficiently process multiple sentence candidates that are likely to arise in spoken language processing. The benefits of the CDG parsing approach are summarized. Additionally, the development CDG grammars using PARSEC grammar writing tools and the implementation of the PARSEC parser for word graphs is discussed.

'Parallel ARchitecture Sentence Constrainer

1 Introduction

In this paper, we adapt the constraint dependency grammar (CDG) formalism introduced by Maruyama [26, 27, 28] to the problem of analyzing spoken language. Constraint dependency grammars are more expressive than context-free grammars (CFGs) and more tractable, but less expressive, than context-sensitive grammars, and they provide an extremely flexible framework for parsing natural language.

In Section 2 of this paper, the CDG parsing algorithm is described. In the remainder of the paper, the following three extensions to the CDG parsing algorithm are discussed:

1. Many words in the English language have lexical ambiguity (i.e., more than a single part of speech) which can cause correctness and efficiency problems for a parser [6]. A related problem is the processing of utterances with multiple word candidates over the same time interval. In Section 3.1, we extend the CDG parsing algorithm to handle lexical ambiguity and multiple word candidates in an integrated and efficient manner.

2. A word can also have ambiguity in the feature information associated with the word, like number, person, or case. For example, the noun fish can take a number/person value of third person singular or third person plural. Often feature information is useful for disambiguating parses for sentences or for eliminating impossible sentence hypotheses; hence, we have also added lexical feature analysis to the CDG parser, as described in Section 3.3.

3. Maruyama's CDG parsing algorithm was designed to process single sentences. If a natural language processor is used in conjunction with a speech recognizer, the input to the natural language component would typically be a set of sentence hypotheses. However, processing all of the sentence hypotheses individually for an utterance provided by a speech recognizer is very inefficient since many hypotheses are similar. Furthermore, a list of sentence hypotheses is not the most compact representation to provide a natural language parser. A better representation is a word graph or lattice of word candidates, which reduces the redundancy and compactly represents the set of sentence hypotheses. In section 4, we extend CDG to operate on a word lattice rather than single sentences. We also describe the implementation of our word graph constraint-based parser, PARSEC (Parallel ARchitecture Sentence Constrainer), and the development of constraint-based grammars using PARSEC grammar development tools.

2 A Description of Maruyama's CDG Parsing

We begin with the definition of Constraint Dependency Grammar (CDG), discuss Maruyama's CDG parsing algorithm, and then relate the algorithm to other approaches.

2.1 Elements of a CDG Grammar

Maruyama defines a CDG grammar as a 4-tuple, (C, R, L, C), where:

Σ = a finite set of preterminal symbols, or lexical categories.
R = a finite set of uniquely named roles (or role-ids) = $\{r_1,\ldots,r_p\}$.
L = a finite set of labels = $\{l_1,\ldots,l_q\}$.
C = a constraint set that an assignment A must satisfy.

A sentence s = $w_1w_2w_3\ldots w_n$ is a string of finite length n and is an element of C*. All of the roles in R are associated with every w; of s yielding $n*p$ roles for the entire sentence. The sentence s is said to be generated by the grammar G if there exists an assignment A which maps role values to each of the $n*p$ roles for s such that the constraint set C is satisfied. A role value is an element of the set L $\times$ {1,2,...,*n*,nil}, in other words, it is a tuple consisting of a label from L and a modifiee, where a modifiee can be the index of a word in the sentence or nil. Role values will be denoted in the examples as *label-modifiee*. $L(G)$ is the language generated by grammar G iff $L(G)$ is the set of all sentences generated by G. Note that the null string ϵ has no roles and is always generated by any grammar according to definition.

A ***constmint set*** is a logical formula in the form: $\forall\ x_1\ x_2\ \ldots\ x_p$: ***role*** (and $P_1\ P_2\ \ldots$ P,), where the x_is range over all of the roles in *s*. Below is the definition of possible components of a subformula P_i[2]:

- **Variables:** $x_1, x_2, \ldots x_p$.
- **Constants:** elements and subsets of $\Sigma \cup$ L $\cup$ R $\cup$ {nil, 1, 2, ..., n), where n corresponds to the number of words in a sentence.
- **Access Functions:**

  (pos x) returns the position of the word for variable x.
  (rid x) returns the role-id for variable x.
  (lab x) returns the label for variable x.
  (mod x) returns the position of the modifiee for variable x.
  (cat i) returns the category (i.e., the element in Σ) for the word [3] in position i.

- **Predicate symbols:**

  (eq x y) returns true if x = y, false otherwise.
  (gt x y) returns true if x > y and x, y $\in$ Integers, false otherwise[4].
  (lt x y) returns true if x < y and x, y $\in$ Integers, false otherwise.
  (elt x y) returns true if x $\in$ y, false otherwise.

[2] Maruyama uses an infix notation; whereas, we use a prefix notation throughout this paper.
[3] Maruyama uses the access function word rather than cat, though the function accesses the category of the word.
[4] For example, (gt 1 nil) is false, because nil is not an integer.

- Logical Connectives:

  (and **p q)** returns true if p and q are true, false otherwise.
  (or p q) returns true if p or q is true, false otherwise.
  (not **p)** returns true if p is false, false otherwise.

Each P_i in C must be of the form (if Antecedent Consequent), where Antecedent and Consequent are predicates or predicates joined by the logical connectives. A CDG grammar has two associated parameters, degree and arity. The degree of a grammar G is the size of R. The arity of the grammar corresponds to the maximum number of variables in the subformulas of C. To simplify the examples in this paper, we use grammars with a degree of one, that is, with a single role governor. The governor role indicates the function a word fills in a sentence when it is governed by its head word. In our implemented grammars (described in section 4.3), we also use several needs roles (e.g, needl, need2) to make certain that a head word has all of the constituents it needs to be complete (e.g., a singular count noun needs a determiner to be a complete noun phrase). Maruyama has proven that a grammar requires a degree and arity of two to be as expressive as a CFG.

To illustrate the use of CDG grammars, consider a very simple example grammar, $G_1 = (\Sigma_1, R_1, L_1, C_1)$ in Figure 1, which has a degree of one and an arity of two[5]. A subformula P_i is called a unary constraint if it contains one variable and a binary constraint if it contains two. For example, U-1, U-2, and U-3 are unary constraints because they contain a single variable, and B-1 is a binary constraint because it contains two variables.

For G_1 to generate the sentence The program runs, there must be an assignment of a role value to the governor role of each word, and that assignment must simultaneously satisfy each of the subformulas in C_1. Note that each word is assumed to have a single lexical category, which is determined by dictionary lookup. Figure 2 depicts an assignment for the sentence which satisfies C_1. This assignment can be interpreted as the parse graph shown in Figure 9.

2.2 CDG Parsing

To determine whether a sentence is generated by a grammar, a CDG parser must be able to assign at least one role value which satisfies the grammar constraints to each of the $n * p$ roles, where n is sentence length, and p is the number of role-ids. Because the role values for the role are selected from the finite set $L_1 \times \{1,2,\ldots,n,\text{nil}\}$, CDG parsing can be viewed as a constraint satisfaction problem over a finite domain. Hence, constraint propagation [21, 30, 51] can be used to develop the parse of a sentence. A CDG parser generates **all** parses for a sentence in a compact representation

[5] **The constraints in this grammar were chosen for simplicity, not to exemplify constraints for a wide coverage grammar.**

Σ_1 = {det , noun, verb)
R_1 = {governor)
L_1 = {DET, SUBJ, ROOT)
C_1 = V x y: role (and

```
;; [U-1] A det receives the label DET and rodifies a word to its right.
(if (eq (cat (pos x)) det)
    (and (eq (lab x) DET)
         (lt (pos x) (rod x))))
;; [U-2] A noun receives the label SUBJ and modifies a vord to its right.
(if (eq (cat (pos x)) noun)
    (and (eq (lab x) SUBJ)
         (lt (pos x) (rod x))))
;; [U-3] A verb receives the label ROOT and rodifies no vord.
(if (eq (cat (pos x)) verb)
    (and (eq (lab x) ROOT)
         (eq (rod x) nil)))
;; [B-1] A DET is governed by a SUBJ.
(if (and (eq (lab x) DET)
         (eq (mod x) (pos y)))
    (eq (lab y) SUBJ))
)
```

Figure 1: $G_1 = \langle \Sigma_1, R_1, L_1, C_1 \rangle$.

pos	word	cat	governor role's value
1	the	det	DET-2
2	program	noun	SUBJ-3
3	runs	verb	ROOT-nil

Figure 2: An assignment for The *program* runs.

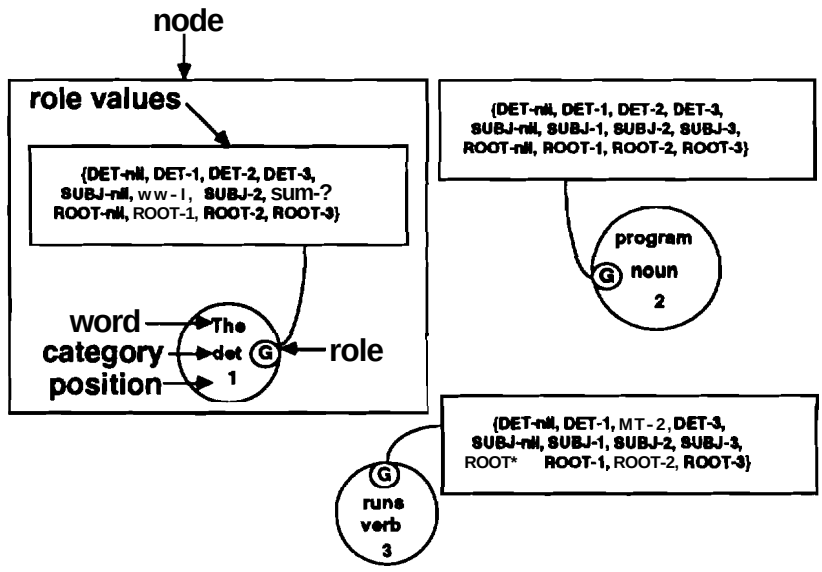


Figure 3: Initialization of roles for the sentence The progmm runs.

because enumeration of the individual parses for a highly ambiguous sentence is intractable. The steps required for parsing the sentence The program runs are provided to illustrate both the process of parsing with constraint propagation and the running time of the algorithm.

To develop a syntactic analysis for a sentence using CDG, a constraint network (CN) of words is created. Each of the n words in a sentence is represented as a node in a CN. Figure 3 illustrates the initial configuration of nodes in the CN for The program runs example. Notice that associated with each node is its word, category, sentence position, and roles (only one for this example). Each of the roles is initialized to the set of all possible role values (i.e., the domain). Given G_1, the domain for the example is $L_1 \times \{1,2,3,\text{nil}\}$ = {DET-nil, DET-1, DET-2, DET-3, SUBJ-nil, SUBJ-1, SUBJ-2, SUBJ-3, ROOT-nil, ROOT-1, ROOT-2, ROOT-3). Since there are $q * (n + 1) = O(n)$ possible role values for each of the $p * n$ roles for a sentence (where p, the number of roles per word, and q, the number of different labels, are grammatical constants, and n is the number of words in the sentence), there are $O(p * n * q * (n + 1)) = O(n^2)$ role values which must be initially generated for the CN, requiring $O(n^2)$ time. Appendix A contains pseudocode for this step and all of the remaining steps in the CDG parsing algorithm.

To parse the sentence using G_1, the unary and binary constraints in C_1 are applied to the CN to eliminate the role values from the roles of each word which are incompatible with C_1. For a sentence to be grammatical, each role in each word node must contain at least one role value after constraint propagation.

The unary constraints are applied to each of the roles in the sentence to eliminate the role values incompatible with each word's role in isolation. To apply the first unary constraint (i.e.,

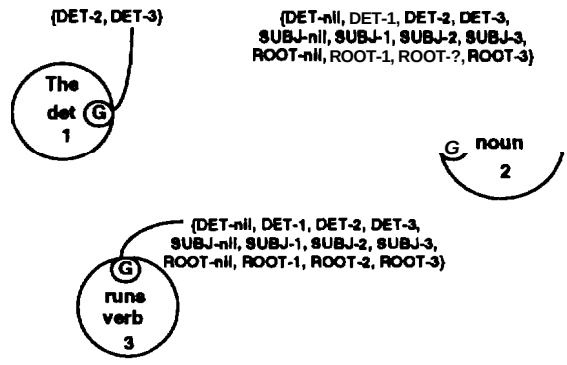


Figure 4: The CN after the propagation of U-1 for the sentence The *program* runs.

U-1, shown below) to the network in Figure 3, each role value for every role is examined to ensure that it obeys the constraint.

```
;; [U-1] A det receives the label DET and modifies a word to its right.
(if  (eq (cat  (pos x)) det)
     (and  (eq (lab  x) DET)
           (lt (pos x) (rod  x))))
```

If a role value causes the antecedent of the constraint to evaluate to TRUE and the consequent to evaluate to FALSE, then that role value is eliminated. Figure 4 shows the remaining role values after U-1 has been applied to the CN in Figure 3.

Maruyama requires that each subformula in a constraint set be evaluated in constant time. Because of this restriction, each constraint can only contain access functions and predicates that operate in constant time (e.g., access functions and predicates like those defined in Section 2.1). So when the unary constraint U-1 is applied to $O(n^2)$ role values, it requires $O(n^2)$ time.

To further eliminate role values which are incompatible with the categories of the words in the example, the remaining unary constraints (i.e., U-2 and U-3) are applied to the CN in Figure 4, producing the network in Figure 5. Given that the number of unary constraints in a grammar is a grammatical constant denoted as k_u, the time required to apply all of the unary constraints in a grammar is $O(k_u * n^2)$.

The binary constraints determine which pairs of role values can legally coexist. To keep track of pairs of role values, arcs connect each role to **all** other roles in the network, and each arc has an associated arc *matrix*, whose row and column indices are the role values associated with the two roles. The elements of an arc matrix can either be a **1** (indicating that the two role values which index the element are compatible) or a **0** (indicating that the role values cannot simultaneously exist). Initially, **all** entries in each matrix are set to 1, indicating that the two role values are initially compatible. Since there are $\binom{n*p}{2} = O(n^2)$ arcs required in the CN, and each arc contains a matrix with $O((q * (n + 1))^2) = O(n^2)$ elements, the time to construct the arcs and initialize

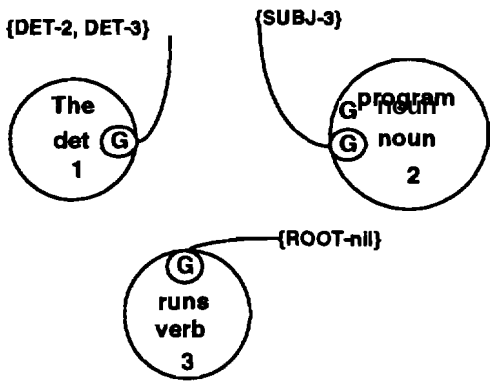


Figure 5: The CN after the propagation of all the unary constraints.

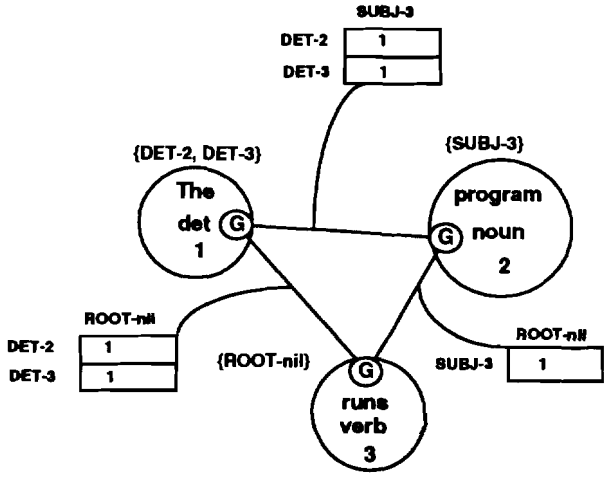


Figure 6: The CN after unary constraint propagation and before binary constraint propagation.

the matrices is $O(n^4)$. Figure 6 shows the matrices associated with the arcs before any binary constraints are propagated. Unary constraints are usually propagated before preparing the CN for binary constraints because they eliminate impossible role values from each role, and hence reduce the dimensions of the arc matrices.

Binary constraints are applied to the pairs of role values indexing each of the arc matrix entries. When a binary constraint is violated by a pair of role values, the entry in the matrix indexed by those role values is set to zero. The binary constraint, B-1, ensures that a DET is governed by a SUBJ:

```
;; [B-1] A DET is governed by a SUBJ.
(if (and (eq (lab x) DET)
         (eq (rod x) (pos y))
    (eq (lab y) SUBJ))
```

After the application of this constraint to the network in Figure 6, the element indexed by the role values x=DET-3 and y=ROOT-nil for the matrix on the arc connecting the governor roles for the

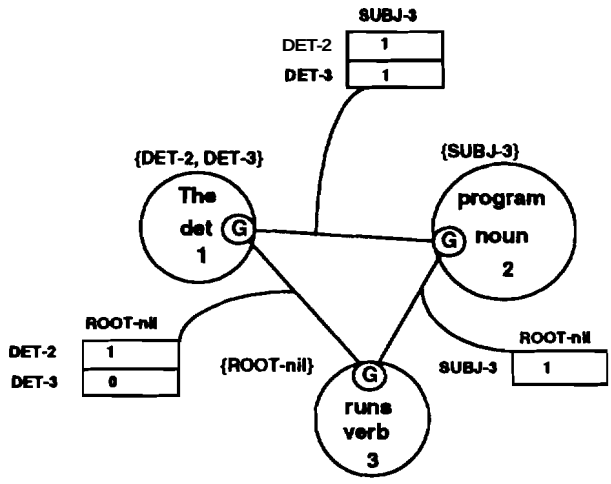


Figure 7: The CN after B-1 is propagated.

and runs is set to zero, as shown in Figure 7. This is because the must be governed by a word with the label SUBJ, not ROOT. Since the constraint must be applied to $O(n^4)$ pairs of role values, the time to apply the constraint is $O(n^4)$. Given that the number of binary constraints in a grammar is a grammatical constant denoted as k_b, the time required to apply all of the binary constraints in a grammar is $O(k_b * n^4)$.

Following the propagation of binary constraints, the roles of the CN could still contain role values which are incompatible with the parse for the sentence. To determine whether a role value is still supported for a role, each of the matrices on the arcs incident to the role must be checked to ensure that the row (or column) indexed by the role value contains at least a single **1**. If any arc matrix contains a row (or column) of **0**s for the role value, then that role value cannot coexist with any of the role values for the second role and so is removed from the list of legal role values for the first role. Additionally, the rows (or columns) associated with the eliminated role value can be removed from the arc matrices attached to the role. The process of removing any rows or columns containing all zeros from arc matrices and eliminating the associated role values from their roles is called filtering. Following binary constraint propagation any of the $O(n^2)$ role values may require immediate filtering. However, filtering must also be applied iteratively since the elimination of a role value from one arc could lead to the elimination of a role value from another arc. The most efficient filtering algorithm requires $O(ea^2)$, where e is the number of arcs, and a is the size of the domain [29]. In the case of CDG parsing, $e = \binom{n*p}{2}$, and the domain size is $n * q$, so the running time of the filtering step is $O(n^4)$ [26, 27].

Consider how filtering is applied to the CN in Figure 7[6]. The matrix associated with the arc

[6] **Our implementation of the algorithm determines whether a role value is supported by its arcs by ORing all the elements it indexes in an arc matrix and ANDing the results from all of those arc matrices. Hence, if any arc matrix fails to support a role value in a role, the result of the AND would be 0, and the role value would be eliminated.**

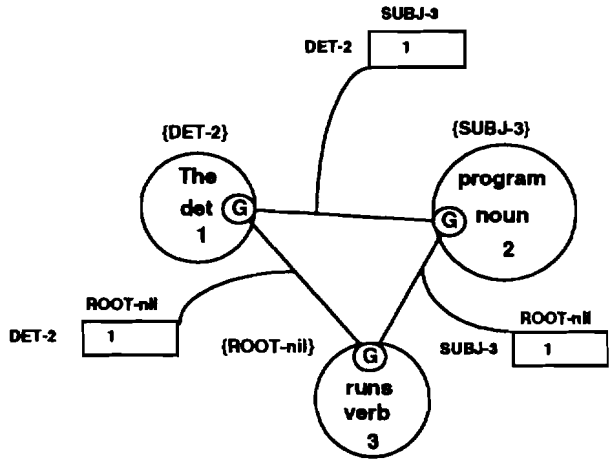


Figure 8: The CN after filtering.

connecting the and runs contains a row with a single element which is a zero. Because DET-3 cannot coexist with the only possible role value for the governor role of runs, it cannot be a legal member of the governor role of the and is therefore eliminated as a role value in that role. It is also removed from the row it indexes in the matrix associated with the other arc emanating from that role. Figure 8 illustrates the resulting CN after filtering the CN from Figure 7. Notice that following filtering, there is precisely one role value per role for the example.

