Syntactic Graphs: A Representation for the Union of All Ambiguous Parse Trees

Jungyun Seo
Robert F. Simmons

Artificial Intelligence Laboratory
University of Texas at Austin
Austin, TX 78712-1188

In this paper, we present a new method of representing the surface syntactic structure of a sentence. Trees have usually been used in linguistics and natural language processing to represent syntactic structures of a sentence. A tree structure shows only one possible syntactic parse of a sentence, but in order to choose a correct parse, we need to examine all possible tree structures one by one. Syntactic graph representation makes it possible to represent all possible surface syntactic relations in one directed graph (DG). Since a syntactic graph is expressed in terms of a set of triples, higher level semantic processes can access any part of the graph directly without navigating the whole structure. Furthermore, since a syntactic graph represents the union of all possible syntactic readings of a sentence, it is fairly easy to focus on the syntactically ambiguous points. In this paper, we introduce the basic idea of syntactic graph representation and discuss its various properties. We claim that a syntactic graph carries complete syntactic information provided by a parse forest—the set of all possible parse trees.

1 Introduction

In natural language processing, we use several rules and various items of knowledge to understand a sentence. Syntactic processing, which analyzes the syntactic relations among constituents, is widely used to determine the surface structure of a sentence, because it is effective to show the functional relations between constituents and is based on well-developed linguistic theory. Tree structures, called **parse trees**, represent syntactic structures of sentences.

In a natural language understanding system in which syntactic and semantic processes are separated, the semantic processor usually takes the surface syntactic structure of a sentence from the syntactic analyzer as input and processes it for further understanding.[1] Since there are many ambiguities in natural language parsing, syntactic processing usually generates more than one parse tree. Therefore, the higher level semantic processor should examine the parse trees one by one to choose a correct one.[2] Since possible parse trees of sentences in ordinary expository text often number in the hundreds, it is impractical to check parse trees one by one without knowing where the ambiguous points are. We have tried to reduce this problem by introducing a new structure, the **syntactic graph**, that can represent all possible parse trees effectively in a compact form for further processing. As we will show in the rest of this paper, since all syntactically ambiguous points are kept in a syntactic graph, we can easily focus on those points for further disambiguation.

Furthermore, syntactic graph representation can be naturally implemented in efficient, parallel, all-path parsers. One-path parsing algorithms, like the DCG (Pereira and Warren 1980), which enumerates all possible parse trees one by one with backtracking, usually have exponential complexity. All-path parsing algorithms explore all possible paths in parallel without backtracking (Early 1970; Kay 1980; Chester 1980; Tomita 1985). In these algorithms, it is efficient to generate all possible parse trees. This kind of algorithm has complexity $O(N^3)$ (Aho and Ullman 1972; Tomita 1985).

We use an all-path parsing algorithm to parse a sentence. Triples, each of which consists of two nodes and an arc name, are generated while parsing a sentence. The parser collects all correct triples and constructs an **exclusion matrix**, which shows co-occurrence constraints among arcs, by navigating all possible parse

0362-613X/ 89 /010019-32$03.00

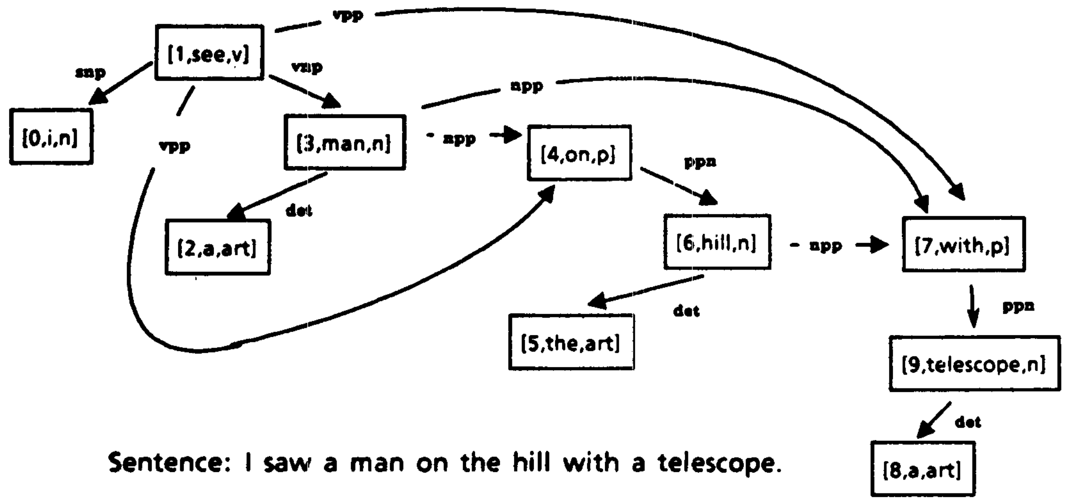

Figure 1: Syntactic Graph of the Example Sentence.

trees in a **shared, packed-parse forest.**[3] We claim that a syntactic graph represented by the triples and an exclusion matrix contains all important syntactic information in the parse forest.