After all the constraints are propagated across the CN and filtering is performed, the CN provides a compact representation for all possible parses. Syntactic ambiguity is easy to spot in the CN since some of the roles in an ambiguous sentence contain more than a single role value. If multiple parses exist, we can propagate additional constraints to further refine the analysis of the ambiguous sentence, or we could just enumerate the parses contained in the CN by using backtracking search. For highly ambiguous grammars, the process of enumerating all possible parses is intractable, making incremental disambiguation a more attractive option. The parse trees in a CN are precedence graphs, which we call parse graphs, and they consist of a compatible set of role values (given the arc matrices) for each of the roles in the CN. The modifiees of the role values, which point to the words they modify, form the edges of the parse graph. Our example sentence has an unambiguous parse graph given G_1, shown in Figure 9.

Below we list the steps in the CDG parsing algorithm and their associated running times:

1. Constraint Network construction prior to unary constraint propagation: $O(n^2)$
2. Unary constraint propagation: $O(k_u * n^2)$
3. Constraint Network construction prior to binary constraint propagation: $O(n^4)$
4. Binary constraint propagation: $O(k_b * n^4)$

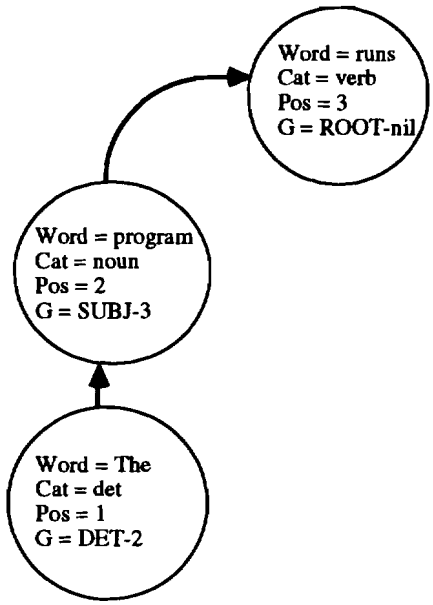


Figure 9: The parse graph for the CN in Figure 8.

5. Filtering: $O(n^4)$

Notice that the time required to propagate binary constraints is the slowest part of the algorithm.

2.3 CDG Parsing Compared with Other Approaches

In this section, the CDG parser is first compared with other constraint-satisfaction problems, and then it is compared with more traditional parsing approaches.

2.3.1 CDG Parsing Compared with Constraint-Satisfaction Problems

Constraint-satisfaction problems (CSP) have a rich history in Artificial Intelligence [3, 4, 5, 9, 10, 25, 26, 51] (see [22] for a survey of CSP). If CSP is restricted to a finite, discrete domain, and only constraints over single variables or pairs of variables are allowed, then there is a mapping between CDG parsing and CSP.

In a typical finite-domain CSP problem, the variables (roles) are depicted as circles, and each variable is assigned a finite set of possible values. Constraints imposed on the variables are depicted along with arcs drawn between the circles (variables) affected by the constraint. Because binary constraints involve two variables, a constraint arc is drawn between the two corresponding circles. However, because unary constraints involve only single variables, the constraint arc loops from a circle to itself. Consider the CSP problem on the left-hand side of Figure 10 (this example is based on a CSP network discussed in [1]), and notice how it maps to the CDG constraint network to the right. In CSP, binary constraints are depicted along with their arcs; whereas, in a CN, the constraints are

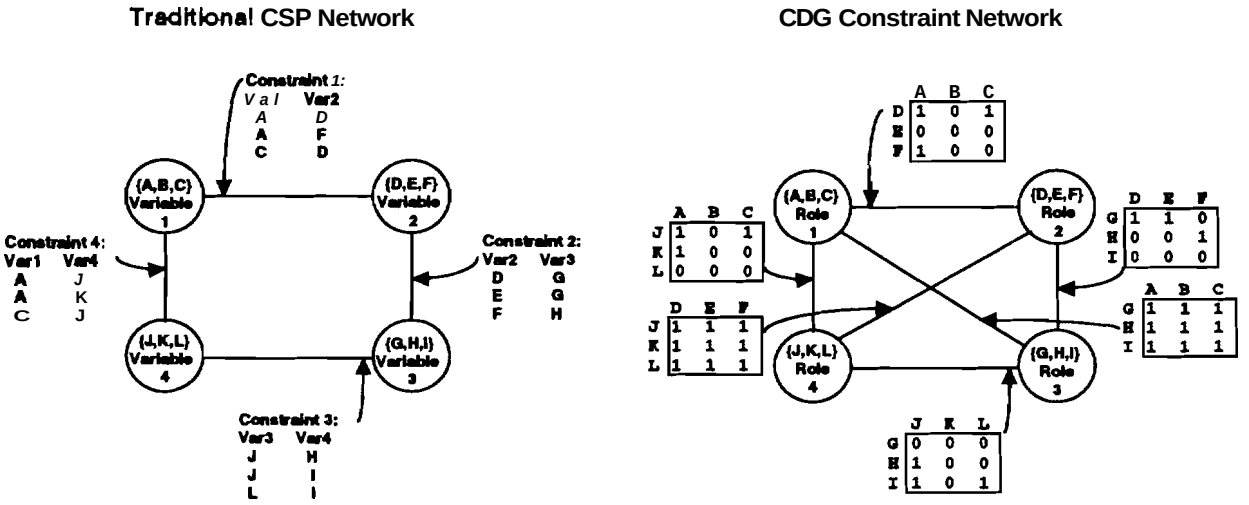


Figure 10: A CSP network compared to an equivalent CDG constraint; network

not depicted. Instead, constraints are applied to the CN, which maintains information about which role values are consistent with the set of applied constraints. The algorithm for CDG filtering is similar to the most efficient algorithm for maintaining arc consistency for CSP problems designed by Mohr and Henderson [29]. In the arc consistency algorithm, arc matrices are used to detect and eliminate values associated with a circle (variable) that are not supported by all of the arcs attached to the circle.

2.3.2 CDG Parsing Compared with Other Parsing Methods

Researchers have developed and used a variety of parsing paradigms which are based on context-free grammars (CFGs) or are equivalent in expressivity to CFGs. A complete survey of CFG parsers is beyond the scope of this paper; however, a comparison of the CDG parser with several representative CFG parsers will illustrate their similarities and differences.

Most of the CFG parsers that are used in computer language applications (e.g., LL and LR parsers) compile grammar rules into a table which is then used by a parser driver to deterministically parse programs. Though these parsers have been used by the natural language processing community, the ambiguity that is common in natural language has caused researchers to seek alternative methods. A recursive transition network parser [55], which encodes a grammar in a network and then searches the network in a top-down fashion, is able to produce all possible parses for an ambiguous sentence. However, if the parser does not cache subresults during that search it can have an exponential running time. Definite Clause Grammar parsers [34] use unification and search to determine the parses for a sentence. Chart-based parsers [19] use a chart to build up constituents in a bottom-up fashion, and for some ambiguous sentences, can create charts of constituents that are

exponential in size. Tomita [46] has developed an LR parsing method which creates a parse forest of possible parses using an LR table with shift-reduce and reduce-reduce ambiguities. Tomita's parse forest also grows exponentially for some highly ambiguous grammars. Early [7] developed a parser which provably operates in $O(G^Z n^3)$ time, but the algorithm builds a forest of parses that, in some cases, includes impossible parses that must be pruned by checking the consistency of leaf nodes in each tree. There are other approaches that improve the average case efficiency of the Early parser, but they do not improve the efficiency under worst-case scenarios [38].

Below we enumerate the differences between CFG parsers and the CDG parser:

1. CFG parsers use production rules (or equivalently, recursive transition networks) in a grammar to determine whether or not a string of terminals is in the language. The CDG parser, on the other hand, uses sets of constraints. Any type of constraint that can be formulated as an if-then rule containing one or two role value variables can be used to constrain a CN.

2. CFG parsers that generate all parses for ambiguous grammars use a variety of methods to search through the rules for all possible parses. To save time during parsing, grammar rules can be preprocessed and sub-results cached and reused. In contrast, the CDG parser assumes anything is possible until constraint propagation is used to eliminate impossible analyses.

3. The best serial running time for a CFG parser operating with an ambiguous grammar is $O(G^2 * n^3 3$ [7], where n is the number of words in a sentence and G is the size of the CFG grammar. On the other hand, the serial running time for the CDG parser of single sentences is $O(k * n^4)$, where k is the number of constraints in grammar. In practice, we have found that k is comparable in size to G for grammars with the same coverage.

4. CFG parsers often build trees for each of the possible parses for a sentence. However, enumerating the trees for highly ambiguous sentences can require exponential time. Hence, some CFG parsers construct a parse forest (or similar structure) to circumvent this problem. The CN constructed by the CDG parser is a forest of parse graphs which is pruned during parsing. A CN differs from a CFG parse forest in that there are no non-terminals in the graph, only links between terminals, which are assigned sets of labels. A packed parse forest generated by a CFG parsing algorithm can be mapped to a syntactic graph [44], which like a CN, compactly encodes all parses of ambiguous sentences by using modifier links between a head word and its modifiers. Exclusion matrices are associated with the syntactic graph, which, like CDG arc matrices, prevent impossible parses from being selected from the graph. However, unlike a CN, a syntactic graph is generated by a grammar which is context-free.

5. The smallest known size for a CFG parse forest is $O(G^Z * n^3)$ [7], regardless of how many

parses there are for an ambiguous sentence. In contrast, a CN has the size[7] of $O(n^4)$, and it is the only data structure used during parsing; there is no stack or agenda to maintain.

6. When a CFG parser generates a set of ambiguous parses for a sentence, it cannot invoke additional production rules to further prune the analyses. In contrast, in CDG parsing, the presence of ambiguity can trigger the propagation of additional constraints to further refine the parse for a sentence. A core set of constraints that hold universally can be propagated first, and then if ambiguity remains, additional, possibly context dependent, constraints can be used. This type of flexibility is easy to achieve, since constraints are not precompiled into a table (like in LR parsing) or into a network (like an ATN).

7. CFG parsing has been parallelized by several researchers. For example, Kosaraju's method [20] using cellular automata can parse CFGs in $O(n)$ time using $O(n^2)$ processors. However, achieving CFG parsing times of less than $O(n)$ has required more powerful and less implementable models of parallel computation, as well as significantly more processors. Ruzzo's method [37] has a running time of $O(\log^2(n))$ using a CREW P-RAM model (Concurrent Read, Exclusive Write, Parallel Random Access Machine), but requires $O(n^6)$ processors to achieve that time bound. In contrast, we have devised a parallelization for the single sentence CDG parser [13, 15] which uses $O(n^4)$ processors to parse in $O(k)$ time for a CRCW P-RAM model (Concurrent Read, Concurrent Write, Parallel Random Access Machine), where n is the number of words in the sentence and k, the number of constraints, is a grammatical constant. Furthermore, this algorithm has been simulated on the MasPar MP-1[8], using the special features of the machine and $O(n^4)$ processors to obtain an $O(k + log(n))$ running time.

8. To parse a free-order language like Latin, CFGs require that additional rules containing the permutations of the right-hand side of a production be explicitly included in the grammar [31]. Unordered CFGs do not have this combinatorial explosion of rules, though the universal recognition problem for this class of grammars is NP-complete for an n-character alphabet. A free-order language can easily be handled by a CDG parser because order between constituents is not a requirement of the grammatical formalism. Furthermore, CDG is capable of efficiently analyzing free-order languages because it is does not have to test for all possible orders of words.

9. The set of languages accepted by a CDG grammar is a superset of the set of languages which

[7] In natural language processing, n is typically much smaller than G. English sentences usually contain fewer than 30 words; whereas, hundreds or even thousands of production rules are not uncommon for broad coverage English grammars.

[8] The MasPar MP-1 is a massively parallel SIMD computer, which supports up to 16K 4-bit processing elements, each with 16KB of local memory.

can be accepted by CFGs. In fact, Maruyama [26, 27] is able to construct CDG grammars with two roles (degree = 2) and two variable constraints (arity = 2) which accept the same language as an arbitrary CFG converted to Griebach Normal form. We have also devised an algorithm to map a set of CFG production rules into a CDG grammar. This algorithm does not assume that the rules are in normal form, and the number of constraints created is $O(G)$. In addition, CDG can accept languages that CFGs cannot, for example, $a^n b^n c^n$ and ww, (where w is some string of terminal symbols). To illustrate the ease of writing such grammars, we have created a grammar which accepts the language $a^n b^n c^n$, $n \geq 0$, shown in figure 11. This grammar determines whether a string is acceptable by ensuring that there is a one-to-one correspondence between each b and a, each c and b, and each a and c, and that all a's occur before all *b*'s and all *b*'s occur before all *c*'s. An assignment for the string aaabbbccc given G_2 is shown in Figure 12.

Constraint-based parsing has received considerable interest in computational linguistics in recent years. For example, Covington [2] outlines a constraint-based parser that uses dependency rules to set up modifiee links between terminals, as in CDG. Covington's parser differs from CDG in that it uses search and unification to provide dependency graphs for sentences, while CDG uses constraint propagation. However, because both approaches use rules limiting dependency links between terminals, the dependency rules of Covington's parser should easily map into CDG constraints. Additionally, both share a capability for handling free order languages. Shieber [45] develops a constraint-based approach to parsing which also uses unification. Shieber's rules consist of two parts, a CFG phrase-structure portion and a feature analysis portion. The strength of Shieber's approach is in his well-defined semantics of feature constraints. CDG differs from Shieber's approach in several ways. First, in CDG, all rules are specified as constraints; there is no separation between phrase-structure rules and other types of constraints. Second, CDG is not limited to the use of CFG phrase structure rules.

There has also been considerable interest in the development of parsers for grammars that are more expressive than the class of context-free grammars, but less expressive than context-sensitive grammars [18, 49, 48]. The running time of the CDG parser compares quite favorably to the running times of parsers for languages which are beyond context-free. For example, the parser for tree adjoining grammars has a running time[9] of $O(n^6)$. A direct comparison between CDG and these more expressive grammars is beyond the scope of this paper.

In summary, CDG is more expressive and flexible than CFGs, making it an attractive alternative to traditional parsers. It is able to utilize a variety of different knowledge sources in a uniform framework to incrementally disambiguate a sentence's parse. The algorithm also has the advantage

[9] This algorithm has also been parallelized, and operates in linear time with $O(n^5)$ processors [32].

```
Σ2 = {a, b, c}.
R2 = {governor}
L2 = {A,B,C}
C2 = V x y: role (and
                    ;;; [U-1] An a receives the label A and modifies an item to its right.
                    (if (eq (root-word x) a)
                        (and (eq (lab x) A)
                             (gt (rod x) (pos x))))
                    ;;; [U-2] A b receives the label B and modifies an item to its left.
                    (if (eq (root-word x) b)
                        (and (eq (lab x) B)
                             (lt (rod x) (pos x))))
                    ;;; [U-3] A c receives the label C and modifies an item to its left.
                    (if (eq (root_word x) c)
                        (and (eq (lab x) C)
                    ;;; [B-1] Every A precedes every B.
                    (if (and (eq (lab x) A)
                             (eq (lab y) B))
                        (lt (pos x) (pos y)))
                    ;;; [B-2] Every B precedes every C.
                    (if (and (eq (lab x) B)
                             (eq (lab y) C))
                        (lt (pos x) (pos y)))
                    ;;; [B-3] If an A occurs after another A, then it rust
                    ;;; modify something after that A's modifiee.
                    (if (and (eq (lab x) A)
                             (eq (lab y) A)
                             (gt (pos x) (pos y)))
                         (gt (rod x) (rod y)))
                    ;;; [B-4] An A rust modify a C.
                    (if (and (eq (lab x) A)
                             (eq (rod x) (pos y))
                             (eq (rid y) governor))
                        (eq (lab y) C))
                    ;;; [B-5] If a B occurs after another B, then it rust
                    ;;; modify something after that B's rodifiee.
                    (if (and (eq (lab x) B)
                             (eq (lab y) B)
                             (gt (pos x) (pos y)))
                        (gt (rod x) (rod y)))
                    ;;; [B-6] A B rust modify an A.
                    (if (and (eq (lab x) B)
                             (eq (rod x) (pos y))
                             (eq (rid y) governor))
                        (eq (lab y) A))
                    ;;; [B-5] If a C occurs after another C, then it must
                    ;;; modify something after that C's modifiee.
                    (if (and (eq (lab x) C)
                             (eq (lab y) C)
                             (gt (pos x) (pos y)))
                        (gt (rod x) (mod y)))
                    ;;; [B-6] A C rust modify a B.
                    (if (and (eq (lab x) C)
                             (eq (rod x) (pos y))
                             (eq (rid y) governor))
                        (eq (lab y) B))
                    )
```

Figure 11: $G_2 = \langle \Sigma_2, R_2, L_2, C_2 \rangle$ accepts the language $a^n b^n c^n$, n ≥ 0.

pos	word	governor role's value
1	a	A-7
2	a	A-8
3	a	A-9
4	b	B-1
5	b	B-2
6	b	B-3
7	c	C-4
8	c	C-5
9	c	C-6

Figure 12: An assignment for *aaabbbccc*.

that is is efficiently parallelizeable. Because CDG parsing differs from traditional parsers in its use of constraint propagation, further study of the relationship between CDG and other parsing approaches could lead to some useful insights about parsing algorithms in general.

3 Enhancements of CDG Text-Based Parsing

This section describes a number of enhancements we have made to the CDG parsing algorithm for single sentences to increase its usefulness for both text-based and spoken natural language processing. First, the algorithm is modified to parse sentences with lexically ambiguous words. Next, a table is introduced to lexically restrict the possible labels for a word during CN construction. Finally, the parsing algorithm is modified to allow constraints to test for features such as number or person.

3.1 Lexical Ambiguity

Many words in the English language have more than a single part of speech. For example, the last two words in The program ***runs*** can be either nouns or verbs. Maruyama's algorithm requires that a word have a single part of speech, which is determined by dictionary lookup prior to the application of the parsing algorithm. Since parsing can be used to lexically disambiguate a sentence, ideally, a parsing algorithm should not require that a part of speech be known prior to parsing. In addition, lexical ambiguity, if not handled in a reasonable manner, can cause correctness and/or efficiency problems for a parser [6]. We examine four strategies for processing lexically ambiguous sentences within the CDG framework.

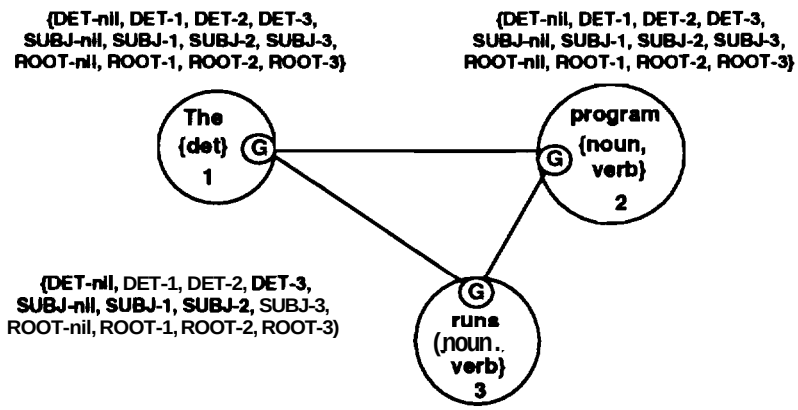


Figure **13:** The initial CN for the second lexical ambiguity strategy.

The first strategy is to create and parse a set of CNs which cover all of the possible combinations of parts of speech for each word. Because of the combinatorial explosion of CNs, this strategy is intractable.

A second strategy is to record in each word node **all** of the applicable parts of speech. For example, in the CN for The program ***runs*** record that the last two words are both nouns and verbs, as shown in Figure **13.** The parsing algorithm can now no longer generate a parse for the sentence because, given C_1, the noun must have the label SUBJ, eliminating role values with label DET and ROOT, and at the same time, the verb must have the label ROOT, eliminating role values with label DET and SUBJ. In other words, after applying the constraints, there are no role values remaining.

A third strategy is to create a word node for each part of speech for a word, as shown in Figure **14.** Using this approach, the maximum size of a constraint network given that each word can have w parts of speech is $\binom{wpn}{2}(qn)^2$, where n is the number of words, p is the number of roles, and q is the number of labels (simplifying to $\frac{wpn(wpn-1)}{2}q^2n^2 = \frac{w^2p^2q^2n^4-wpq^2n^3}{2}$). However, this solution requires a change in the CDG parsing algorithm, which assumes that a CN is an AND/OR tree such that the values assigned to the roles account for the only OR nodes in the tree, as shown in Figure **15.** Hence, for a sentence to have a parse, every role in the CN must have a least one role value after filtering. On the other hand, for the proposed lexically ambiguous network to have a solution, only one of the nodes created to represent the multiple parts of speech for a word needs to be supported. To use this proposed network, we could change the semantics of a constraint network to include an OR node above the level of the role, but this solution requires modification of the CDG network construction and filtering algorithms. Furthermore, there is a fourth solution which requires requires $O(n^3)$ less space than the current strategy, minor modification of the CN construction routines, and no modification of the filtering algorithm.

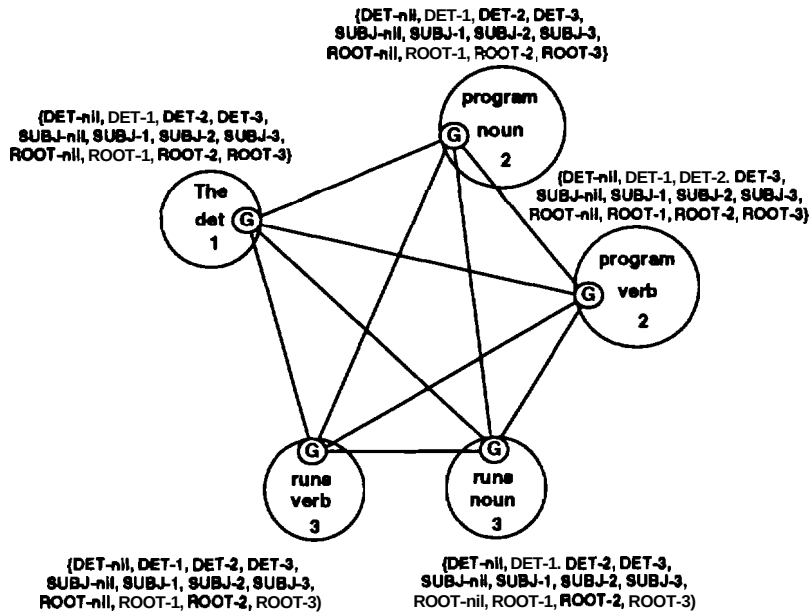


Figure 14: The initial CN for the third lexical ambiguity strategy.

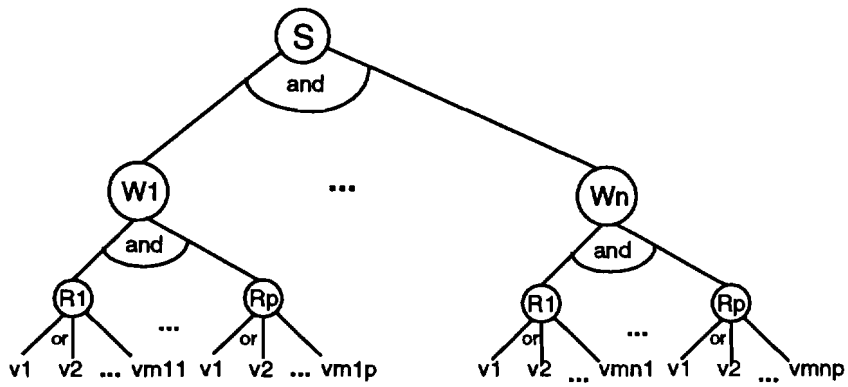


Figure 15: The AND/OR tree for the CDG parsing algorithm.

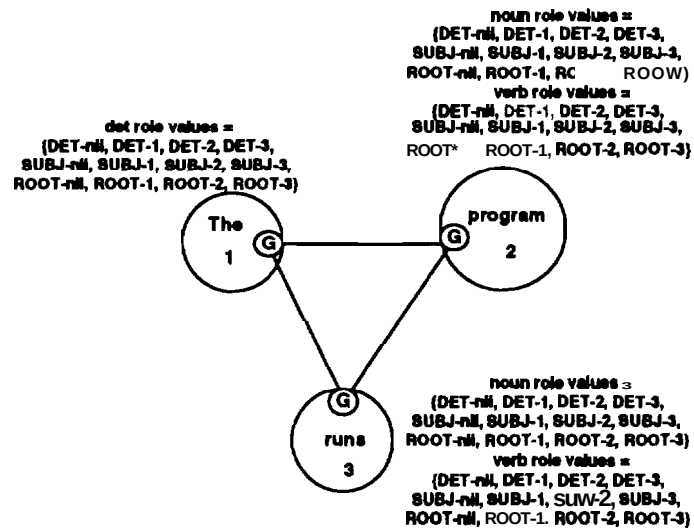


Figure 16: The initial CN for the fourth lexical ambiguity strategy.

The fourth strategy is to allow role values within the same node to have their own parts of speech as shown in Figure 16[10]. When the CN is constructed, the parts of speech for each word are determined by looking the word up in the dictionary. If a word is lexically ambiguous then for each part of speech, a set of role values in the domain is created and assigned that part of speech. This solution makes use of the fact that, in CDG parsing, no more than one role value for a role can occur in the same parse graph. This strategy places the disjunction associated with lexical ambiguity at the level of the role, and hence requires no modification of the filtering algorithm. As shown in F'igure 17, the categories are represented as the c_is, which are ORed below the level of the role; hence, no modification of the CDG filtering algorithm is required.

Because each role value has its own part of speech, the constraints in G_1 in Figure 1 are rewritten so that the access function **cat** operates on a role value rather than on a word node addressed by its position. For example, U-1 is rewritten as follows:

```
;; [U-1] A det receives the label DET and
;; modifies a word to its right.
(if (eq (cat x) det)
    (and (eq (lab x) DET)
         (lt (pos x) (rod x))))
```

This modification of CDG parsing requires less space than the previous strategy. Note that the maximum size of this modified constraint network is $\binom{pn}{2}(wqn)^2$, which simplifies to $\frac{pn(pn-1)}{2}w^2q^2n^2 = \frac{w^2p^2q^2n^4 - w^2pq^2n^3}{2}$. Hence, the space requirement for this modified CN is $\binom{w}{2}pq^2n^3$ smaller than for the previous strategy.

[10] Rather than show each role value with its corresponding part of speech in the figures, we show the set of all of the role values with a particular part of speech to save space.

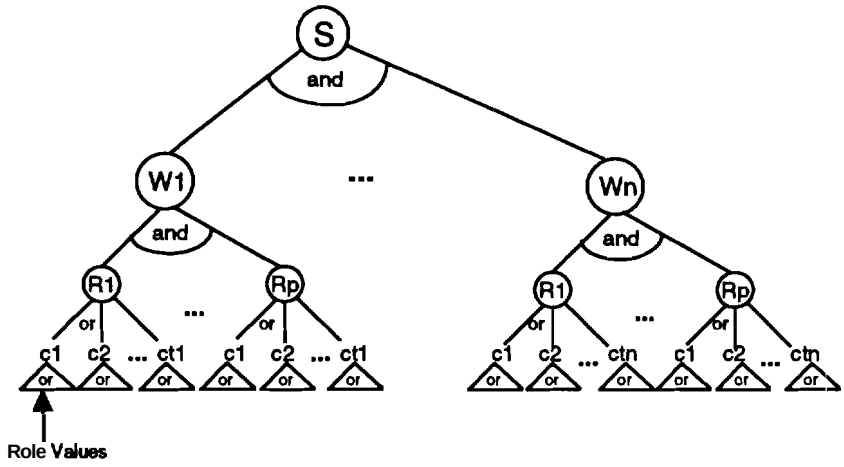


Figure 17: The AND/OR tree for the CDG parsing algorithm with lexical ambiguity.

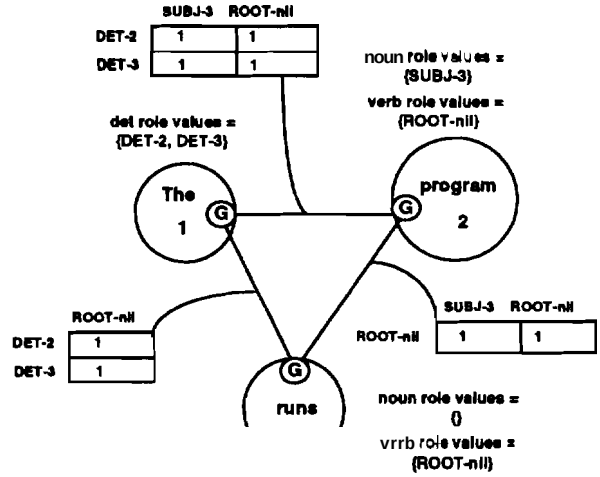


Figure 18: The CN after unary constraint propagation.

By using the modified CN and constraints, the CDG parsing algorithm, operating on the lexically ambiguous CN for The ***program runs,*** produces the same parse graph as the CN without lexical ambiguity for the same sentence from section 2.2. Following the propagation of unary constraints, the labels in the CN in Figure 16 are reduced in number, as shown in Figure 18. Notice that all of the role values for the noun ***runs*** have been eliminated, therefore, the word cannot be used as a noun in the sentence given G_1. Figure 19 depicts the CN after the propagation of binary constraints, and Figure 20 shows the CN after filtering. Also note that the words in the sentence have been lexically disambiguated by the parsing process.

Our approach to handling lexical ambiguity can easily be extended to handle multiple word candidates in the same time interval. In this case, each role value keeps track of its word candidate, as well as its lexical category. This extension is a first step toward processing multiple sentence

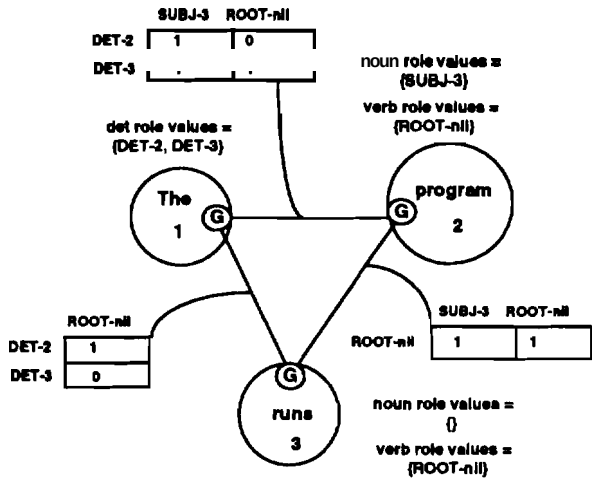


Figure 19: The CN after binary constraint propagation.

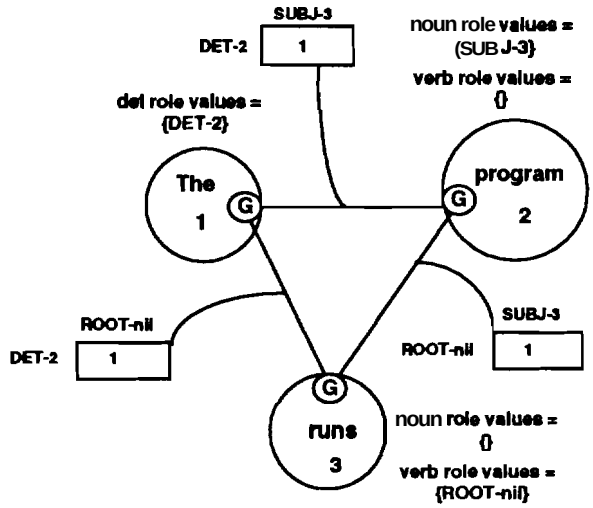


Figure 20: The CN after filtering.

		L: SUBJ	ROOT	DET
C:	noun	1	0	0
	verb	0	1	0
	det	0	0	1

Figure 21: A table of legal labels for word categories in the governor role for G_1.

hypotheses provided by a speech recognizer. In Section 4, we describe the necessary modifications of the CDG parsing algorithm for processing general word graphs.

3.2 An Efficiency Issue: Label Pruning Using a Table

Given that the role values assigned to a role for a word in CDG are affected by both the role and the parts of speech, it is possible, at network construction time, to restrict the role values assigned to a role for each part of speech for a word to those that are appropriate for the role and the lexical category under consideration. To do so, we add a fifth parameter to the CDG grammar tuple, T, where T is a ***table*** which restricts the possible labels for each role according to the category of the word and its role id. Now a CDG grammar consists of a quintuple, (C, R, L, C, T). Though T is not a necessary aspect of the grammar, it does make the analysis of a sentence more efficient because the roles are initialized to smaller domains, and many of the unary constraints (i.e., those which restrict labels of role values to lexically appropriate values) can be omitted. The table for augmenting grammar G_1 is shown in Figure 21. It shows the legal labels for the governor role given the word categories in Σ_1. If this table is used during CN construction for the sentence ***The program runs*** along with the assumption that no word modifies itself, the resulting CN is depicted in Figure 22. See Appendix A for the pseudocode for CN initialization using T.

In practice, the table reduces the number of role values in the initial CN by a factor of five to seven, and eliminates the need to propagate some unary constraints. Hence, it does affect the actual running time of the CDG algorithm, though it does not improve the asymptotic running time.

3.3 Lexical Features in CDG

Many times, even if a word is not lexically ambiguous, it can have ambiguity in the feature information associated with the word, like number, person, or case. For example, the noun ***fish*** can take the number/person feature value of third person singular or third person plural. Lexical features are

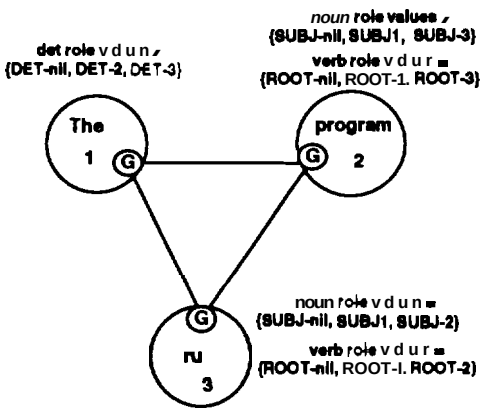


Figure 22: The initial CN given the table in Figure 21

used in many natural language parsers to enforce subject-verb agreement, determiner-head noun agreement, and case requirements for pronouns. This information can be very useful for disambiguating parses for sentences or for eliminating impossible sentence hypotheses, hence we have also added lexical feature analysis to the CDG parser

The incorporation of feature tests into CDG parsing requires the same special care used to introduce lexical ambiguity into the parsing algorithm. Consider another simple example grammar[11] $G_3 = \langle \Sigma_3, R_3, L_3, C_3, T_3 \rangle$, shown in Figure 23.

We will consider how to add number/person feature tests to CDG to process the sentence *A ***fish eat the worm.*** The grammar should not generate this sentence because it is ungrammatical given appropriate number/person tests. To add number/person feature tests to CDG grammars, the algorithm for constructing the initial CN must be modified to look up number/person information for the word and store this information with the role values. The lexical entries for this example follow:

```
(a (category det (number 3s)))
(eat (category verb (number 1s 2s 1p 2p 3p)))
(fish (category noun (number 3s 3p)))
(the (category det (number 3s 3p)))
(worm (category noun (number 3s)))
```

To simplify the example, we assume that ***fish*** is not lexically ambiguous. In addition to storing number/person information in the lexicon, two agreement constraints must be added to G_3 to ensure that the number of the determiner agrees with the number of the head noun and that the number of the SUBJ agrees with the number of the ROOT.

[11] This grammar was designed to illustrate feature testing, and is not a general constraint-based grammar. Among other things, it is missing constraints to support auxiliary verb structures in the sentences it accepts.

Σ_3 = {det , noun, verb).
R_3 = {governor)
L_3 = {DET, SUBJ, DO, ROOT)
T_3 = see Figure 24.
C_3 = V x y role (and

```
;; [U-1] A DET modifies a word to its right.
(if (eq (lab x) DET)
    (lt (pos x) (mod x)))
;; [U-2] A SUBJ modifies a word to its right.
(if (eq (lab x) SUBJ)
    (lt (pos x) (mod x)))
;; [U-3] A DO modifies a word to its left.
(if (eq (lab x) DO)
    (gt (pos x) (mod x)))
;; [U-4] A ROOT modifies no word.
(if (eq (lab x) ROOT)
    (eq (mod x) nil))
;; [B-1] A DET is governed by a SUBJ or DO.
(if (and (eq (lab x) DET)
         (eq (mod x) (pos y)))
    (or (eq (lab y) SUBJ)
        (eq (lab y) DO)))
;; [B-2] A SUBJ and DO are governed by a ROOT.
(if (and (or (eq (lab x) SUBJ)
             (eq (lab x) DO))
         (eq (mod x) (pos y)))
    (eq (lab y) ROOT)))
;; [B-3] A DET to the left of a ROOT must modify a noun to the
;; left of the ROOT.
(if (and (eq (lab x) DET))
         (eq (lab y) ROOT)
         (lt (pos x) (pos y)))
    (lt (mod x) (pos y)))
)
```

Figure 23: $G_3 = \langle \Sigma_3, R_3, L_3, T_3, C_3 \rangle$.

L:

C:	SUBJ	DO	ROOT	DET
noun	1	1	0	0
verb	0	0	1	0
det	0	0	0	1

Figure 24: A table of legal labels for word categories in the governor role for G_3.

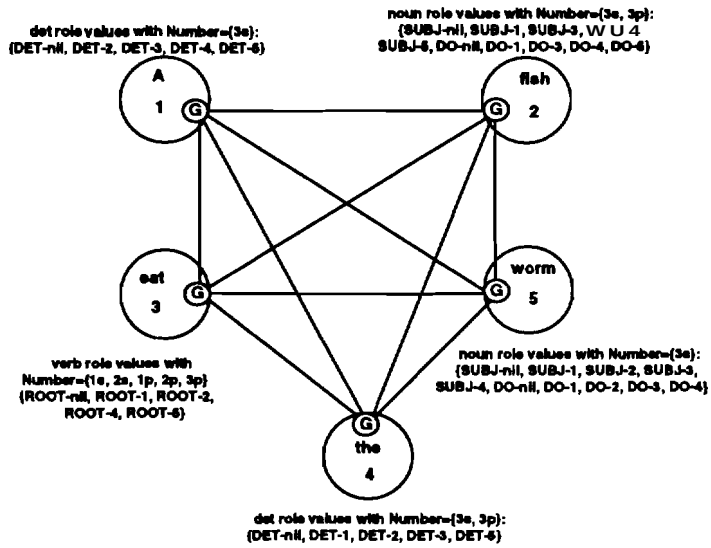


Figure 25: The initial CN given G_3.

```
;; [B-4] A DET which is governed by a noun must agree in number
;; with the noun's number.
(if (and (eq (lab x) DET)
         (eq (cat y) noun)
         (eq (mod x) (pos y)))
    (agree (number x) (number y)))

;; [B-5] A SUBJ which is governed by a ROOT must agree in number
;; with the verb's number.
(if (and (eq (lab x) SUBJ)
         (eq (lab y) ROOT)
         (eq (mod x) (pos y)))
    (agree (number x) (number y)))
```

These constraints require the addition of one access function **number** and a predicate **agree.** The function **(number x)** returns the number/person information associated with the role value, and the predicate **(agree (number x) (number y))** returns true only if its two number/person arguments agree. We consider two ways to store number/person information with a role value. One way is to store the entire set of features with each role value. In this case, agree returns true iff the intersection of the two number sets is non-empty. The second approach is to store one number/person feature value per role value, and the agree predicate becomes equivalent to an equality test.