In the next section, we motivate this work with an example. Then we briefly introduce $\overline{X}$ (X-bar) theory with head projection, which provides the basis of the graph representation, and the notation of graph representation in Section 3. The properties of a syntactic graph are detailed in Section 4. In Section 5, we introduce the idea of an exclusion matrix to limit possible tree interpretations of a graph representation. In Section 6, we will present the definition of completeness and soundness of the syntactic graph representation compared to parse trees by showing an algorithm that enumerates all syntactic readings using the exclusion matrix from a syntactic graph. *We claim that those readings include all the possible syntactic readings of the corresponding parse forest.* Finally, after discussing related work, we will suggest future research and draw some conclusions.

2 Motivational Example

We are currently investigating a model of natural language text understanding in which syntactic and semantic processors are separated.[4] Ordinarily, in this model, a syntactic processor constructs a surface syntactic structure of an input sentence, and then a higher level semantic processor processes it to understand the sentence—i.e., syntactic and semantic processors are pipelined. If the semantic processor fails to understand the sentence with a given parse tree, the semantic processor should ask the syntactic processor for another possible parse tree. This cycle of processing will continue until the semantic processor finds the correct parse tree with which it succeeds in understanding the sentence.

Let us consider the following sentences, from Waltz (1982):

I saw a man on the hill with a telescope.
I cleaned the lens to get a better view.

When we read the first sentence, we cannot determine whether the man has a telescope or the telescope is used to see the man. This is known as the PP-attachment problem, and many researchers have proposed various ways to solve it (Frazier and Fodor 1979; Shubert 1984, 1986; Wilks et. al 1985). In this sentence, however, it is impossible to choose a correct syntactic reading in syntactic processing—even with commonsense knowledge. The ambiguities must remain until the system extracts more contextual knowledge from other input sentences.

The problems of tree structure representation in the pipelined, natural language processing model are the following:

- First, since the number of parse trees of a typical sentence in real text easily grows to several hundreds, and it is impossible to resolve syntactic ambiguities by the syntactic processor itself, a semantic processor must check all possible parse trees one by one until it is satisfied by some parse tree.[5]
- Second, since there is no information about where the ambiguous points are in a parse tree, the semantic processor should check all possibilities before accepting the parse tree.
- Third, although the semantic processor might be satisfied with a parse tree, the system should keep the status of the syntactic processor for a while, because there is a fair chance that the parse tree may become unsatisfactory after the system processes several more sentences. For example, attaching the prepositional phrase (PP) "with a telescope" to "hill" or "man" would be fine for the semantic processor, since there is nothing semantically wrong with these attachments. However, these attachments become unsatisfactory after the system understands the next

sentence. Then, the semantic processor would have to backtrack and request from the syntactic processor another possible parse tree for the earlier sentence.

We propose the syntactic graph as the output structure of a syntactic processor. The syntactic graph of the first sentence in the previous example is shown in Figure 1. In this graph, nodes consist of the positions, the root forms, and the categories of words in the sentence. Each node represents a constituent whose head word is the word in the node. Each arc shows a dominator-modifier relationship between two nodes. The name of each arc is uniquely determined according to the grammar rule used to generate the arc. For example, the snp arc is generated from the grammar rule, SNT → NP VP, vpp is from the rule, VP → VP PP, and ppn from the rule, PP → Prep NP, etc.

As we can see in Figure 1, all syntactic readings are represented in a directed graph in which every ambiguity—lexical ambiguities from words with multiple syntactic categories and structural ambiguities from the ambiguous grammar—is kept. The nodes which are pointed to by more than one arc show the ambiguous points in the sentence, so the semantic processor can focus on those points to resolve the ambiguities. Furthermore, since a syntactic graph is represented by a set of triples, a semantic processor can directly access any part of a graph without traversing the whole. Finally, syntactic graph representation is compact enough to be kept in memory for a while.[6]

3 $\overline{X}$ Theory and Syntactic Graphs

$\overline{X}$ theory was proposed by Chomsky (1970) to explain various linguistic structural properties, and has been widely accepted in linguistic theories. In this notation, the head of any phrase is termed X, the phrasal category containing X is termed $\overline{X}$, and the phrasal category containing $\overline{X}$ is termed $\overline{\overline{X}}$. For example, the head of a noun phrase is N (noun), $\overline{N}$ is an intermediate category, and $\overline{\overline{N}}$ corresponds to noun phrase (NP). The general form of the phrase structure rules for $\overline{X}$ theory is roughly as follows:

- $\overline{\overline{X}} \rightarrow \overline{\overline{Y}} * \overline{X}$
- $\overline{X} \rightarrow X\overline{\overline{Z}}\ *$, where $*$ is a Kleene star.

$\overline{\overline{Y}}$ is the phrase that specifies $\overline{X}$, and $\overline{\overline{Z}}$ is the phrase that modifies X.[7] The properties of the head word of a phrase are projected onto the properties of the phrase. We can express a grammar with $\overline{X}$ conventions to cover a wide range of English.

Since, in $\overline{X}$ theory, a syntactic phrase consists of the head of the phrase and the specifiers and modifiers of the head, if there are more than two constituents in the right-hand side of a grammar rule, then there are dominator-modifier (DM) relationships between the head word and the specifier or modifier words in the phrase. Tsukada (1987) discovered that the DM relationship is effective for keeping all the syntactic ambiguities in a compact and handy structure without enumerating all possible syntactic parse trees. His representation, however, is too simple to maintain some important information about syntactic structure that will be discussed in detail in this paper, and hence fails to take full advantage of the DM-relationship representation.

We use a slightly different representation to maintain more information in head-modifier relations. Each head-modifier relation is kept in a **triple** that is equivalent to an **arc** between two **nodes** (i.e., words) in a syntactic graph. The first element of a triple is the arc name, which represents the relation between the head and modifier nodes. The second element is the lexical information of the head node, and the third element is that of the modifier node. The direction of an arc is always from a head to a modifier node. For example, the triple `[snp, [1,see,v], [0,i,n]]` represents the arc `snp` between the two nodes `[1,see,v]` and `[0,i,n]` in Figure 1.

Since many words have more than one lexical entry, we have to keep the lexical information of each word in a triple so that we can distinguish different usages of a word in higher level processing. The triples corresponding to some common grammar rules are as follows:

1. $\overline{\overline{N}} \rightarrow \underline{Det}\ \overline{N} \Leftrightarrow [det,[[n_1,R_1]|L_1],[[n_2,R_2]|L_2]]$
2. $\overline{N} \rightarrow \underline{\overline{\overline{Adj}}}\ \overline{N} \Leftrightarrow [mod,[[n_3,R_3]|L_3],[[n_4,R_4]|l_4]]$
3. $\overline{N} \rightarrow N\ \overline{\overline{\mathrm{Prep}}} \Leftrightarrow [npp,[[n_5,R_5]|L_5],[[n_6,R_6|L_6]]$

Each n_i represents the position, each R_i represents the root form, and each L_i represents a list of the lexical information including the syntactic category of each word in a sentence. Parentheses signify optionality and the asterisk (*) allows repetition.

Figure 2 shows the set of triples representing the syntactic graph in Figure 1 and the grammar rules used to parse the sentence. The sentence in Figure 2 has five possible parse trees in accordance with the grammar rules. All of the dependency information in those five parses is represented in the 12 triples. Those 12 triples represent all possible syntactic readings of the sentence with the grammar rules. Not all triples can co-occur in one syntactic reading in the case of an ambiguous sentence.

The pointers of each triple are the list of the indices that are used as the pointers pointing to that triple. For example, Triple 2 in Figure 2 has a list of three indices as the pointers. Each of those indices can be used as a pointer to access the triple. These indices are actually used as the names of the triple. One triple may have more than one index. The issues of why and how to produce indices of triples will be discussed later in this section.

Triple 3 in Figure 2 represents the vnp arc in Figure 1 between two nodes, `[1,see,v]` and `[3,man,n]`. The node `[1,see,v]` represents a VP with head word

GRAMMAR RULES AND CORRESPONDING TRIPLES:

Grammar rules		arc-name	head	modifier
1. SNT → NP	VP	snp	head of VP	head of NP
2. NP → art	NP	det	head of NP	art
3. NP → N'			head of N'	
4. N' → N'	PP	npp	head of N'	head of PP
5. N' → noun			noun	
6. PP → prep	NP	ppn	prep	head of NP
7. VP → V'			head of V'	
8. V' → V'	NP	vnp	head of V'	head of NP
9. V' → V'	PP	vpp	head of V'	head of PP
10. V' → verb			verb	

SENTENCE: I SAW A MAN ON THE HILL WITH A TELESCOPE

	Triples for the Input Sentence	Pointers
1.	[snp, [[1,see],categ,verb,tns,past]; [[0,i],categ,noun, nbr,sing]]	[22]
2.	[det, [[3,man],categ,noun,nbr,sing], [[2,a],categ,art,ty,indef]]	[02, 09, 20]
3.	[vnp, [[1,see],categ,verb,tns,past], [[3,man],categ,noun,nbr,sing]]	[03, 10, 21]
4.	[vpp, [[1,see],categ,verb,tns,past], [[4,on],categ,prep]]	[13, 24]
5.	[npp, [[3,man],categ,noun,nbr,sing], [[4,on],categ,prep]]	[08, 19]
6.	[det, [[6,hill],categ,noun,nbr,sing], [[5,the],categ,art,ty,def]]	[06, 17]
7.	[ppn, [[4,on],categ,prep], [[6,hill],categ,noun,nbr,sing]]	[07, 18]
8.	[vpp, [[1,see],categ,verb,tns,past], [[7,with],categ,prep]]	[26]
9.	[npp, [[6,hill],categ,noun,nbr,sing], [[7,with],categ,prep]]	[16]
10.	[npp, [[3,man],categ,noun,nbr,sing], [[7,with],categ,prep]]	[25]
11.	[det, [[9,telescope],categ,noun,nbr,sing], [[8,a],categ,art,ty,indef]]	[14]
12.	[ppn, [[7,with],categ,prep], [[9,telescope],categ,noun,nbr,sing]]	[15]

Figure 2 Grammar Rules and Example triples.

[1,see,v], and the node [3,man,n] represents an NP with head word [3,man,n]. [1,see,v] becomes the head word, and [3,man,n] becomes the modifier word, of this triple. The number 1 in [1,see,v] is the position of the word "see" in the sentence, and v (verb) is the syntactic category of the word. Since a word may appear in several positions in a sentence, and one word may have multiple categories, the position and the category of a word must be recorded to distinguish the same word in different positions or with different categories.

A meaningful relation name is assigned to each pair of head and modifier constituents in a grammar rule. Some of these are shown at the top of Figure 2. Rules for generating triples augment each corresponding grammar rule. Some grammar rules in Prolog syntax used to build syntactic graphs are shown in Figure 3.