If the CDG parsing algorithm stores the set of number/person feature values with each role value, the CN depicted in Figure 25 for the sentence *A ***fish eat the worm*** is constructed. Following the propagation of unary constraints, the network is as depicted in Figure 26. After the binary constraints B-1, B-2, and B-3 are propagated and the network is filtered, the CN is in the state depicted in Figure 27. This figure highlights the matrices corresponding to the arcs that are most relevant to the agreement constraints, B-4 and B-5. Now consider the impact of constraints B-4 and

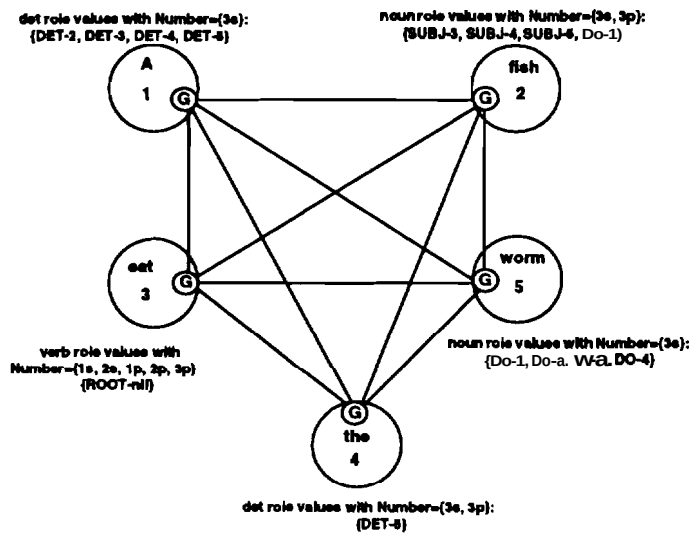


Figure 26: The CN after the propagation of unary constraints.

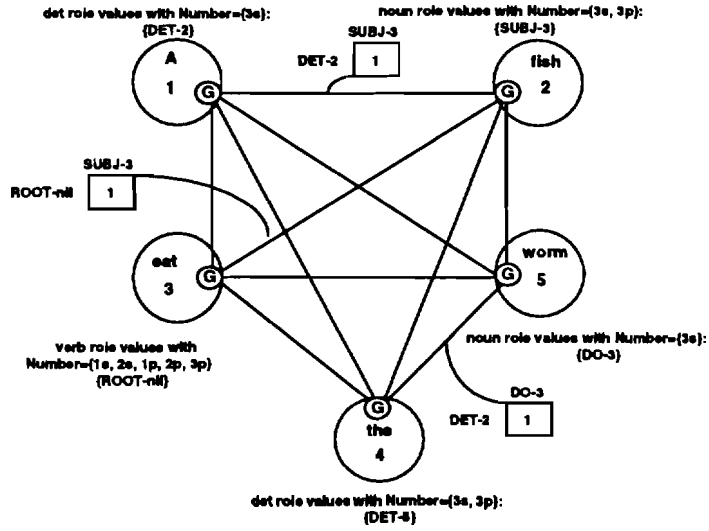


Figure 27: The CN after propagating B-1, B-2, and B-3 and filtering.

B-5 on the network, and notice that the constraints succeed for the CN, despite the fact that the sentence is ungrammatical. This occurs because the words are checked pairwise for agreement. The word a agrees with fish, and the word fish agrees with eat, but the numbers that cause agreement on the two arcs are incompatible with each other. Using this approach, the only way to ensure that sets of numbers jointly agree for the determiner, subject, and verb is by propagating an agreement constraint over the three role values. This constraint would contain three variables[12] as shown below:

```
;; A DET that is governed by a SUBJ, which is governed by a ROOT
;; rust agree with the ROOT also.
(if (and (eq (lab x) DET)
         (eq (lab y) SUBJ)
         (eq (lab z) ROOT)
         (eq (rod x) (pos y))
         (eq (rod y) (pos z)))
    (agree (number x) (number z)))
```

To propagate this constraint requires the addition of arcs linking triples of roles in the sentence and the use of three dimensional arc matrices. Because there are $\binom{n*p}{3} = O(n^3)$ arcs required in a CN with 3-variable constraints, and each arc contains a matrix with $(q * n)^3 = O(n^3)$ elements, the time to construct the arcs and initialize the matrices is $O(n^6)$, and the time to propagate a three variable constraint is $O(n^6)$. This constraint will work for the current example, but to handle four-way agreement for sentences like *The fish which are eating swims would require constraints with an arity of four. Because of cases like these, we have developed another approach to feature testing.

To correctly utilize number and person features in agreement tests for CDG parsing without resorting to greater than two-variable constraints, each role value must be assigned a single feature value, not a set of values. If there is more than one feature value, then the role values are duplicated for each feature value. Given this modification, the initialization of the CN for *A. fish eat the worm is shown in Figure 28. Figure 29 depicts the CN after unary constraint propagation, and Figure 30 shows the state of the network after binary constraints B-1, B-2, and B-3 have been propagated and the network has been filtered. After applying constraints B-4 and B-5 to the CN in Figure 30, the matrix entries indexed by role values with incompatible feature values are set to 0, as shown in Figure 31. When this network is filtered, there are no remaining role values (see figure 32), and so the sentence is not generated by the grammar.

If there are two feature types (say number/person and case) to be used in constraints for a grammar, then the role values will have to be duplicated and assigned feature values from the cross product of the features' values. This could easily lead to a combinatorial explosion of role values.

[12] One might expect that the illegal parse would be detected during backtracking search, but this assumption would be incorrect. The sets of features are unaffected by parsing in this approach, and there is no way to selectively require non-empty feature intersection for sets of tree roles.

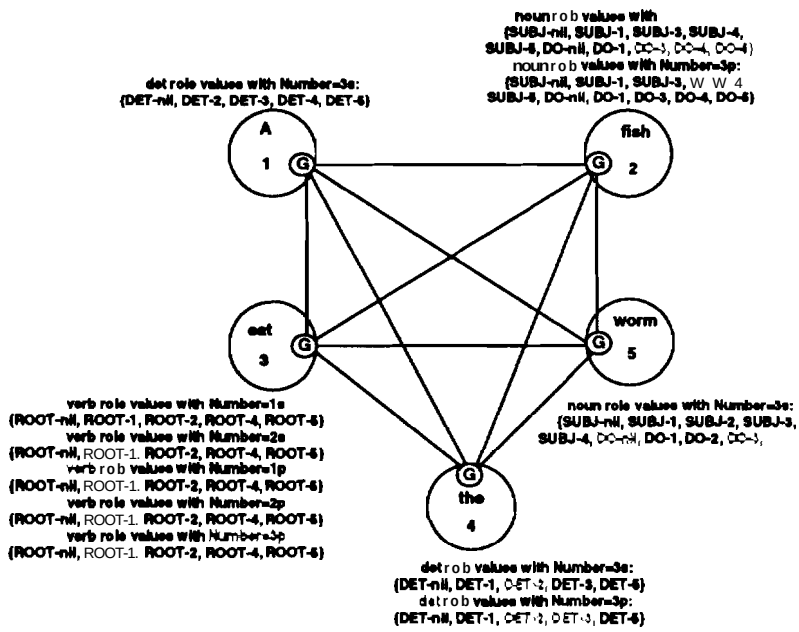


Figure 28: The initial CN given that each role value has one feature value.

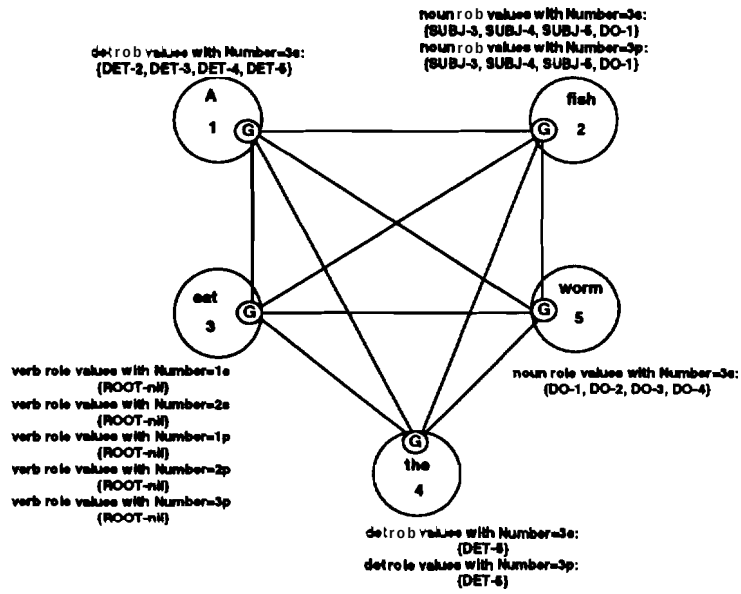


Figure 29: The CN after the propagation of unary constraints.

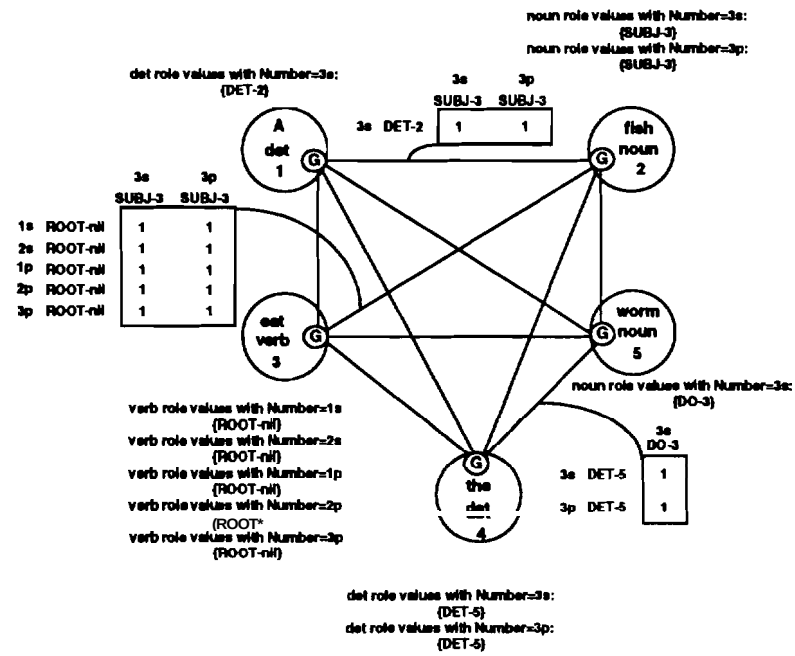


Figure 30: The CN after propagating B-1, B-2, and B-3 and filtering.

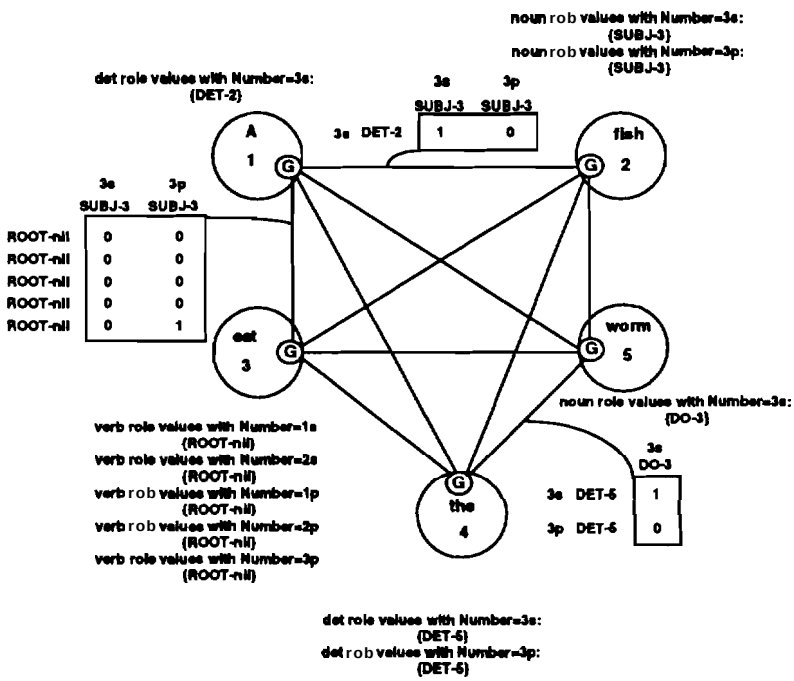


Figure 31: The CN after propagating B-4 and B-5.

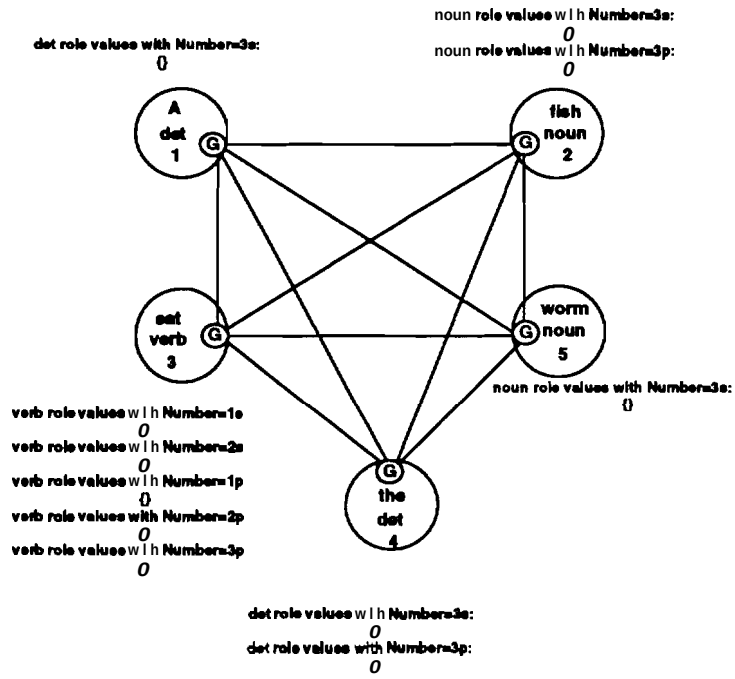


Figure **32:** The CN for an ungrammatical sentence after filtering.

Fortunately, there is an excellent strategy for limiting the number of role values. The basic idea is to store the sets of feature values with a single role value and to duplicate the role values only on demand, when a particular feature type is being tested by a constraint. A grammar writer can then order constraints in a constraint file in such a way that role values are reduced by pure phrase structure constraints prior to the feature constraints. Also, feature constraints can be ordered to minimize useless role value duplication. When the parser is preparing to propagate a constraint with a particular feature test, each of the role values having multiple values for that feature is duplicated and assigned one of the feature values. The corresponding feature constraints should then eliminate many of the duplicated role values before other types of feature constraints are propagated.

Consider how the sentence *A ***fish eat the worm*** is processed given this strategy, assuming that constraints are propagated in the order they appear in our grammar. The initial CN is constructed as depicted in Figure **25.** Once the non-feature constraints have been propagated and filtering has been performed, the CN would be in the state depicted in Figure 27. Note that many of the role values have been eliminated by the constraints before the feature constraints are propagated. Now in preparation for the propagation of constraints using the number/person feature, the role values in Figure 27 must be duplicated for each number/person feature value, giving the CN in Figure **30.** After the feature constraints have been applied (Figure **31)** and filtering (Figure **32)** has been performed, no parse for the sentence remains.

4 Spoken Language Modification

CDG has several advantages that make it attractive for use with spoken language understanding systems. First, it is able to handle grammars that are beyond context-free. Second, it uses a single representation, the constraint, to encode syntactic rules and feature tests. This uniformity is especially compelling for speech understanding because such a system could potentially use lexical, syntactic, semantic, prosodic, and contextual rules. Third, CDG is able to support the use of context when determining the meaning of a sentence (especially to reduce ambiguity). Fourth, the flexibility of incremental constraint-based parsing should also be less sensitive than CFG parsers to the syntactic irregularities common in spontaneous speech. Finally, the algorithm is amenable to effective parallel implementation.

One drawback of the CDG parser as defined by Maruyama is that it is only able to process one sentence at a time. However, since a speech recognizer can generate multiple sentence hypotheses for a given utterance, a one-sentence-at-a-time parser would be very inefficient. Hence, in this section, we extend the CDG parser to process word graphs containing multiple sentence hypotheses.

In the Section 4.1, we briefly describe current spoken language approaches and motivate the use of word graphs for processing the multiple sentence hypotheses provided by a speech recognizer. In Section 4.2, we adapt the CDG parsing algorithm to operate directly on a word graph. In Section 4.3, we describe how the filtering algorithm must be modified to simultaneously process multiple sentence hypotheses. Finally, in Section 4.4, we describe our implementation of the spoken language parser and the development of two constraint-based grammars.

4.1 Current Approaches to Speech

Among the most successful current speech recognition systems which process continuous speech for a limited (1000 word) vocabulary are those which utilize hidden Markov models (HMM). Most systems utilizing this approach (e.g., [22, 40])) have reduced recognition errors by incorporating some language information (syntactic and semantic) directly into the HMM to reduce perplexity, but since the goal of these systems is recognition, not understanding, no structural analysis of the utterance is performed. Instead, the output of such systems is an ordered list of the N most likely sentence hypotheses (where N is a constant usually less than 100) (39, 41].

Graceful integration of speech recognition and natural language systems remains a difficult problem. Early systems [23, 53] grappled with knowledge source interaction and flow of control. The trend in recent systems has been to use stochastic language models [33, 56]. However, this approach is limited to relatively simple cases (e.g., bigram or trigram) in order to control network size and complexity of training. These techniques have proven promising for some speech recognition

1. Clear all windows.
2. Clear windows.
3. Clear all the windows.
4. Get all windows.
5. Give all windows.
6. Clear all of the windows.
7. Clear the windows.
8. Get all the windows.
9. Give all the windows.
10. Get all of the windows.
11. Give all of the windows.

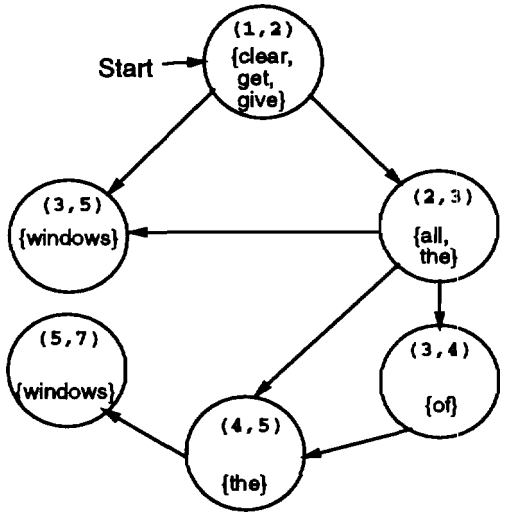


Figure 33: The word graph for the N-best command sentences.

tasks, but are inadequate for representing the complex linguistic information required to perform speech understanding. Systems that are attempting to integrate speech recognition processing with more traditional natural language processing techniques include CMU's Phoenix, using frame based parsing and semantic phrase grammars [52]; CSELT's system, based on firdte-state language models [11]; MIT's Voyager, using LR parsing [58]; and Seneff's robust parsing [42, 43].

To construct a speech understanding system which builds on current recognizers, a researcher might pass the N-best sentence hypotheses generated by a recognizer through a natural language parser as a first step toward producing meaning representations. However, processing each sentence hypothesis provided by a speech recognizer individually is inefficient since the sentence hypotheses often differ only slightly from each other. Furthermore, a list of sentence hypotheses is not the most compact representation to provide a natural language parser. A better representation for the sentence hypotheses is a word graph or lattice of word candidates which contains information on the approximate beginning and end point of each word's utterance.

We have conducted an experiment which demonstrates the compactness of a word graph. For this experiment, we selected three sets of N-best sentence hypotheses[13] for three different types of utterances: a command, a yes-no question, and a wh-question. The list of the N-best sentences was converted to a word graph in which the duration of the node was approximated by using the syllable count for the words in the utterance. Figure 33 depicts a set of N-best sentences and the word graph our algorithm constructed for those sentences. In figure 34, the size and expressive power of the constructed word graphs is compared with N-best sentence lists. The word graphs were more expressive than the N-best sentence lists while providing an 83% reduction in storage.