An informal description of the algorithm for generating triples of a syntactic graph using the grammar rules in Figure 3 is the following: The basic algorithm of the parser is an all-path, bottom-up, chart parser that constructs a shared, packed-parse forest. Unlike an ordinary chart parser, the parser uses two charts, one for

```
%% 1. snt → np + vp
gr([snt, Vhd],                  % category and head of LHS of rule.
   [[np, Nhd], [vp, Vhd]],      % categories and heads of RHS.
   ( true ),                    % constraints, in this case, none
   [[snp, Vhd, Nhd]]).          % list of triples generated
                                % Vhd is head word, Nhd is modifier.
%% 2. np → article + npl
gr([np, Nhd],                   % Nhd, the head of npl, becomes new head
   [[art, Det], [npl, Nhd]],
   ( true ),
   [[det, Nhd, Det]]).          % Nhd is head and Det is modifier.

%% 3. np → npl
gr([np, Nhd],
   [[npl, Nhd]],                % since there is only one constituent
   ( true ),                    % in here
   [ ]).                        % no triple will be generated in
                                % this rule
  %% 4. vp → be_aux + vp        (be + vp) either passive or progressive
  gr([vp, Aux],
    [[be_aux, Aux], [vp, Vhd]],
    ( mempr([inflection, INFL], Vhd),
      ( INFL = paprt            % if inflection of vp is passive
        →                       % participle, then
        Triples = [[be_aux, Aux, Vhd], [voice, Vhd, passive]]
       ;                        % otherwise,
        ( INFL = prprt          % if inflection is present participle
          →                     % then,
          Triples = [[be_aux, Aux, Vhd], progressive, Vhd, yes]]
         ;                      % otherwise,
          fail ) ) ),           % this rule cannot be applied.
Triples).
```

Figure 3 Augmented grammar rules for triple generation.

constituents and the other for triples. Whenever the parser builds a constituent and its triple, the parser generates an index for the triple,[9] and records the triple on the chart of triples using the index. Then it records the constituent with the index of the triple on the chart of constituents.

We use Rule 4 in Figure 3 to illustrate the parser. Rule 4 states that if there are two adjacent constituents, a be-aux followed by a vp, execute the procedure in the third argument position of the rule. The procedure contains the constraints that must be satisfied to make the rule to be fired. If the procedure succeeds, the parser records a new constituent [vp,Vhd]—the first argument of the rule—on the chart. Before the parser records the constituent, it must check the triples for the constituent. The procedure in the third argument position also contains the processes to produce the triples for the constituent.

The fourth argument of a grammar rule is a list of triples produced by executing the augmenting procedure at the third argument position of the rule. If the constraints in the procedure are satisfied, the triples are also produced. The parser generates a unique index for each triple, records the triples on the chart of triples, and adds to the new constituent, the indices of the new triples. Then, the new constituent is recorded on the chart of constituents. In this example, the head of the new constituent is the same as that of be-aux; i.e., the be-aux dominates the vp.

After finishing the construction of the shared, packed-parse forest of an input sentence, the parser navigates the parse forest to collect the triples that

```
pointer [category, head, list of childnodes and index
                                             of triple]
1047    [snt,      [1,see],         [[1002, 1046], 22] ]
1002    [np,       [0,i],           [[1001], notriple] ]
1001    [npl,      [0,i],           [[1000], notriple] ]
1000    [n,        [0,i],           [] ]
1046    [vp,       [1,see],         [[1045], notriple] ]
1045    [vpl       [1,see],         [[1004, 1044], 21],
                                    [[1013, 1041], 24],
                                    [[1027, 1037], 26] ]
1027    [vpl,      1,see],          [[1004, 1026], 10],
                                    [[1013, 1023], 13] ]
1013    [vpl,      [1,see],         [[1004, 1012], 03] ]
1004    [vpl,      [1,see],         [[1003], notriple] ]
1003    [verb,     [1,see],         [] ]
1012    [np,       [3,man],         [[1008, 1011], 02] ]
1008    [art,      [2,a],           [] ]
1011    [np,       [3,man],         [[1010], notriple] ]
1010    [npl,      [3,man],         [[1009], notriple] ]
1009    [noun,     [3,man],         [] ]
1023    [pp,       [4,on],          [[1017, 1022], 07] ]
1017    [prep,     [4,on],          [] ]
1022    [np,       [6,hill],        [[1018, 1021], 06] ]
1018    [art,      [5,the],         [] ]
1021    [np,       [6,hill],        [[1020], notriple] ]
1020    [npl,      [6,hill],        [[1019], notriple] ]
1019    [noun,     [6,hill],        [] ]
1026    [np,       [3,man],         [[1008, 1025], 09] ]
1025    [np,       [3,man],         [[1024], notriple] ]
1024    [npl,      [3,man],         [[1010, 1023], 08] ]
1037    [pp,       [7,with],        [[1031, 1036], 15] ]
1031    [prep,     [7,with],        [] ]
1036    [np,       [9,telescope],   [[1032, 1035], 14] ]
1032    [art,      [8,a],           [] ]
1035    [np,       [9,telescope],   [[1034], notriple] ]
1034    [npl,      [9,telescope],   [[1033], notriple] ]
1033    [noun,     [9,telescope],   [] ]
1041    [pp,       [4,on],          [[1017, 1040], 18] ]
1040    [np,       [6,hill],        [[1018, 1039], 17] ]
1039    [np,       [6,hill],        [[1038], notriple] ]
1038    [npl,      [6,hill],        [[1020, 1037], 16] ]
1044    [np,       [3,man],         [[1008, 1043], 20] ]
1043    [np,       [3,man],         [[1042], notriple] ]
1042    [npl,      [3,man],         [[1010, 1041], 19],
                                    [[1024, 1037], 25] ]
```

Figure 4 Shared, Packed-Parse Forest.

participate in each correct syntactic analysis of the sentence. The collecting algorithm is explained in Section 5.2 in detail.