[13] We thank BBN for providing us with the N-best lists of sentences.

Sentence Type	Number of N-Best Sentences	Number of N-Best Words	Distinct N-Best Words	Number of Graph Nodes	Words in Graph	Sentences in Graph
Command	11	41	7	6	9	21
Yes-No-Q	20	129	17	11	18	48
Wh-Q	20	133	19	11	19	432

Figure 34: N-best sentences versus word graphs.

Even though a word graph is a compact representation for the output of a speech recognition system, current systems do not provide this type of representation. However, parsers that can process the graph representation should more efficiently process all sentence hypotheses. Tomita [47] has developed an LR parsing algorithm capable of processing a word graph. Chart parsers can also process a lattice by storing the words in its chart. Though these approaches handle sentences in a lattice, the CDG approach to parsing, once extended to operate on a word graph, has advantages discussed in Section 2.3.2 which make it an even more promising approach for speech processing.

4.2 Parsing Word Graphs with Constraints

We have adapted the CDG constraint network to handle the multiple sentence hypotheses stored in a word graph, calling it a Spoken Language Constraint Network (SLCN). Figure 35 depicts an SLCN derived from a word graph constructed for the sentence hypotheses: *A fish eat and *Offices eats. By representing these hypotheses in a word graph, we are able to process additional sentences (i.e., A fish eats and Offices eat) not present in the list of hypotheses, one of which might represent the intended utterance. Notice that word nodes contain a list of all word candidates with the same beginning and end points, and edges join word nodes that can be adjacent in a sentence hypothesis (see Figure 35). A sentence hypothesis must include one word node from the beginning of the utterance, one word node from the end of the utterance, and these two word nodes must be connected by a path of edges. The word nodes along a path can contain multiple word candidates, so the number of sentence hypotheses for a particular path of edges can be quite large. In the SLCN of Figure 35, each word node contains information on the beginning and end point of the word's utterance, represented as an integer tuple (b, e), with b < e. The tuple is more expressive than the point scheme used for CNs and requires modification of some of the access functions and predicates defined for the CN scheme. The access functions (pos x) and (mod x) now return a tuple (b, e) which describes the position of the word associated with the role value x. The equality predicate must be extended to test for equality of intervals (e.g., (eq (1,2) (1,2)) should return true). The

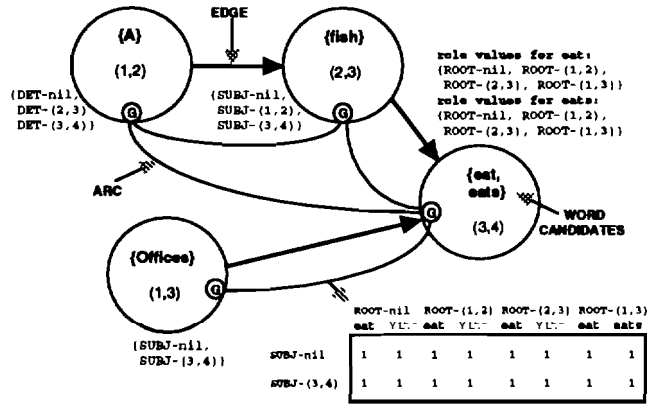


Figure 35: Example of a spoken language constraint network constructed frorn a word graph.

less-than predicate, (lt (bl, el) (b2, e2)), returns true if el < b2, and the greater than predicate, (gt (bl, el')(b2, e2)), returns true if bl > e2.

To parse an SLCN, each word candidate contained in a word node is assigned a set of role values for each role, requiring $O(n^2)$ time, where n is the number of word candidates in the graph. Unary constraints are applied to each of the role values in the network, and like CNs, require $O(k_u * n^2)$ time. The preparation of the SLCN for the propagation of binary constraints is similar to that for a CN. All roles within the same word node are joined with an arc as in a CN; however, roles in different word nodes are joined with an arc iff they can be members of at least one common sentence hypothesis (i.e., they are connected by a path of directed edges). To construct the arcs and arc mirtrices for an SLCN, it suffices to traverse the graph from beginning to end and string arcs from each of the current word node's roles to each of the preceding word node's roles (where a node precedes a node iff there is a directed edge from the preceding to the current node) and to each of the roles that the preceding word nodes' roles have arcs to. For example, there should be an arc between the roles for *a* and *fish* in Figure *35* because they are located on a path from the beginning to the end of the sentence ***a fish {eats, eat).*** However, there should not be an arc. between the roles for *a* and *offices* since they are not found in any of the same sentence hypotheses (See Appendix A for the SLCN arc construction pseudocode). After the arcs for the SLCN are constructed, the arc matrices are constructed in the same manner as for a CN. The time required to construct the SLCN network in preparation for binary constraint propagation is $O(n^4)$ because there may be up $O(n^2)$ arcs constructed, each requiring the creation of a matrix with $O(n^2)$ elements. Once the SLCN is constructed, binary constraints are applied to pairs of role values associated with arc matrix entries (in the same manner as for the CN), requiring $O(k_b * n^4)$ time, where n is the number of word candidates.

Filtering in an SLCN is complicated by the fact that the limitation of one word's function

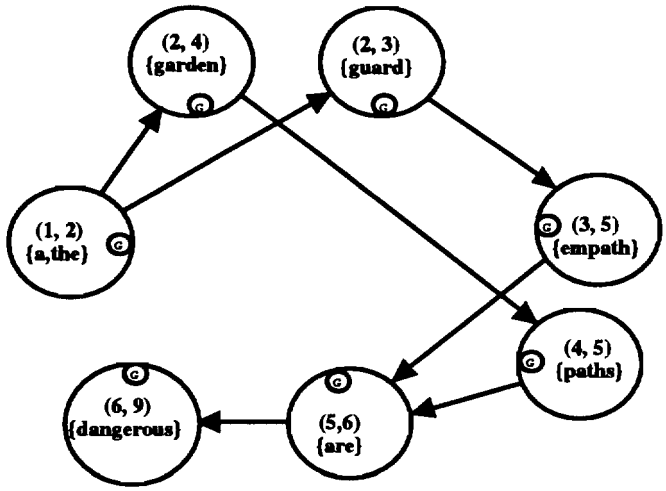


Figure **36:** A simple SLCN.

in one sentence hypothesis should not necessarily limit that word's function in another sentence hypothesis. For example, consider the SLCN depicted in Figure **36.** Even tlhough all the role values for are would be disallowed by the singular subject *empath*, those role values cannot be eliminated since they are supported by paths, the subject in a different hypothesis. To demonstrate the differences between a single sentence CN and an SLCN, we map the SLCN in Figure **36** to the AND/OR tree shown in Figure **37.** Because the SLCN is based on a parse graph containing multiple word candidates, not all of which can participate in the same sentence hypotheses, an OR node is required at the top level of the tree to represent the contribution of various word nodes to the different sentence hypotheses. Though the individual sentence hypotheses are not indicated explicitly in the SLCN (this would require exponential space in some cases), the logical presence of the OR node must be captured by the filtering algorithm for an SLCN.

4.3 SLCN Filtering

The following notational conventions are used to develop the filtering algorithm. The capital letters A, B, X, Y, U, V represent roles and the letter **r** represents a role value. Two roles A and B are arc ***connected*** if there exists an **arc(A,B)** connecting the two roles with an associated arc matrix, **arc_matrix(A,B)**. Note that **arc(A,B)** and **arc(B,A)** are the same arc.

The implicit top level OR node in the SLCN requires significant revision of the SLCN filtering algorithm. The filtering algorithm can no longer delete a role value when a single arc matrix fails to support it, because all of the words in another sentence hypothesis might support that value. Instead, we must determine how to propagate role value deletion from one arc associated with a role to other arcs associated with the same role. After propagating role value deletion over the arcs, if a role value **r** is supported by at least one arc matrix associated with an arc emanating

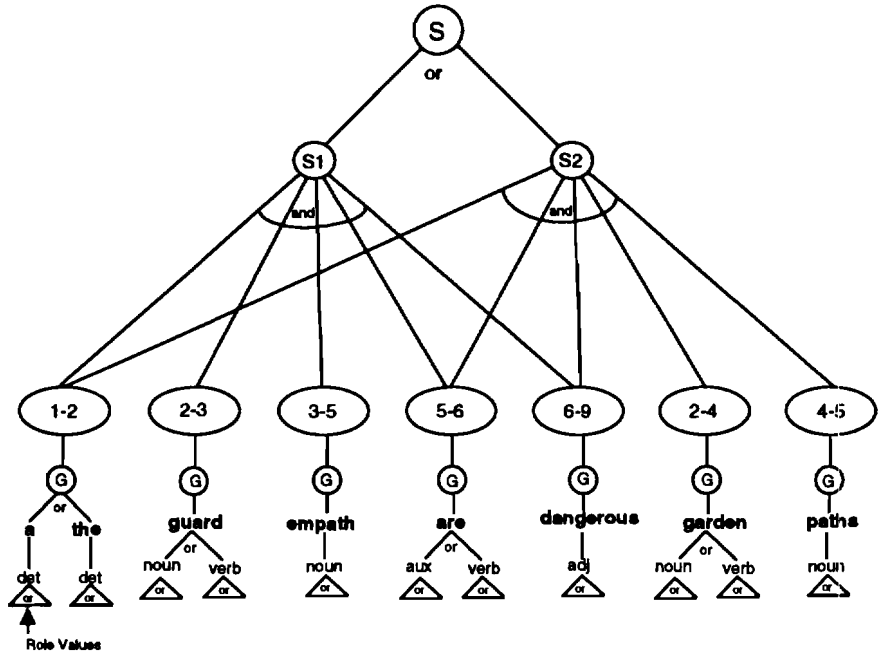


Figure 37: The AND/OR tree for the SLCN in figure 36.

from its role, then the role value cannot be eliminated from the role. However, if none of the arc matrices associated with the role's arcs supports the role value, then it should be eliminated from the role. Also, if all of the role values in a role for a particular word candidate are eliminated, then that word candidate should no longer be a supported word. If all of the word candidates for a word node are unsupported, then the word node should also be unsupported. Furthermore, word nodes which are no longer members of a legal sentence hypothesis because the only word they are adjacent to is unsupported should, through filtering, lose support.

In order to develop a correct filtering algorithm for SLCN parsing, we must consider how to propagate role value deletion from one arc associated with a role to others associated with the same role. Suppose role **A** is connected by arcs to roles B and X and the arc matrix associated with the arc between **A** and B no longer supports the role value **r** associated with role **A.** Should the elimination of $\mathbf{r} \in \mathbf{A}$ from the arc between **A** and B cause the value to be removed from the arc matrix associated with the arc between **A** and X? Should it also cause **r's** removal from the set of A's role values? Our basic strategy for developing the filtering algorithm is to group the arcs of a role into classes which will allow us to efficiently determine which arcs should be affected if a role value is disallowed by an arc matrix. We begin by introducing two axioms and some basic classes of arcs.

Filtering Axioms and Elementary Arc Classes:

Our filtering algorithm was developed given two fundamental axioms. The first axiom is shown

below:

Axiom 1 (Modifiee Axiom) *If a role value* **r** *associated with role* A *is eliminated from* **arc_matrix(A,B)** *and that role value's modifiee points to the word node containing role* B, *then it should be eliminated from all of the arc matrices associated with arcs attached to the role* A.

The role **B** has the right to directly eliminate any role values if their modifiees point to **B**'s node and none of B's role values support them. If B cannot support the role value, then none of the roles associated with other words should.

When setting up the classes of arcs (and their associated matrices), we are guided by the second axiom:

Axiom 2 (Arc Class Axiom) *An* **arc_matrix(A,B)** *should disallow* **r** *if* **r** *is not legal in any sentence hypotheses that contain* **arc(A,B)**, *i.e., there exists no path of edges from a beginning node to an ending node such that every role for every node contains at least one role value compatible with* r.

This suggests that we should group arcs into sets of arcs which are in the same sentence hypotheses. Because the topology of an SLCN is a directed acyclic graph (DAG), such sets will be recursive and hierarchical in nature. Inspection of Figure 38 leads to some initial observations about which arcs should be grouped together. Note that the small circles in this figure are roles, the large ovals are word nodes, the straight lines with arrows are edges, and the curved lines are arcs. The first class of arcs are *intm-arcs,* which are arcs that connect two roles belonging to the same word node.

Theorem 1 (intra-arcs) *If an* **arc(A,B)** *is an* intra-arc *and* **arc_matrix(A,B)** *disallows* **r** *from* A, *then* **r** *should be disallowed by* all *arcs incident to* A *and removed from the role* A.

Proof: The intra-arc **arc(A,B)** is a member of every sentence hypothesis that contains **A**'s word node; therefore, it is a member of every sentence hypothesis that includes the arcs incident to A. Hence, by Axiom 2, if **r** is eliminated by the intra-arc **arc_matrix(A,B)**, it should be removed by all of the arc matrices associated with the arcs emanating from A. Furthermore, **r** should be eliminated from the role A.

The second class of arcs are *iso-arcs.* The two arcs, **arc(A,B)** and **arc(A,C)**, are said to be iso-arcs if roles B and C are located in the same word node (i.e., they are different roles associated with the same word node) and are incident on a common role A. Figure 38 depicts a set of iso-arcs.

Theorem 2 (iso-arcs) *If* **arc(A,B)** *is a member of a set* Θ *of iso-arcs incident to* A, *and* **arc_matrix(A,B)** *disallows* **r** *from* A, *then all of the matrices associated wzth the iso-arcs in the set* Θ *should also eliminate* **r** *from* A *by zeroing the row or column indexed by* r.

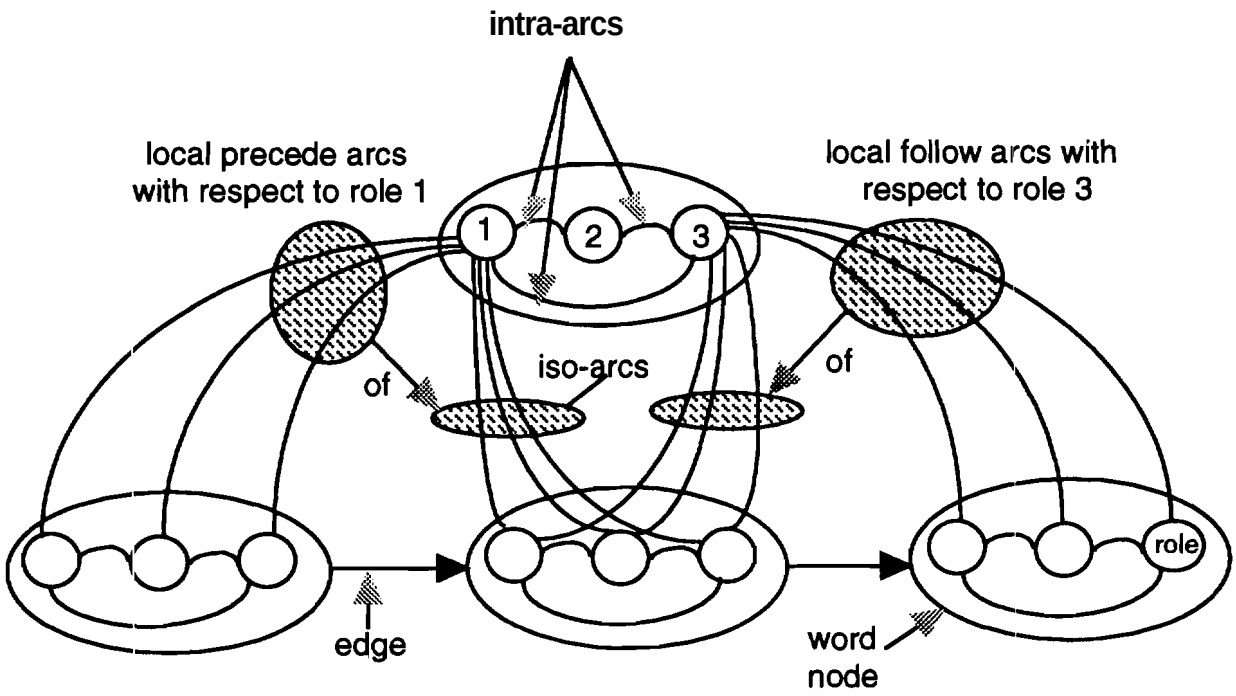


Figure 38: Illustration of the terms used in this report.

Proof: Because iso-arcs connect the same two word nodes, they are members of exactly the same set of sentence hypotheses. Hence, by Axiom 2, if one of them eliminates **r** from **A**, they all should.

If a role value $\mathbf{r} \in \mathbf{A}$ is eliminated by **arc_matrix(A,B)**, depending on the type of roles **A** and **B,** that role value may be eliminated from all the arc matrices of arcs connected to role **A,** or it may be removed from some but not others. Clearly, not all of the arcs in the netvvork are intra-arcs or iso-arcs of each other. Hence to filter an SLCN as thoroughly as possible, we must also be able to determine whether the deletion of a role value in **A** from **arc_matrix(A,B)** should affect the deletion of the same role value for other matrices corresponding to other types of arcs emanating from **A.** Figure 39 depicts the temporal dependency of **arc(A,B')** on **arc(A,X')** and **arc(A,B)** on **arc(A,X)** in a restricted view of an SLCN. Assume for simplicity that there is only one role per word node for this discussion. In Figure 39, **arc(A,X)** provides **local precede arc support** for the role values of **A** for **arc(A,B)** because there exists a directed edge joining the word node containing role X to the word node containing role **B.** Also, **arc(A,X')** provides **local follow arc support** for the role values of **A** for **arc(A,B')** because there is a directed edge joining the word node containing role **B'** to the word node containing role **X'.** Furthermore, **arc(A,B)** provides local follow support for the role values of **A** for **arc(A,X)** because of the directed edge between the word nodes for roles X and **B,** and **arc(A,B')** provides local precede support for the role values of **A** for **arc(A,X')** because of the directed edge between the word nodes for roles B' and **X'.**

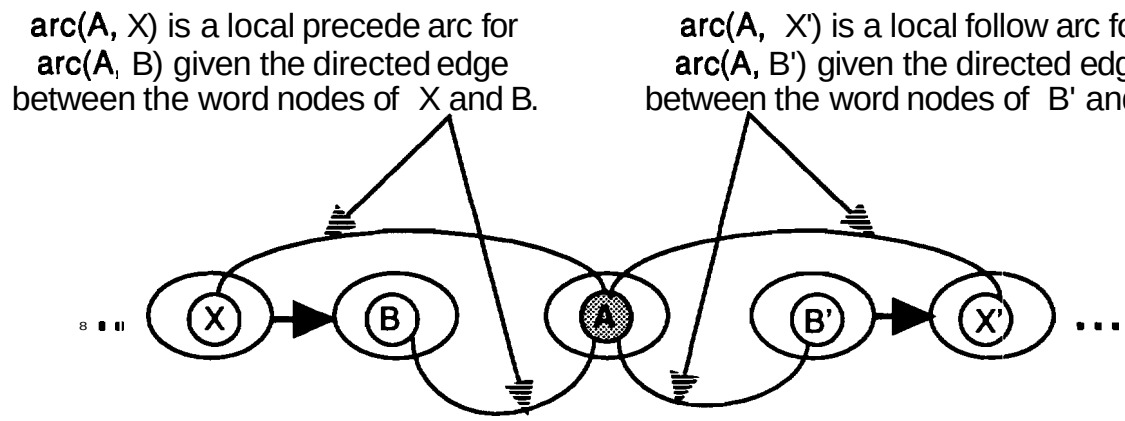
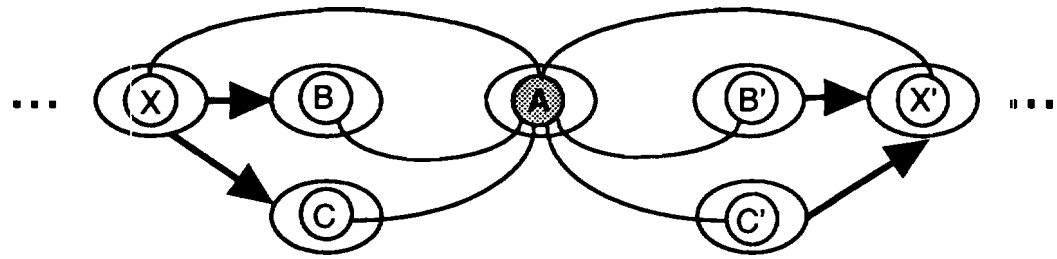


Figure 39: Temporal Dependencies between the arcs for role A.



arc(A, B) and arc(A,C) both provide local follow support for a role value on arc(A, X). Hence, both must remove support for a role value in A for that role value to be disallowed by arc(A, X).

arc(A, B') and arc(A,C') both provide local precede support for a role value on arc(A, X'). Hence, both must remove support for a role value in A for that role value to be disallowed by arc(A, X').