The representation of the shared, packed-parse forest for the example in Figure 2 is in figures 4 and 5.[10] It is important to notice that the shared, packed-parse forest generated in this parser is different from that of other parsers. In the shared, packed-parse forest defined by Tomita (1985), any constituents that have the same category and span the same terminal nodes are regarded as the same constituent and packed into one node. In the parser for syntactic graphs, the packing condition is slightly different in that each constituent is identified by the head word of the constituent as well as the category and the terminals it spans. Therefore, although two nodes might have the same category and span the same terminals, if the nodes have different head words, then they cannot be packed together.

A **packed node** contains several nodes, each of which contains the category of the node, its head word, and the list of the pointers to its child nodes and the indices of the triples of the node. Node 1045 in Figure 4 is a packed node in which three different constituents are packed. Those three constituents have the same category, vp1, span the same terminals, (from [1,see,v] to [9,telescope,n]), with the same head word, ([1,see,v]), but with different internal substructures.

Note that several constituents may have different indices that point to the same triple. For example, in Figure 4, the first vp1 in the packed node 1045 has the index 21, the first vp1 in the packed node 1027 has the index 10, and the vp1 node in the packed node 1013 has the index 13 as the indices of their triples. Actually, these three indices represent the same triple, Triple 3 in Figure 2. Those three constituents have the same category, vp1, the same head, [1,see,v], and the same modifier, [3,man,n], but have different inside structures of the modifying constituent, np, whose head is [3,man,n]. The modifying constituent, np, may span from [2,a] to [3,man], from [2,a] to [6,hill], or from [2,a] to [9,telescope].

There are different types of triples that do not have head-modifier relations. These types of triples are for syntactic characteristics of a sentence such as mood and voice of verbs. For example, grammar rule 4 in Figure 3 generates not only triples of head-modifier relations, but also triples that have the information about the voice or progressiveness of the head word of the VP, depending on the type of inflection of the word. This kind of information can be determined in syntactic processing and is used effectively in higher level semantic processing.

4 Properties of Syntactic Graphs

We first define several terms used frequently in the rest of the paper.

Definition 1: *An **in-arc** of a node in a syntactic graph is an arc which points to the node, and an **out-arc** of a node points away from the node.*

Since, in the syntactic graph representation, arcs point from dominator to modifier nodes, a node with an in-arc is the modifier node of the arc, and a node with an out-arc is the dominator node of the arc.

Definition 2: *A **reading** of the syntactic graph of a sentence is one syntactic interpretation of the sentence in the syntactic graph.*

Since a syntactic graph is a union of syntactic analyses of a sentence, one reading of a syntactic graph is analogous to one parse tree of a parse forest.

Definition 3: *A **root node** of one reading of a syntactic graph is a node which has no in-arc in the reading.*

In most cases, the root node of a reading of the syntactic graph of a sentence is the head verb of the sentence in that syntactic interpretation. In a syntactically ambigu-

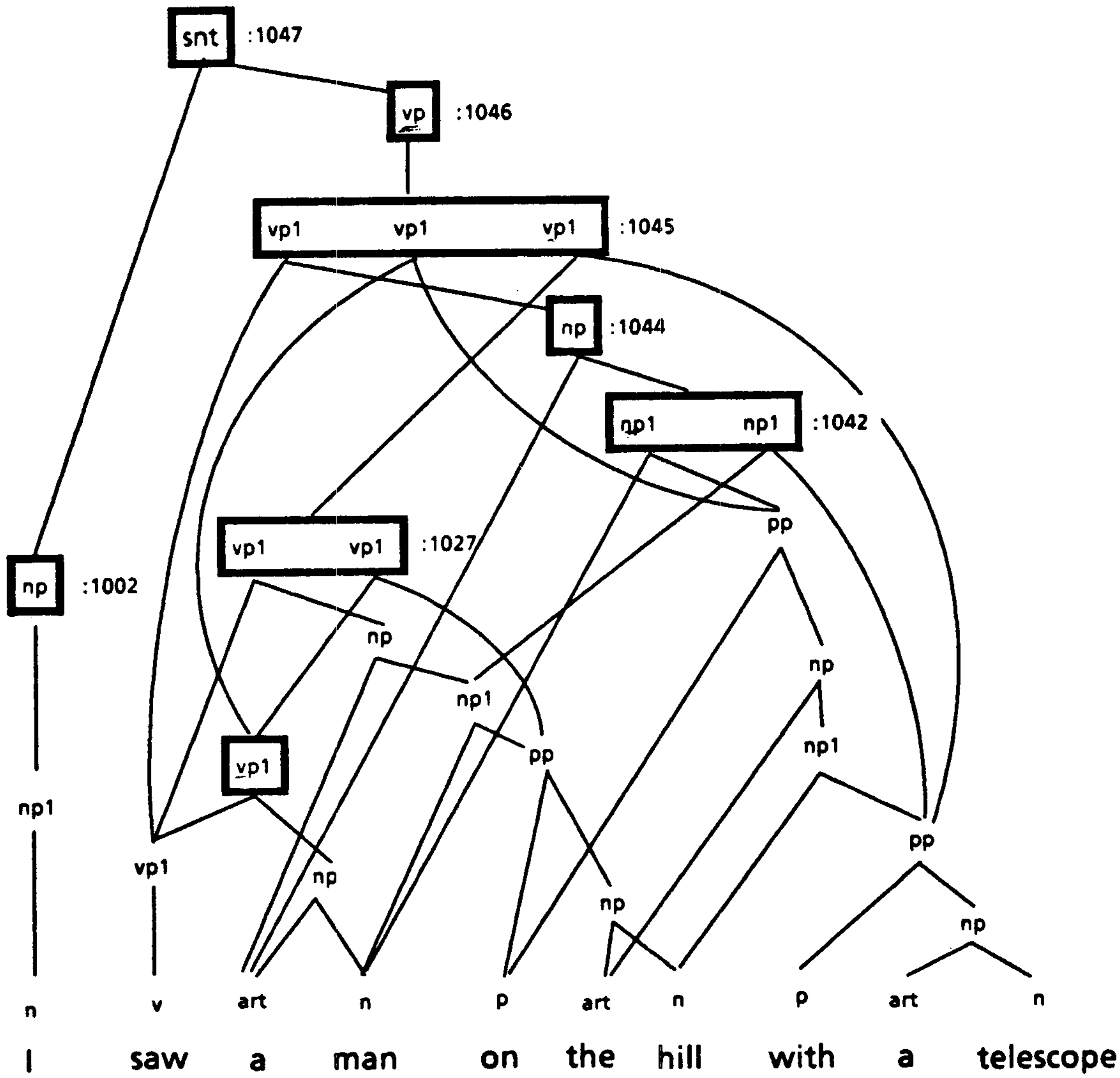

Figure 5 Shared, Packed-Parse Forest-A Diagram.

ous sentence, different syntactic analyses of the sentence may have different head verbs; thus there may be more than one root node in a syntactic graph. For example, in the syntactic graph of one famous and highly ambiguous sentence—"Time flies like an arrow"—shown in Figure 6, there are three different root nodes. These roots are [0,time,v], [1,fly,v], and [2,like,v][11].

Definition 4: *The **position of a node** is the position of the word which is represented by the node, in a sentence.*

Since a word may have several syntactic categories, there may be more than one node with the same position in a syntactic graph. For example, since the word "time" in Figure 6, which appeared as the first word in the sentence, has two syntactic categories, noun and verb, there are two nodes, [0,time,n] and [0,time,v], in the syntactic graph, and the position of the two nodes is 0.

One of the most noticeable features of a syntactic graph is that ambiguities are explicit, and can be easily detected by semantic routines that may use further knowledge to resolve them. The following property explains how syntactically ambiguous points can be easily determined in a syntactic graph.

Property 1: *In a syntactic tree, each constituent except the root must by definition be dominated by a single constituent. Since a syntactic graph is the union of all syntactic trees that the grammar derives from a sentence, some graph nodes may be dominated by more than one node; such nodes with multiple dominators have multiple in-arcs in the syntactic graph and show points at which the node participates in more than one syntactic tree interpretation of a sentence. In a graph resulting from a syntactically unambiguous sentence, no node has more than a single in-arc, and the graph is a tree with the head verb as its root.*

According to Property 1, no pair of arcs which point to the same node can co-occur in any one syntactic

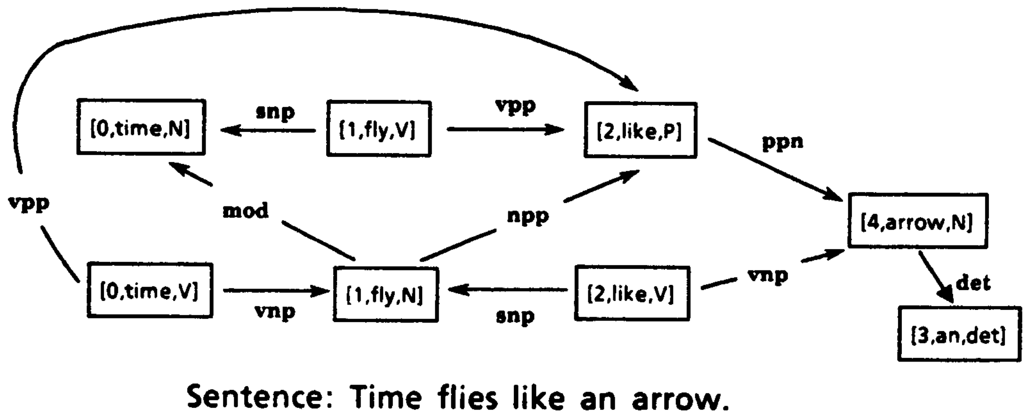

Figure 6 Graph Representation and Parse Trees of a Highly Ambiguous Sentence.

reading, because each node can be a modifier node only once in one reading. Therefore, we can focus on the arcs pointing to the same node as ambiguous points. In terms of triples, any two triples with identical modifier terms reveal a point of ambiguity, where a modifier term is dominated by more than one node.