Figure 40: Temporal Dependencies between the arcs for role A.

If **arc_matrix(A,B)** no longer supports **r** ∈ A, then **arc_matrix(A,X)** may no longer have reason to support that role value because of the loss of the local follow support.. If all paths from X's word node to A's word node must include the edge from **X**'s to **B**'s word node, then all of the sentence hypotheses containing **arc(A,X)** must also contain role B, and hence, **arc(A,B)**. In this case, the role value should be deleted from **arc_matrix(A,X)** during filtering. However, if there exists a path of edges from **X**'s word node to **A**'s word node that does not include the edge from **X**'s word node to **B**'s, as in Figure 40, then **arc_matrix(A,X)** loses some support for **r** ∈ A if **r** is deleted from **arc_matrix(A,B)**, but so long as **arc_matrix(A,C)** supports that role value, it cannot be deleted from **arc_matrix(A,X)**.

Symmetrically, if **arc_matrix(A,B')** no longer supports **r** ∈ A, then **arc_matrix(A,X')** may no longer have reason to support that role value. If the edge from the word node of B' to the word node of X' is a required link on all paths of edges from the word node of A to the word

node of X', then all of the sentence hypotheses containing **arc(A,X')** must also contain role B' and **arc(A,B')**. Hence, the role value would be deleted from **arc_matrix(A,X')** during filtering. On the other hand, if the edge from the word node of B' to the word node of X' is not a required link on all paths of edges from the word node of A to the word node of X', as in Figure 40, then **arc_matrix(A,X')** loses some support for r ∈ A if **arc_matrix(A,B')** delete!; r, but so long as **arc_matrix(A,** C') supports that role value, it cannot be deleted from **arc_matrix(A,** X').

Local precede arcs and follow arcs provide two additional useful classes of arcs for the filtering algorithm, as the following theorem shows:

Theorem 3 (local precede and follow arcs) ***If a role value*** r ∈ A ***for*** **arc(A,B)** ***is disallowed from all of*** **arc(A,B)*'s local precede arcs with respect to*** A ***or all of its local follow arcs with respect to*** A, ***then it should be disallowed by*** **arc(A,B)**.

Proof: Every role value r ∈ A for **arc(A,B)** must be allowed by at least one of the local follow arcs of **arc(A,B)** with respect to A to be a legal role value in at least one sentence hypothesis. Therefore, if all of the local follow arcs of **arc(A,B)** with respect to A have disallowed r, then r is incompatible with all of **arc(A,B)**'s sentence hypotheses. A symmetric argument holds for local precede arcs.

In order to create a correct algorithm for filtering an SLCN, we must determine which arcs temporally support the continued existence of a role value so that if the support is removed, the role value can be deleted. To utilize the temporal dependencies between arcs, we must associate a set of local precede and follow supporters with each role value on an arc. If an arc matrix for an arc eliminates a role value, then it must remove support for that role value from its local precede and follow arcs. If the role value on one of those arcs has that arc as its only local precede or follow supporter, then the role value must be removed from that arc's matrix and the arc must then remove support for the role value from its local precede and follow arcs. On the other hand, if the role value on each of those arcs has more than one local precede or follow supporter for a role value, then the loss of support must be recorded in some way even thoughlh the role value is not deleted.. In the next section, we develop the filtering algorithm for SLCNs using the insights described in this section.

The SLCN Filtering Algorithm:

To implement the filtering algorithm, we must add information to the constraint network. A role value of a role may not be eliminated until all of its arc's matrices disallow that value. To determine whether the role values for a role are supported by the network, we keep a count of all of the arcs supporting each of the role values for each of the roles. If the support count drops to zero, then the role value is eliminated since none of its arcs support that role value. Additionally, the elimination

of a role value in one arc matrix can cause the elimination of the role value from other arc matrices. Below, we enumerate the list of cases for propagating role value elimination to other arcs matrices.

1. If a role value, **r**, associated with the role A is eliminated from the intra-arc **arc_matrix(A,B)** (say between A, the governor role for a word node, and B, the needs role for the same word node), it must be eliminated from **all** of the arc matrices associated with the arcs emanating from the role A.

2. If a role value, **r**, associated with role A is eliminated from **arc_matrix(A,B)** and that role value's modifiee points to the word node containing role B, then it should be eliminated from all of the arc matrices associated with arcs attached to the role A.

3. If a role value, r, associated with the role A is eliminated from **arc_matrix(A,B)**, it must also be eliminated from the arc matrices of its iso-arcs. For example, if B is the governor role for a word node, then for **all** the other roles, X, associated with that word node, the role value must also be eliminated from the arc matrices associated with the arcs between A and X.

4. If a role value, r, associated with the role A is eliminated from **arc_matrix(A,B)** and neither condition 1 nor condition 2 from above holds, then support for that role value should be removed for that role value on the arc matrices associated with the local precede or follow arcs of **arc(A,B)** given A. To determine whether **r** should be eliminated by any of these arc matrices, we must determine whether they have additional support for **r** once **arc(A,B)**'s support is removed.

To keep track of which arcs temporally influence other arcs in the the elimination of role values, we must determine for each of the roles on each arc in the SLCN which arcs are local precede and follow arcs. Each **arc_matrix(A,B)** must maintain a list of local precede arcs and a list of local follow arcs for each of the role values associated with roles A and B.

- A-precede-support- The role values associated with role A for **arc_matrix(A,B)** each maintain a list of local precede support arcs with respect to B containing:
  - All arcs of the form **arc(A,X)**, where there exists a directed edge from the word node containing X to the word node containing B. The directed edge implies X's word node precedes **B's** word node, and so **arc(A,X)** is a local precede supporter for A's role values given B.
  - All arcs of the form **arc(A,B)**, where there exists a directed edge from the word node containing A to the word node containing B. The directed edge implies the word node

for A precedes the word node for B, hence **arc(A,B)** is a local precede supporter for its own role values given B.

- A dummy arc, say **arc(B,B)**, if B's word node is the first word in the sentence.

- A-follow-support- The role values associated with role A for **arc_matrix(A,B)** each maintain a list of local follow support arcs with respect to B containing:
  - All arcs of the form **arc(A,X)**, where there exists a directed edge from the word node containing B to the word node containing X. The directed edge implies X's word node follows **B's**, and so **arc(A,X)** is a local follow supporter for A's role values.
  - All arcs of the form **arc(A,B)**, where there exists a directed edge from the word node containing B to the word node containing A. The directed edge implies A's word node follows **B's**, hence **arc(A,B)** is a local follow supporter for its own role values.
  - A dummy arc, say **arc(B,B)**, is included in the set if B's word node is the last word in the sentence.

- B-precede-support- The role values associated with role B for **arc_matrix(A,B)** each maintain a list of local precede support arcs with respect to A containing:
  - All arcs of the form **arc(B,X)**, where there exists a directed edge from the word node containing X to the word node containing A.
  - All arcs of the form **arc(A,B)**, where there exists a directed edge from the word node containing B to the word node containing A.
  - A dummy arc, say **arc(A,A)**, if A's word node is the first word in the sentence.

- B-follow-support- The role values associated with role B for **arc_matrix(A,B)** each maintain a list of local follow support arcs with respect to A containing:
  - All arcs of the form **arc(B,X)**, where there exists a directed edge from the word node containing A to the word node containing X.
  - All arcs of the form **arc(A,B)**, where there exists a directed edge from the word node containing A to the word node containing B.
  - A dummy arc, say **arc(A,A)**, if A's word node is the last word in the sentence.

The algorithm in Figure 41 is used to calculate the local precede and follow arcs for each of the roles on each arc. Word nodes in the network are initially assigned an arbitrary index between one and n, where n is the number of nodes required to represent the SLCN. The word node for an index can be retrieved by using the function get_word_node(index), which returns the word

node associated with the index. Note that the store-support routine duplicates and stores the local precede and follow sets for each of the role values associated with a role on an arc matrix. This operation requires $O(n^2)$ time because each set can contain $O(n)$ arcs, which must be copied and stored in support of $O(n)$ role values.

Once the SLCN network is created and local role value arc support is calculated, we can propagate constraints as described earlier and then filter the network much as for a CN. We must first determine which arc matrices disallow a role value; however, when a role value is disallowed by an arc matrix, it may or may not be eliminated from all arcs associated with the role. See Appendix A for the full filtering algorithm. This algorithm relies on the two routines in Figures 42 and 43. The first routine is performed if **arc_matrix(A,B)** deletes **r** from A. If a role value is eliminated from an arc matrix during filtering, support must also be removed from that role value on all of the local precede and follow arcs for that arc matrix. Hence, the first routine invokes the second (Figure 43) to handle the removal of local arc support for a role value. If all of the local precede arcs or the local follow arcs for a role value no longer support it, then it should be eliminated from the row or column associated with the role value maintaining the list of supporters in the matrix. By using the dummy arc supporters for word nodes at the beginning and end of the sentence, we are able to detect when a role value loses support by examining whether either its local precede or follow sets become empty. If at any time the local precede or follow arc set of support becomes empty, then the role value must be deleted from the arc because there is no sentence hypothesis which supports the role value. If the role value is eliminated by the arc matrix, then its support for the role value must also be removed from those arcs it locally supports.

There are $O(n^3)$ role values indexing rows or columns in arc matrices that can be deleted by SLCN filtering, where n is the number of word candidates. Each of the indices may require the removal of $O(n)$ arc supporters from their local precede and follow lists before being deleted. Once all support is removed from a role value index, it may need to zero $O(n)$ elements in its arc matrix and remove support from $O(n)$ role value indices associated with the other arcs incident to its role. Hence, the running time of the SLCN filtering algorithm is $O(n^4)$.

To illustrate SLCN filtering, we provide a simple example. Consider the local precede and follow support sets constructed for the role values in the roles joined by each arc of the simple SLCN depicted in Figure 44. The support sets were calculated using the **create_matrix_support** procedure. The reader should keep in mind that **arc(1,2)** and **arc(2,1)** are equivalent, so we store support arcs with the convention that the lowest index is always first.

To illustrate how the procedure disallow propagates role value elimination. correctly, we will consider two types of role value elimination, one which eliminates a deleted role value from all arcs emanating from its role and one which simply removes support of the role value from its local arcs.

```
create_matrix_support – an O(n^4) operation.
Procedure create_matrix_support (){
   For index1 = 1 to n – 1 do {
      wl = get_word_node(index1);
      For I11 = each of the roles in wl do {
         For index2 = index1+1 to n do {
            w2 = get-wordnode(index2);
            For R2 = each of the roles in w2 do {
               precede-support1 = get-precede-support(R1,R2);
               precede-support2 = get-precede-support(R2,R1);
               follow-support1 = get-follow-support(R1,R2);
               follow-support2 = get-follow-support(R2,R1);
               /* Store the local precede and follow support for each role value in R1 and R2 */
               store-support(precede-support1, follow-supportl, R1, arc_matrix(R1,R2));
               store-support(precede-support2, follow-support2, R2, arc_matrix(R1,R2))}}}}
}


get-precede-support – an O(n) operation.
Function get-precede-support(For: role; Given: role){
    precede-support = {};
    If (edge(word-node-of(For), word-node-of(Given)) exists) then
        Add arc(For, Given) to precede-support;
    If (Given is a role for a word node at the beginning of a sentence) then
        Add arc(Given, Given) to precede-support;
    /* Loop requires O(n) time */
    For (each directed edge of the form edge(WORD, word-node-of(Given))
         with WORD ≠ For) do {
            For X = each of the roles in WORD do {
                Add arc(For, X) to precede-support));
    Return(precede-support);
}


get-follow-support – an O(n) operation.
Function get-follow-support(For: role; Given: role){
    follow-support = {};
    If (edge(word-node-of(Given), word-node-of(For)) exists) then
        Add arc(For, Given) to follow-support;
    If (Given is a role for a word node at the end of a sentence) then
        Add arc(Given, Given) to follow-support;
    /* Loop requires O(n) time */
    For (each directed edge of the form edge(word-node-of(Given), WORD)
         with WORD ≠ For) do {
            For X = each of the roles in WORD do {
                Add arc(For, X) to follow-support));
    Return(follow-support);
}
```

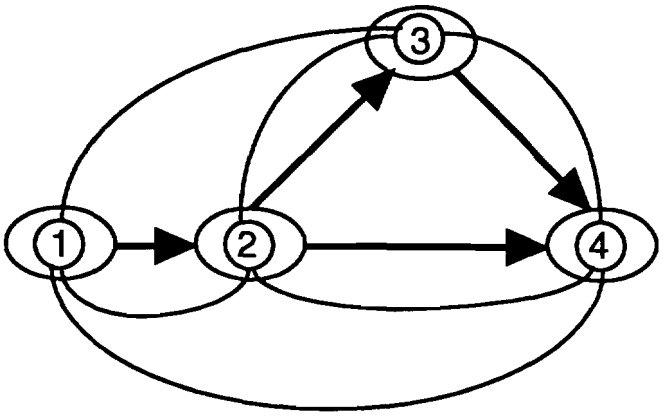
Figure 41: Local precede and follow support routines.

```
Procedure, disallow(arc_matrix(A,B): arc-matrix; A: role; r: role-value){
    If arc_matrix(A,B) is an intra-arc of A then{
        /* :Disallow r for all arcs incident to role A. Check all roles          */
        /* connected to A but not equal to B to see if their role values         */
        /* are still supported. If not add them to the to-be-deleted list.       */
        For R1 = all roles not equal to role A or B do {          /* O(n) iterations */
            remove r from the arc_matrix(A,R1);                   /* O(n) operation */
        For r-index = each of the role values in R1 do {          /* O(n) iterations. */
                if   (the role value r-index for role R1 is not   /* O(1) operation */
                      supported by arc_matrix(A,R1)) then
                         add r-index to the to-be-deleted-list;));
        arc_support_count(r,A) = 0;
        Eliminate r from role A;
        Return();
    };
    If the niodifiee of r == the position of B {
        /* :Disallow r for all arcs incident to role A. Check all roles          */
        /* connected to A but not equal to B to see if their role values         */
        /* are still supported. If not add them to the to-be-deleted list.       */
        For R1 = all roles not equal to role A or B do {          /* O(n) iterations */
            remove r ∈ A from the arc_matrix(A,R1);               /* O(n) operation */
            For r-index = each of the role values in R1 do {      /* O(n) iterations */
                if   (the role value r-index for role R1 is not   /* O(1) operation */
                      supported by arc_matrix(A,R1)) then
                         add r-index to the to-be-deleted-list;}};
        arc_support_count(r,A) = 0;
        Eliininate r from role A;
        Return();
    };
    /* Remove r from the iso-arcs of arc(A,B) */
    For R1 = all roles in A's word node not equal to A do {
            remove r from arc_matrix(A,R1);                       /* O(n) operation */
            arc_support_count(r,A) = arc_support_count(r,A) - 1;
            For r-index = each of the role values in R1 do {      /* O(n) iterations */
                if   (the role value r-index for role R1 is not   /* O(1) operatioil */
                      supported by arc_matrix(A,R1)) then
                         add r-index to the to-be-deleted-list;}};
    arc_support_count(r,A) = arc_support_count(r,A) - 1;
    /* Eliminate support for the role value r from all local precede and follow arcs */
    Inform-delete-update(arc_matrix(A,B), A, r);
    /* If no arc supports the role value r then it is eliminated from the role. */
    If (arc_support_count(r,A) = 0) then eliminate r from A;
}
```

Figure 42: **disallow(arc_matrix(A,B), A, r)** is invoked whenever **r** is disallowed for role **A** by **arc_matrix(A,B)**.

```
Procedure inform-delete-update(arc_matrix(U,V): arcmatrix; U: role; r: role-value){
        precede-list = precede-list-of(r, arc_matrix(U,V), U);
        For arc = each of the arcs in precede-list do {
                arc-matrix = the arc matrix associated with arc;
                R1 = get-role1-of-arc(arc);
                R2 = get-role2-of-arc(arc);
                If (R1 ≠ R2 && arc ≠ arc(U,V)) then {
                        /* Remove arc(U,V) as follow support for r ∈ U on arc-matrix. */
                        Remove-follow-support(arc(U,V), arc-matrix, U, r);
                        If unsupported(r, arc-matrix) then {
                                remove r ∈ U from the arc_matrix(U,V);          /* O(n) operation */
                                disallow(arc_matrix, U, r);))
        };
        precede-list-of(r, arc_matrix(U,V), U) = {};
        follow-list = follow-list-of(r, arc_matrix(U,V), U);
        For arc = each of the arcs in follow-list do {
                arc-matrix = the arc matrix associated with arc;
                R1 = get-role1-of-arc(arc);
                R2 = get-role2-of-arc(arc);
                If (R1 ≠ R2 && arc ≠ arc(U,V)) then {
                        /* Remove arc(U,V) as precede support for r ∈ U on arc-matrix. */
                        Remove-precede-support(arc(U,V), arc-matrix, U, r);
                        If unsupported(r, arc_matrix) then {
                                remove r ∈ U from the arc_matrix(U,V);          /* O(n) operation */
                                disallow(arc_matrix, U, r);))
        }
        follow-list-of(r, arc_matrix(U,V), U) = {};
}
```

Figure 43: Update the local and precede arcs for **r** in role **U** associated with **arc_matrix(U,V)**.




```
arc(1,2):                          arc(3,4):
1 precede: (arc (1,2))             3 precede: {arc(3,4), arc(2,3))
1 :follow: {arc(1,3), arc(1,4))    3 follow:  {arc(4,4)}
2 precede: (arc (1,1))             4 precede: {arc (2,4))
2 follow:  (arc (1,2))             4 follow:  (arc (3,4))

arc(2,3):                          arc(1,3):
2 precede: {arc(2,3)}              1 precede: {arc(1,2)}
2 follow:  (arc (2,4))             1 follow:  {arc(1,4)}
3 precede: (arc (1,3))             3 precede: {arc(1,1)}
3 follow:  {arc(2,3), arc(3,4)}    3 follow:  (arc (2,3))

arc(2,4):                          arc(1,4):
2 precede: {arc(2,4), arc(2,3)}    1 precede: {arc(1,2), arc(1,3)}
2 follow:  (arc (4,4)}             1 follow:  {arc (4,4)}
4 precede: (arc (1,4))             4 precede: {arc(1,1)}
4 follow:  {arc(3,4), arc(2,4)}    4 follow:  {arc(2,4)}
```

Figure 44: Temporal dependencies between the arcs of a simple SILCN.

Suppose that **r** is to be deleted from the arc matrix associated with **arc(1,2)** for role 1. In this case, support must also be removed for **r** on its local precede and follow arcs. The **1** precede arcs for **arc(1,2)** include only **arc(1,2)** and so no propagation is required to other arcs. On the other hand, the 1 follow arcs for **arc(1,2)** include **arc(1,3)** and **arc(1,4)**. When **arc(1,2)** is deleted from the **1** precede support list for **r** on **arc(1,3)**, its local precede list becomes, empty indicating that **r** should no longer receive support on the arc. The reason that **arc(1,3)** loses support for the role value is that the word node for role **2** must be a member of any path through the SLCN from the word node containing role **1** to the word node containing role 3. Because **r** on **arc(1,3)** has no remaining **1** precede supporter, it is deleted from **arc_matrix(1,3)**. Since **arc_matrix(1,3)** no longer has **1** precede support for **r**, it must remove its support from its local follow arc, **arc(1,4)**. When **arc(1,3)** is removed from the **1** precede arcs for **r** on **arc(1,4)**, there is one remaining supporter, **arc(1,2)**. However, **arc(1,2)** has not completed removing support for role value **r** from its remaining **1** local follow arc, **arc(1,4)**. When **arc(1,2)** is deleted from the **1** precede support list for **r** on **arc(1,4)**, its local precede list becomes empty indicating that **r** should no longer receive support on the arc. Because **arc(1,4)** has no remaining **1** precede supporter, the role value **r** is deleted for role **1** in **arc_matrix(1,4)**. Since arc-matrix(1,4) no longer has local precede support for **r**, it must also remove its support from its local follow arc, **arc(4,4)**, a dummy arc. Since the arc is a dummy, no loss of support is propagated to other arcs. Once the local support removal is complete, none of the arcs emanating from role **1** supports **r**, and so it is deleted from the role (because it's arc support count has become 0).