In the example in Figure 1, the syntactic ambiguities are found in two arcs pointing to [4,on,p] and in three arcs pointing to [7,with,p]. The PP with head [4,on] modifies the VP whose head is [1,see] and it also modifies the NP with head [3,man]. Similarly three different in-arcs to the node [7,with] show that there are three possible choices to which Node 7 can be attached. The semantic processor can focus on these three possibilities (or on the earlier two possibility set), using semantic information, to choose one dominator. Lacking semantic information, the ambiguities will remain in the graph until they can be resolved by additional knowledge from the context.

Property 2: *Since all words in a sentence must be used in every syntactic interpretation of the sentence and no word can have multiple categories in one interpretation,* ***one and only one*** *node from each position* ***must*** *participate in every reading of a syntactic graph. In other words, each syntactic reading derived from a syntactic graph must contain one and only one node from every position.*

Since every node, except the root node, must be attached to another node as a modifier node, we can conclude the following property from properties 1 and 2.

Property 3: *In any one reading of a syntactic graph, the following facts must hold:*

1. No two triples with the same modifier node can co-occur.
2. One and only one node from each position, except the root node of the reading, must appear as a modifier node.

Another advantage of the syntactic graph representation is that we can easily extract the intersection of all possible syntactic readings from it. Since one node from each position must participate in every syntactic reading of a syntactic graph, every node which is not a root node and has only one in-arc, must always be included in every syntactic reading. Such unambiguous nodes are common to the intersections of all possible readings. When we know the exact locations of several pieces in a jigsaw puzzle, it is much easier to place the other pieces. Similarly, if a semantic processor knows which arcs must hold in every reading, it can use these arcs to constrain inferences to understand and disambiguate.

Property 4: *There is no information in a syntactic graph about the range of terminals spanned by each triple, so one triple may represent several constituents which have the same head and modifying terms, with the same relation name, but which span different ranges of terminals.*

The compactness and handiness of a graph representation is based on this property. One arc between two nodes in a syntactic graph can replace several complicated structures in the tree representation, and multiple dominating arcs can replace a parse forest.

For example, the arc vnp from [1,see,v] to [3,man,n] in Figure 1 represents three different constituents. Those constituents have the same category, vp1, the same head, [1,see,v], and the same modifier, [3,man,n], but have different inside structures of the modifying constituent, np, whose head is [3,man,n]. The modifying constituent, np, may span from [2,a] to [3,man], from [2,a] to [6,hill], or from [2,a] to [9,telescope]. Actually, in the exclusion matrix described below, each triple with differing constituent structure is represented by multiple subscripts to avoid the generation of trees that did not occur in the parse forest.

Another characteristic of a syntactic graph is that the number of nodes in a graph is not always the same as that of the words in a sentence. Since some words may have several syntactic categories, and each category may lead to a syntactically correct parse, one word may require several nodes. For example, there are eight nodes in the syntactic graph in Figure 6, while there are only five words in the sentence.

5 Exclusion Matrix

A syntactic graph is clearly more compact than a parse forest and provides a good way of representing all possible syntactic readings with an efficient focusing mechanism for ambiguous points. However, since one triple may represent several constituents, and there is no information about the relationships between triples, it is possible to lose some important syntactic information.

This section consists of two parts. In Section 5.1, we investigate a **co-occurrence** problem of arcs in a syntactic graph and suggest the **exclusion matrix,** to avoid the problem. The algorithms to collect triples of a syntactic graph and to construct an exclusion matrix are presented in Section 5.2.

5.1 Co-occurrence Problem Between Arcs

One of the most important syntactic displays in a tree structured parse, but not in a syntactic graph, is the co-occurrence relationship between constituents. Since one parse tree represents one possible syntactic reading of a sentence, we can see whether any two constituents can co-occur in some reading by checking all parse trees one by one. However, since the syntactic graph keeps all possible constituents as a set of triples, it is sometimes difficult to determine whether two triples can co-occur.

If a syntactic graph does not carry the information about exclusive arcs, its representation of all possible syntactic structures may include interpretations not allowed by the grammar and cause extra overhead. For example, after a syntactic processor generates the triples, a semantic processor will focus on the ambiguous points such as triples 4 and 5, and triples 8, 9, and 10 in Figure 2 to resolve the ambiguities. In this case, if the semantic processor has a strong clue to choose Triple 4 over Triple 5, it should not consider Triple 10 as a competing triple with triples 8 and 9 since 10 is exclusive with 4.

Some of the co-occurrence problems can be detected easily. For example, due to Property 1, since there can be only one in-arc to any node in any one reading of a syntactic graph, the arcs that point to the same node cannot co-occur in any reading. Triples including these arcs are called **exclusive triples.** The following properties of the syntactic graph representation show several cases when arcs cannot co-occur. These cases, however, are not exhaustive.