Now, suppose that **r** is to be deleted from the arc matrix associated with **arc(1,3)** for role **1**. In this case, the word node for role 3 provides an optional path through the SLCN from the word node containing role **1**, and so the role value **r** should not be deleted from **arc(1,2)** and **arc(1,4)**. The **1** precede arcs for **r** on **arc(1,3)** include only **arc(1,2)**. When **arc(1,3)** is removed from the **1** follow arcs for **r** on **arc(1,2)**, there is one remaining supporter, **arc(1,4)**, and so **r** should not be deleted from **arc_matrix(1,2)**. Furthermore, the **1** follow arcs for **r** on **arc(1,3)** include only **arc(1,4)**. When **arc(1,3)** is removed from the **1** precede arcs for **r** on **arc(1,4)**, there is one remaining supporter, **arc(1,2)**, and so **r** should not be deleted from **arc_matrix(1,2)**. Because **r** is supported on role **1** by **arc_matrix(1,2)** and **arc_matrix(1,4)**, it should not be deleted from the role.

The SLCN constraint parsing algorithm requires $O(n^4)$ time to parse a network with n word candidates. However, by using a CRCW P-RAM model and $O(n^4)$ processors, the parse time for an SLCN can be improved [15]. We are currently implementing the SLCN parsing algorithm on the MasPar MP-1. Because of the power of the MasPar and the parallel nature of the algorithm, we can make use of the flexibility and expressivity of CDG grammars to process SLCNs without

severe performance penalties.

4.4 An SLCN Parsing Experiment

We have developed a C++ implementation of a CDG parser, PARSEC, which is capable of parsing CNs or SLCNs on a Sun workstation. The software allows users to specify parameters of the grammar, to design features, and to write unary and binary constraints. It also checks the constraints for correctness of form given a grammar's parameters. Once a grammar is written and constraints are checked for well-formedness, the parser is ready to parse a sentence. When parsing a sentence, it first applies the core set of unary constraints, then it constructs arcs and arc matrices and propagates the core binary constraints. At this point, an X-windows interface for the parser appears on the screen. Figure 45 depicts the interface to the SLCN constructed from the word graph in Figure 33. Once the window is present, the user can choose from several menu options. Among the options available are: perform a single step of filtering, filter completely, view the arcs joining the roles of two word nodes, apply constraints from a file, print node information to a file, and print arc information to a file. The user can also view the state of the parse. This interface allows the remaining role values for each word candidate's roles to be viewed by clicking on the word in the word node For example, the role values for the three roles for the word windows over the interval (3,5) in Figure 45 are viewed by clicking on that word. This interface also allows viewing of the matrices stored on each of the arcs in the network. The arcs joining the roles of two words are displayed when a user selects the appropriate menu item along with two word nodes. Figure 46 shows the arcs joining the roles for the words of and the. The matrix associated with each of the arcs can then be viewed by clicking on its arc. Figure 47 shows the matrix for the arc joining the governor roles of the two words. This implementation of the SLCN parser provides useful tools for developing and testing constraint-based grammars.

In order to demonstrate the effectiveness of our SLCN parser, we have developed two grammars. The first grammar was designed to parse sets of sentences in the Resource Management database [36] The second grammar covered sentences in the ATIS database [16, 35] (Air Travel Information System).

The Resource Management database grammar contains 3 roles, 11 categories, 70 labels, 100 unary constraints, and 200 binary constraints, capable of parsing statements, yes-no questions, commands, and wh-questions [57]. The constraints consist of phrase structure rules and features tests. The feature tests include subject-verb agreement, determiner-head-noun agreement, and case restrictions on pronouns. Additionally, the subcategorization feature is used to make certain that a verb has the appropriate set of objects and complements to be complete. The lexicon used along with this grammar contains many lexically ambiguous words, and its word entries contain

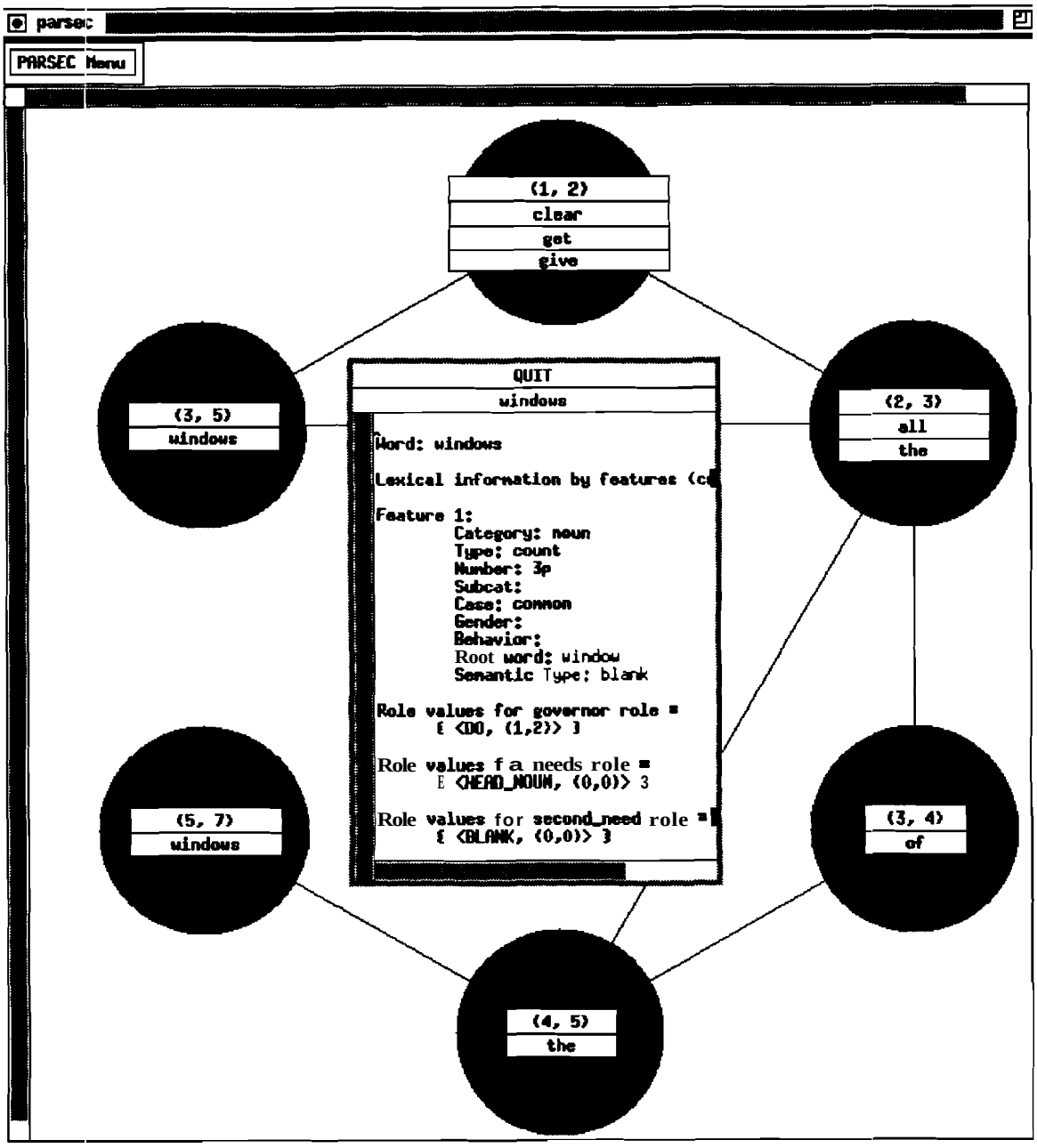


Figure 45: The X-windows interface to the SLCN for the N-best commands.

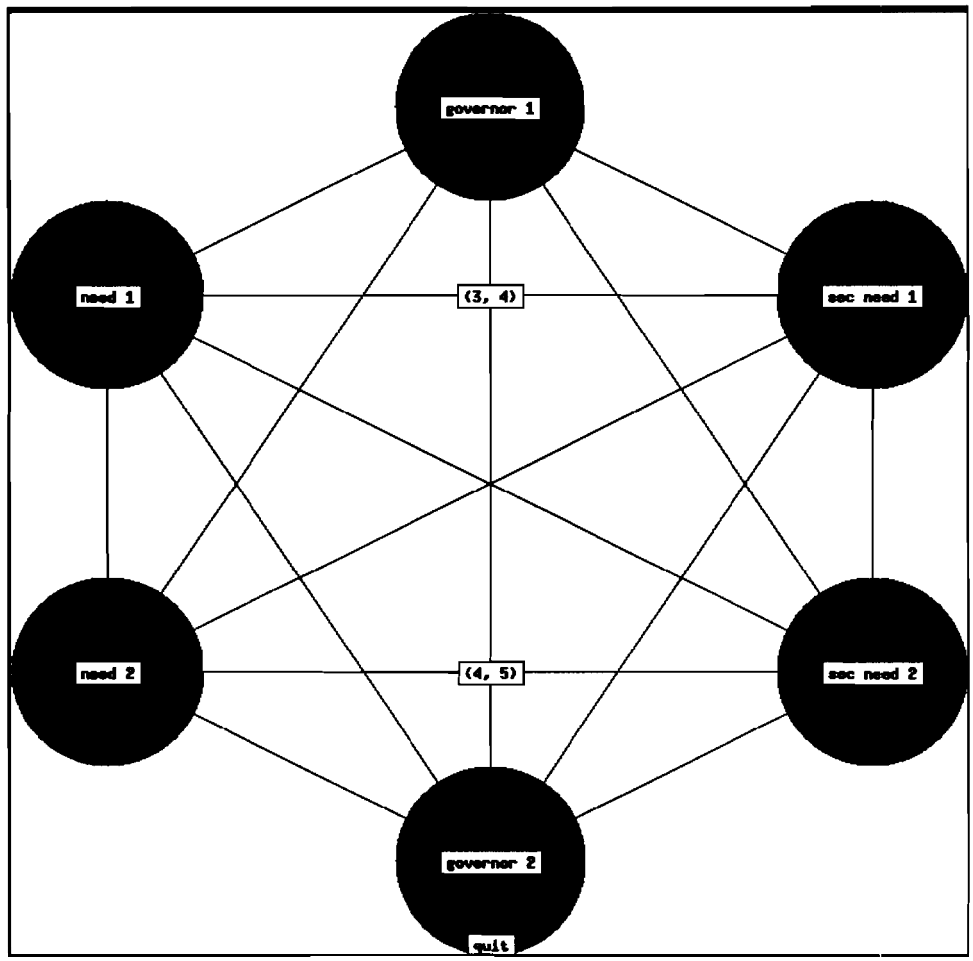


Figure 46: The X-window interface to the arcs connecting the roles of of and *the*

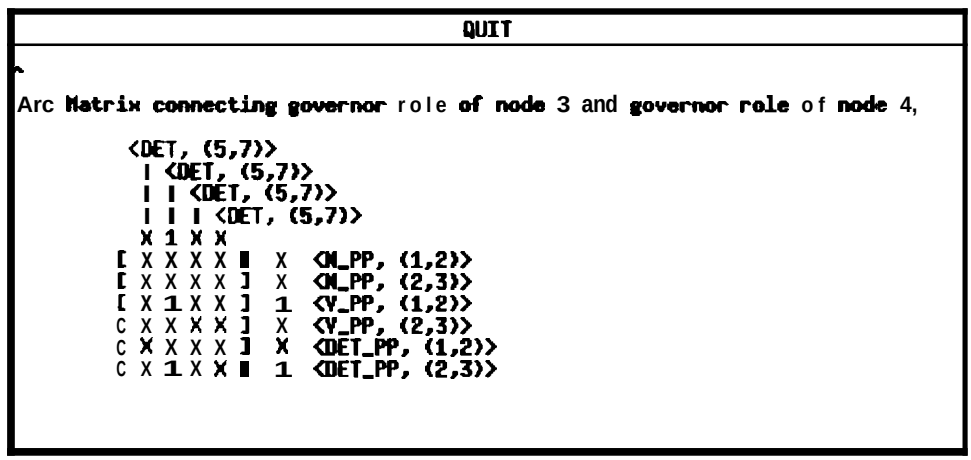


Figure 47: The X-window interface to the arc matrix for the governor roles of of and *the*.

Sentence Type	Number Grammatical Sentences in N-best	Number Grammatical Sentences in SLCN
Command	11	15
Yes-No-Q	8	12
Wh-Q	7	16

Figure 48: **N-best versus word graph sentence parses.**

the necessary feature information to support our feature value constraints.

To denionstrate the effectiveness of CDG parsing for eliminating sentence :hypotheses from a word graph, we converted the word graphs described in Section 4.1 into SLCNs and parsed them using the first grammar. More grammatical sentences were parsed in the SLCN than were available in the original sets of sentences, as can be seen in Figure 48; however, all of the additional parses had similar meanings to one of the original grammatical N-best sentences. For example, the SLCN depicted in Figure 45, constructed from the word graph in Figure 33, contains three verbs: clear, give, and *get*. Each is the main verb in 5 minor variations of the same sentence.

Syntactic constraints are effective at pruning a word graph of many ungrammatical sentence hypotheses and limiting the possible parses for the remaining sentences. However, it is often the case that syntactic information alone is insufficient for selecting a single sentence hypothesis from a word graph. Effective use of multiple knowledge sources plays a key role in human spoken language understanding. It is, therefore, likely that advances in spoken language understanding will require effective utilization of higher level knowledge.

To demonstrate the flexibility of constraint-based parsing for utilizing a variety of knowledge sources, we have incorporated semantic constraints into our parser. To do so, we developed a second grammar for sentences in the ATIS database (Air Travel Information System), which was chosen because of its semantic richness. First, syntactic constraints were added to the Resource Management grammar to demonstrate the ease of adding additional grammar constraints to a previously developed grammar. Then semantic constraints were constructed to further limit ambiguity [12]. Semantic constraints were relatively easy to create and incorporate into our parser because they were based on feature testing for certain syntactic configurations. For example, some constraints limited the semantic type associated with a prepositional phrase based on the semantic type of its object, and others limited the cites of attachment for a prepositional phrase based on semantic type compatibility. We then conducted a simple experiment to compare the effectiveness of syntactic and semantic constraints for reducing the ambiguity of word networks constructed from sets of BBN's N-best sentence hypotheses [41] from the ATIS database.

For this experiment, we selected twenty sets of 10-best sentence hypotheses for three different types of utterances: a command, a yes-no question, and a wh-question. The lists of the N-best sentences were converted to +word graphs using syllable count, as described in Section 4.1. These word graphs were then converted to SLCNs and parsed with the constraints. We determined for each eliminated word candidate whether a syntactic constraint or a semantic constraint was responsible for the deletion. Syntactic and semantic constraints together were very effective at reducing the number of parses for sentences in the SLCN when compared with syntactic constraints alone. However, syntactic constraints alone played the major role in pruning inappropriate word candidates from the network. On the average, syntactic constraints alone eliminated 3.11 word candidates per SLCN; whereas, semantic constraints, when applied after syntactic constraints, eliminated an average of .66 additional word candidates per SLCN[14].

5 Conclusion

SLCN parsing has several advantages that make it attractive for speech. First, it is able to handle grammars that are beyond context-free. Second, it provides a flexible uniform framework for using lexical, syntactic, semantic, prosodic, and contextual constraints to incrementally reduce the ambiguity found in a word graph provided by a speech recognition system (We have already developed lexical, syntactic, and semantic constraints for our parser (see Section 4.4) and are currently developing prosodic constraints). Third, the parser is able to support the use of context when determining the meaning of a sentence. Fourth, the flexibility of incremental constraint-based parsing should allow us to develop strategies for reducing sensitivity to the syntactic irregularities common in spontaneous speech. Current spoken language recognition systems are not as accurate as humans, in part, because they do not utilize the wide range of information that people do when understanding speech. Hence, we believe that further investigations along these lines will result in more effective processing of speech.

The filtering algorithm developed in this paper is useful, not only for processing speech, but also for other CSP problems. Up to this time, CSP arc consistency has always assumed perfect segmentation of input. Speech recognition is only one area where segmenting the signal into higher-level chunlrs is problematic. Vision systems and handwriting analysis systems have a comparable problem.

[14] Since most semantic rules use some syntactic information, it makes sense to propagate tht: syntactic constraints before the semantic constraints. When we propagate the semantic constraints first, no word candidates are typically eliminated because of the high ambiguity in the CN without syntactic constraints.

6 Acknowledgments

This work was supported in part by Purdue Research Foundation, NSF grant number IRI-9011179, and NSF Parallel Infrastructure Grant CDA-9015696. We thank BBN for providing us with the N-best lists of sentences. We would especially like to thank those students who were involved in the implementation of PARSEC: Yin Chan, Mark Rowland, Todd Stewart, Christopher White, Boon Lock Yeo, and Carla Zoltowski. We would also like to thank Carl Mitchell, Carla Zoltowski, and Leah Jamieson for their encouragement and comments on various drafts of this paper.

References

[1] J. M. Conrad and D. P. Agrawal. A graph partitioning-based load balancing strategy for a distributed memory machine. In *Proceedings* of the Sixth International *Conference* on *Parallel Processing*, August 1992.

[2] M. A. Covington. A parsing algorithm that extends phrases. Computational Linguistics, 4:234–236, 1990.

[3] A. L. Davis and A. Rosenfeld. Cooperating processes for low-level vision: A survey. Artificial *Intelligence*, 17:245–263, 1981.

[4] R. Dechter, I. Meiri, and J. Pearl. Temporal constraint networks. Artificial Intelligence, 34:1–38, 1988.

[5] R. Dechter and J. Pearl. Network-based heuristics for constraint-satisfaction problems. Artificial Intelligence, 34:1–38, 1988.

[6] P. Dey and B. R. Bryant. Lexical ambiguity in tree adjoining grammars. Information Processing Letters, 34:65–69, 1990.

[7] J. Early. An efficient context-free parsing algorithm. Communications of the ACM, 13:94–102, 1970.

[8] L. D. Erman and V. R. Lesser. The Hearsay-I1 speech understanding system: A tutorial. In W. A. Lea, editor, Trends in Speech Recognition, pages 361–381. Speech Science Publications, Apple Valley, MN, 1986.

[9] E. Freuder. Partial constraint satisfaction. In Proceedings of the *International* Joint Conference on Artificial Intelligence, pages 278–283, 1989.

[10] E. Freuder. Complexity of K-tree-structured constraint-satisfaction problems. In Proceedings of the Eighth National Conference on Artificial Intelligence, pages 4–9, 1990.

[11] E. P. Giachin. Automatic training of stochastic finite-state language models; for speech understanding. In IEEE Int. Conf. Acoustics, Speech, and Signal Processing, March 1992.

[12] M. P. Harper, L. H. Jamieson, C. B. Zoltowski, and R. A. Helzerman. Semantics and constraint parsing of word graphs. In Proceedings of the International Conference on Acoustics, Speech, and Signal Processing, volume II, pages 63–66, April 1992.

[13] R. A. Helzerman. PARSEC: A framework for parallel natural language understanding. Master's thesis,, Purdue University, School of Electrical Engineering, West Lafayette, IN, 1993.

[14] R. A. Helzerman and M. P. Harper. Log time parsing on the MasPar MP-1. In Proceedings of the Sixth International Conference on Parallel Processing, August 1992.

[15] R. A. Helzerman, M. P. Harper, and C. B. Zoltowski. Parallel parsing of spoken language. In Proceedings of the Fourth Symposium on the Frontiers of Massively Parallel Computation, October 1992.

[16] C. T. Hemphill, J. J. Godfrey, and G. R. Doddington. The ATIS spoken language systems pilot corpus. Technical Report NTIS PB91-505354, 1990. NIST Speech Disc 5-1.1.

[17] F. Jelinek. Self-organized language modeling for speech recognition. In Alex 'Waibel and Kai-Fu Lee, editors, Readings in Speech Recognition. Morgan Kaufman Publishers, Inc., San Mateo, CA, 1990.

[18] A. K. Joshi, L.S. Levy, and M. Takahashi. Tree adjunct grammars. Journal of Computer and System Sciences, 10:136–163, 1975.

[19] M. Kay. The MIND system. In R. Rustin, editor, Natuml Language Processing. Algorithmics Press, New York, 1973.

[20] S. R. Kosaraju. Speed of recognition of context-free languages by array automata. SIAM Journal of Computing, 4(3) :331–340, September 1975.

[21] V. Kumar. Algorithms for constraint-satisfaction problems: A survey. AI Magazine, 13(1):32–44, 1992.

[22] K. F. Lee, H. W. Hon, and R. Reddy. An overview of the SPHINX speech recognition system. In IEEE Tmnsactions on Acoustic, Speech, Signal Processing, pages 35–45, January 1990.

[23] V. R. Lesser, R. D. Fennell, L. D. Erman, and D.R. Reddy. Organization of the Hearsay-I1 speech understanding system. IEEE Tmns. Acoust., Speech, Signal Processing, ASSP-23:ll–23, 1975.

[24] A. K. Mackworth. Consistency in networks of relations. Artificial Intelligence, 8(1):99–118, 1977.

[25] A. K. Mackworth and E. Freuder. The complexity of some polynomial network-consistency algorithms for constraint-satisfaction problems. Artificial Intelligence, 25:65–74, 1985.

[26] H. Maruyama. Constraint dependency grammar. Technical Report #RT0044, IBM, Tokyo, Japan, 1990.

[27] H. Maruyama. Constraint dependency grammar and its weak generative capacity. Computer *Software*, 1990.

[28] H. Maruyama. Structural disambiguation with constraint propagation. In 'The Proceedings of the *Annual* Meeting of ACL, 1990.

[29] R. Mohr and T. Henderson. Arc and path consistency revisited. Artificial Intelligence, 28, 1986.

[30] U. Montanari. Networks of constraints: Fundamental properties and applications to picture processing. Information Science, 1976.

[31] M. D. Moshier and W. C. Rounds. On the succinctness properties of unordered context-free grammars. In Proceedings of the 25th Annual Meeting of the Association *for* Computational L*inguistics*, 1987.

[32] M. A. Palis, S. Shende, and D. S. L. Wei. An optimal linear-time parallel parser for tree adjoining languages. SIAM Journal of Computing, 19:1–31, 1990.

[33] D. B. Paul. An efficient A* stack decoder algorithm for continuous speech recognition with a stochastic language model. In IEEE Int. Conf. Acoustics, Speech, and Signal Processing, March 1992.

[34] F. C. N. Pereira and D. H. D. Warren. Definite clause grammars for language analysis– A survey of the formalism and a comparison with augmented transition networks. Artificial Intelligence, 13(3):231–278, 1980.

[35] P. J. 1[3]rice. Evaluation of spoken language systems: The ATIS domain. In Proceedings of the *DARPA* Workshop on Speech and Natural Language, pages 91–95, 1990.

[36] P. J. Price, W. Fischer, J. Bernstein, and D. Pallett. A database for continuous speech recognition in a 1000-word domain. In Proceedings of the International *Conference* on Acoustics, *Speech*, and Signal Processing, 1988.

[37] W. Ruzzo. Tree-size bounded alternation. Journal of Computers and *System* Sciences, 21:218–235, 1980.

[38] Y. Schabes. Polynomial time and space shift-reduce parsing of arbitrary context-free grammars. In *The* Proceedings of the Annual Meeting of ACL, 1991.

[39] R. Schwartz and S. Austin. A comparison of several approximate algorithms for finding multiple N-best sentence hypotheses. In IEEE Int. Conf. Acoustics, Speech, and *Signal* Processing, May 1991.

[40] R. Schwartz, Y. Chow, O. Kimball, S. Roucos, M. Krasner, and J. Miskhoul. Context-dependent modeling for acoustic-phonetic recognition of continuous speech. In IEEE Int. Conf. Acoustics, Speech, and Signal Processing, March 1985.

[41] R. Schwartz and Y-L. Chow. The N-best algorithm: An efficient and exact procedure for finding; the N most likely sentence hypotheses. In IEEE Int. Conf. Acoustics, Speech, and Signal Processing, April 1990.

[42] S. Seneff. Robust parsing for spoken language systems. In IEEE Int. Conf. Acoustics, Speech, and *Signal* Processing, March 1992.

[43] S. Seneff. TINA: A natural language system for spoken language applications. American *Journal* of Computational Linguistics, 18:61–86, 1992.

[44] J. Seo and R. F. Simmons. Syntactic graphs: A representation for the union of all ambiguous parse trees. Computational Linguistics, 15:19–32, 1989.

[45] S. M. Shieber. Constraint-based Gmmmar Formalisms. MIT Press, Cambridge, MA, 1992.

[46] M. Tomita. *Efficient* Parsing for Natuml Language. Kluwer Academic Publishers, Boston, MA, 1985.

[47] M. Tomita. An efficient word lattice parsing algorithm for continuous speech recognition. In IEEE Int. Conf. Acoustics, Speech, and Signal Processing, April 1986.

[48] K. Vijayashanker and A. K. Joshi. Some computational properties of tree adjoining grammars. In Proceedings of the *24th* Annual Meeting of the Association for Computational Linguistics, 1986.

[49] K. Vijayshanker, D.J. Weir, and A. K. Joshi. Characterizing structural descriptions produced by various grammatical formalisms. In Proceedings of the 25th Annual *Meeting* of the Association *for* Computational Linguistics, 1987.

[50] M. Villain and H. Kautz. Constraint-propagation algorithms for temporal reasoning. In Proceedings of the Fifth National Conference on Artificial Intelligence, pages 3'77–382, 1986.

[51] D. L. Waltz. Understanding line drawings of scenes with shadows. In P.H. Winston, editor, The Psychology of Computer Vision. McGraw Hill, New York, 1975.

[52] W. Ward. Understanding spontaneous speech: The Phoenix system. In IEEE Int. Conf. *Acoustics,* Speech, and Signal Processing, May 1991.

[53] J. J. Wolf and W. A. Woods. The HWIM speech understanding system. In W. A. Lea, editor, *Trends* in Speech Recognition, pages 316–339. Speech Science Publications, .Apple Valley, MN, 1986.

[54] W. A. Woods. Transition network grammars for natural language analysis. Communications of the ACM, 13:591–606, 1970.

[55] W. A. Woods, M. Bates, G. Brown, B Bruce, C. Cook, J. Klovstad, J Makhoul, B. Nash-Webber, R. Schwartz, J. Wolf, and V. Zue. Speech understanding systems: Final technical progress report. Technical Report 3438, Bolt, Beranek, and Newman, Inc., Cambridge, MA, 1976.

[56] J.H. Wright. LR parsing of probabilistic grammars with input uncertainty for speech recognition. *Computer* Speech and Language, 4:298–323, 1990.

[57] C. B. Zoltowski, M. P. Harper, L. H. Jamieson, and R. A. Helzerman. PARSEC: A constraint-based framework for spoken language understanding. In Proceedings of the *International* Conference on Spoken Language Understanding, October 1992.

[58] V. Zue, J. Glass, D. Goodine, H. Leung, M. Phillips, J. Polifroni, and S. Seneff. Integration of speech recognition and natural language processing in the MIT Voyager system. In IEEE Int. *Conf.* Acoustics, Speech, and Signal Processing, May 1991.

A Appendix: CDG Parsing Pseudocode for CNs and SLCNs

The pseudocode for each of the steps of the CN and the SLCN parsing algorithms is listed in this appendix.

Initial Network Construction for CNs and SLCNs– $O(n^2)$ operation.

```
{
   For index = 1 to n do {                                   /* n iterations */
      w = get_word_node(index);
      For r = w's roles 1 to p do {                          /* p iterations */
         For l = each of the labels in L do {                /* q iterations */
            add role value (l,nil) to the role r for w;
            For node = 1 to n do {                           /* n iterations */
               pos = get_tuple(node);
               add role value (l,pos) the role r for w))))
```

Initial Network Construction with TABLE– $O(n^2)$ operation

```
I
   For index = 1 to n do {                                   /* n iterations */
      w = get_word_node(index);
      parts = get_parts_of_speech(w);
      For part = each part of speech in parts do {           /* O(Max_parts) iterations */
         For r = each of w's roles 1 to p do {               /* p iterations */
            ll = get_labels(part,r);
            For l = the labels in ll do {                    /* O(Max_labels/parts) iterations */
               add role value (l,nil) to the role r with p part of speech;
               For node = 1 to indez – 1 and indez + 1 to n do {   /* n iterations */
                  pos = get_tuple(node);
                  add role value (l,pos) the role r)))))
```

Unary Constraint Propagation on a CN or an SLCN– an $O(k_u * n^2)$ operation.

```
I
   For index = 1 to n do {                                   /* n iterations */
      w = get_word_node(index);
      For r = w's roles 1 to p do {                          /* p iterations */
         rv = get_role_values(r);
         For r-index = each of the role values in rv do {    /* O(n) iterations */
            For c = each unary constraint do {               /* k_u iterations */
               apply get_constraint(c) to the role value r-index))))
```

CN Arc Construction– an $O(n^4)$ operation.

```
{
   /* Create arc matrices for arcs joining roles within the word– a (p 2) * O(n^3) operation. */
   For index = 1 to n do {                                   /* n iterations */
```

```
    w = get_word_node(index);
    For r1 = w's roles 1 to p – 1do {
        rv1 = get_role_values(r1);
        For r2 = w's roles (r1 + 1) to p do {                    /* (;)iterations */
            rv2 = get_role_values(r2);
            create arc joining r1 to r2;
            assign arc a matrix of size |rv1| x |rv2| with entries    /* O(n^2) operation*/
                initialized to 1 and indices from rv1 and rv2}}};
/* Create arc matrices for arcs joining roles in different words– an (n*p 2) * O(n^2) operation. */
For index1 = 1 to n – 1 do {
    w1 = get_word_node(index1);
    For r1 = w1's roles 1 to p do {
        rv1 = get_role_values(r1)
        For index2 = index1+1 to n do {                          /* (n*p 2) iterations */
            w2 = get-wordnode(index2);
            For r2 = w2's roles 1 to p do {
                rv2 = get_role_values(r2);
                create arc joining r1 to r2;
                assign arc a matrix of size |rv1| × |rv2| with entries    /* O(n^2) operation*/
                    initialized to 1 and indices from rv1 and rv2}}}}
```

SLCN Arc: **Construction– an** $O(n^4)$ **operation.**

```
/* Create arc matrices for arcs joining roles within the word– a (p 2) * O(n^3) operation. */
For index = 1 to n do {                                          /* n iterations */
    w = get_word_node(index);
    For r1 = w's roles 1 to p – 1 do {
        rv1 = get_role_values(r1);
        For r2 = w's roles (r1 + 1) to p do {                    /* (p 2) iterations */
            rv2 = get_role_values(r2);
            create arc joining r1 to r2;
            assign arc a matrix of size |rv1| x |rv2| with entries    /* O(n^2) operation*/
                initialized to 1 and indices from rv1 and rv2}}};
/* Create arc matrices for arcs joining roles in different words that can be in */
/* at least one common sentence hypothesis – an (n*p 2) * O(n^2) operation. */
sortedmodes = nodes sorted in increasing order by beginning point;
For w1 = each word node in sortedmodes do {
    For r1 = w1's roles 1 to p do {
        rv1 = get_role_values(r1);
        /* Get the list of word nodes that have a directed edge to w1 */
        prec_nodes = get_preceding_nodes(w1);
        connected-roles = O;
        For w2 = each word node in prec_nodes do {
            For r2 = w2's roles 1 to p do {
                connected-roles = connected-roles ∪ {r2} ∪ get-arc-connected-roles(r2:);
                /* get-arc-connected-roles returns the list of roles r2 is arc-connected to */
                }}
        For r2 = all roles in connected-roles do {
            create arc joining r1 to r2;
            rv2 = get_role_values(r2);
            assign arc a matrix of size |rv1| × |rv2| with
                entries initialized to 1 and indices from rv1 and rv2}}}
create_matrix_support();      /* See the algorithm in Section 4.3 */
```

Binary Constraint Propagation on a CN or an SLCN– an $O(k_b * n^4)$ operation.

```
{
  /* Propagate constraints over elements of intra-arc matrices – a (p 2) * k_b * O(n^3) operation. */
  For index = 1 to n do {                                          /* n iterations */
    w = get_word_node(index);
    For r1 = w's roles 1 to p – 1 do {
      rv1 = get_role_values(r1);
      For r2 = w's roles (r1 + 1) to p do {                        /* (p 2) iterations */
        rv2 = get_role_values(r2);
        For r-index1 = each of the role values in rv1 do {         /* O(n) iterations */
          For r-index2 = each of the role values in rv2 do {       /* O(n) iterations */
            For c = 1 to k_b do {                                  /* k_b iterations */
              apply get_constraint(c) to the role values           /* O(1) operation */
                X = r-index1 and Y = r-index2;
              apply get_constraint(c) to the role values           /* O(1) operation */
                Y = r-index1 and X = r-index2}}}}};
  /* Propagate constraints over elements of the other matrices – a (n*p 2) * k_b * O(n^2) operation. */
  For index1 = 1 to n – 1 do {
    w1 = get_word_node(index1);
    Forr1 = w1's roles 1 to p do {
      rv1 = get_role_values(r1);
      For index2 = index1+1 to n do {
        w2 = get-wordmode(index2);
        For r2 = w2's roles 1 to p do {                            /* (n*p 2) iterations */
          rv2 = get_role_values(r2);
          For r-index1 = each of the role values in rv1 do {       /* O(n) iterations */
            For r-index2 = each of the role values in rv2 do {     /* O(n) iterations */
              For c = 1 to k_b do {                                /* k_b iterations */
                apply get_constraint(c) to the role values         /* O(1) operation */
                  X = r-index1 and Y = r-index2;
                apply get_constraint(c) to the role values         /* O(1) operation */
                  Y = r-index1 and X = r-index2}}}}}}
```

CN Filtering– an $O(n^4)$ operation.

```
{
  /* Preprocess– an O(n^3) operation */
  /* Get role values to be deleted from arc matrices for arcs      */
  /* joining roles within a word– a (p 2) * O(n^2) operation.      */
  For index = 1 to n do {                                          /* n iterations */
    w = get_word_node(index);
    For r1 = w's roles 1 to p – 1 do {
      rv1 = get_role_values(r1);
      For r2 = w's roles (r1 + 1) to p do {                        /* (p 2) iterations */
        rv2 = get_role_values(r2);
        For r-index1 = each of the role values in rv1 {            /* O(n) iterations */
          if (the role value r-index1 for role r1 is not           /* O(1) operation */
              supported in the matrix corresponding to
```

```
                    the arc of role r1 and role r2) {
                        add (r-indexl, r1) to the to-be deleted list));
                For r-index2 = each of the role values in rv2 {          /* O(n) iterations */
                    if (the role value r-index2 for role r2 is not      /* O(1) operation */
                        supported in the matrix corresponding to
                        the arc of role r1 and role r2) {
                            add (r-index2, r2) to the to-be deleted list)))))
    /* Get role values to be deleted from arc matrices for arcs        */
    /* joining roles across words– an (n*p 2) * O(n) operation.        */
    For index1 = 1 to n – 1 do {
        w1 = get_word_node(index1);
        For r1 = w1's roles 1 to p do {
            rv1 = get_role_values(r1);
            For index2 = index1+1 to n do {                             /* (n*p 2) iterations */
                w2 = get-wordnode(index2);
                For r2 = w2's roles 1 to p do {
                    rv2 = get_role_values(r2)
                    For r-index1 = each of the role values in rv1 {     /* O(n) iterations */
                        if (the role value r-index1 for role r1 is not  /* O(1) operation */
                            supported in the matrix corresponding to
                            the arc of role r1 and role r2) {
                                add (r-indexl, r1)to the to-be deleted list));
                    For r-index2 = each of the role values in rv2 {     /* O(n) iterations */
                        if (the role value r-index2 for role r2 is not  /* O(1) operation */
                            supported in the matrix corresponding to
                            the arc of role r1 and role r2) {
                                add (r-index2, r2) to the to-be deleted list))))))
    /* Loop until there are no more to be deleted – an O(n^4) operation */
    Loop until to-be-deleted-list is empty {                            /* O(n^2) iterations */
        pair = pop(to-be-deleted-list);
        rv1 = the role value from the pair;
        r1 = the role from the pair;
        remove rv1 from role r1;
        For r2 = all roles not equal to r1 do {                         /* O(n) iterations */
            remove the role value rv1 from the arc_matrix(r1,r2);       /* O(n) operation */
            rv2 = get_role_values(r2);
            For r-index2 = each of the role values in rv2 {             /* O(n) iterations */
                if (the role value r-index2 for role r2 is not          /* O(1) operation */
                    supported in the matrix corresponding to
                    the arc of role r1 and role r2) {
                        add r-index2 to the to-be-deleted-list))))
}
```

SLCN Filtering– an $O(n^4)$ operation.

```
{
    /* Preprocess– an O(n^3) operation */
    /* Get role values to be deleted from arc matrices for arcs joining */
    /* roles within a word– a (p 2) * O(n^2) operation.                 */
    For index = 1 to n do {                                             /* O(n) iterations */
        w = get_word_node(index);
        For11= w'sroles 1 top– 1do {
            rv1 = get_role_values(r1);
            For r2 = w's roles (r1 + 1) to p do {                       /* (p 2) iterations */
                rv2 = get_role_values(r2);
```

```
            For r-index1 = each of the role values in rvl {                  /* O(n) iterations */
               if  (the role value r-index1 for role r1 is not               /* O(1) operation */
                    supported in the matrix corresponding to
                    the arc of role r1 and role r2) {
                       add (r-indexl, r1, r2) to the to-be deleted list));
            For r-index2 = each of the role values in rv2 {                  /* O(n) iterations */
               if  (the role value r-index2 for role r2 is not               /* O(1) operation */
                    supported in the matrix corresponding to
                    the arc of role r1 and role r2) {
                       add (r-index2, r2, r1) to the to-be deleted list)))))
   /* Get role values to be deleted from arc matrices for arcs joining      */
   /* roles across words– an (n*p 2) * O(n) operation.                      */
   For index1 = 1 to n – 1 do {
      w1 = get_word_node(index1);
      For r1 = w1's roles 1 to p do {
         rv1 = get_role_values(r1);
         For index2 = index1+1 to n do {                                     /* (n*p 2) iterations */
            w2 = get-wordmode(index2);
            For r2 = w2's roles 1 to p do {
               rv2 = get_role_values(r2)
               For r-index1 = each of the role values in rv1 {               /* O(n) iterations */
                  if  (the role value r-index1 for role r1 is not            /* O(1) operation */
                       supported in the matrix corresponding to
                       the arc of role r1 and role r2) {
                          add (r-indexl, r1, r2) to the to-be deleted list));
               For r-index2 = each of the role values in rv2 {               /* O(n) iterations */
                  if  (the role value r-index2 for role r2 is not            /* O(1) operation */
                       supported in the matrix corresponding to
                       the arc of role r1 and role r2) {
                          add (r-index2, r2, r1) to the to-be deleted list))))))
   /* Loop until there are no more to be deleted – an O(n^4) operation */
   Loop until to-be-deleted-list is empty {
      item = pop(to-be-deleted-list);
      r = the role value from item to delete;
      rA = the role from item to delete r from;
      rB = the role from item joined to rA with the arc we are deleting from;
      /* Each role value can only remove support once from its precede and next arcs */
      if (the precede and follow lists for r in rA for arc_matrix(rA, rB) are non-empty) then
         disallow (arc(rA, rB),rA, r);      /* See the algorithm in Section 4.3 */
   }
}
```