Property 5: *No two crossing arcs can co-occur. More formally, if an arc has* n^1 *-th and* n^2 *-th words as a head and a modifier, and another arc has* m^1 *-th and* m^2 *-th words as a head and a modifier node, then, if* $n_1<m_1<n_2<m_2$ *or* $m_1<n_1<m_2<n_2$*, the two arcs cannot co-occur.*

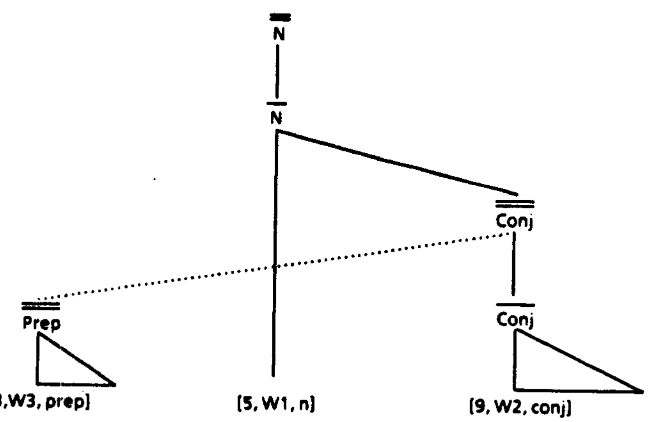

Figure 7 Illegal Parse Tree from Exclusive Arcs.

In the syntactic graph in Figure 1, the arcs vpp from [1,see,v] to [4,on,p] and npp from [3,man,n] to [7,with,p] cannot co-occur in any legal parse trees because they violate the rule that branches in a parse tree cannot cross each other.

The following property shows another case of exclusive arcs which cross each other.

Property 6: *In a syntactic graph, any modifier word which is on the right side of its head word cannot be modified by any word which is on the left side of the head word in a sentence. More formally, let an arc have a head word* W_1 *and a modifier word* W_2 *whose positions are* n_1 *and* n_2 *respectively, and* $n_1<n_2$*. Then if another arc has* W_2 *as a head word and a modifier word with position* n_3 *where* $n_3 \leq n_1$*, then those two arcs cannot co-occur.*

Assume that there are two arcs—one is [npp, [5,W1,noun], [9,W2,conj]], and the other is [conjpp, [9,W2,conj], [3,W3,prep]]. The first arc said that the phrase with head word W2 is attached to the noun in position 5. The other triple said that the phrase with head word W3 is attached to the conjunction. This attachment causes crossing branches. The corresponding parse tree for these two triples is in Figure 7. As we can see, since there is a crossing branch, these two arcs cannot co-occur in any parse tree.

The following property shows the symmetric case of Property 6.

Property 7: *In a syntactic graph, any modifier word which is on the left side of its head word cannot be modified by any word which is on the right side of the head word in a sentence.*

Other exclusive arcs are due to lexical ambiguity.

Definition 5: *If two nodes,* W_i *and* W_j*, in a syntactic graph have the same word and the same position but with different categories,* W_i *is in* ***conflict*** *with* W_j*, and we say the two nodes are* ***conflicting nodes.***

Property 8: *Since words cannot have more than one syntactic category in one reading, any two arcs which have* ***conflicting nodes*** *as either a head or a modifier cannot co-occur.*

The example of exclusive arcs involves the vpp arc from [1,fly,v] to [2,like,p] and the vnp arc from [0,time,v] to [1,fly,n] in the graph in Figure 6. Since the two arcs have the same word with the same position, but with different categories, they cannot co-occur in any syntactic reading. By examination of Figure 6, we can determine that there are 25 pairwise combinations of exclusive arcs in the syntactic graph of that five word sentence.

The above properties show cases of exclusive arcs but are not exhaustive. Since the number of pairs of exclusive arcs is often very large in real text (syntactically ambiguous sentences). if we ignore the co-occurrence information among triples, the overhead cost to the semantic processor may outweigh the advantage gained from syntactic graph representation. Therefore we have to constrain the syntactic graph representation to include co-occurrence information.

We introduce the exclusion matrix for triples (arcs) to record constraints so that any two triples which cannot co-occur in any syntactic tree, cannot co-occur in any reading of a syntactic graph. The exclusion matrix provides an efficient tool to decide which triples should be discarded when higher level processes choose one among ambiguous triples. For an exclusion matrix (**Ematrix**), we make an N x N matrix, where N is the number of indices of triples. If Ematrix(i,j) = 1 then the triples with the indices i and j *cannot* co-occur in any syntactic reading. If Ematrix(i,j) = 0 then the triples with the indices i and j *can* co-occur in some syntactic reading.

5.2 An Algorithm to Construct the Exclusion Matrix

Since the several cases of exclusive arcs shown in the previous section are not exhaustive, they are not sufficient to construct a complete exclusion matrix from a syntactic graph. A complete exclusion matrix can be guaranteed by navigating the parse forest when the syntactic processor collects the triples in the forest to construct a syntactic graph.

As we have briefly described in Section 3, when the parser constructs a shared, packed forest, triples are also produced, and their indices are kept in the corresponding nonterminal nodes in the forest.[12] The parser navigates the parse forest to collect the triples—in fact, pointers pointing to the triples—and to build an exclusion matrix.

As we can see in the parse forest in Figure 5, there may be several nonterminal nodes in one packed node. For each packed node, the parser collects all indices of triples in the subforests whose root nodes are the nonterminal nodes in the packed node, and then records those indices to the packed node. After the parser finishes collecting the indices of the triples in the parse forest, each packed node in the forest has a pointer to the list of collected indices from its subforest. Therefore, the root node of a parse forest has a pointer to the list of all indices of all possible triples in the whole forest, and those triples represent the syntactic graph of the forest.

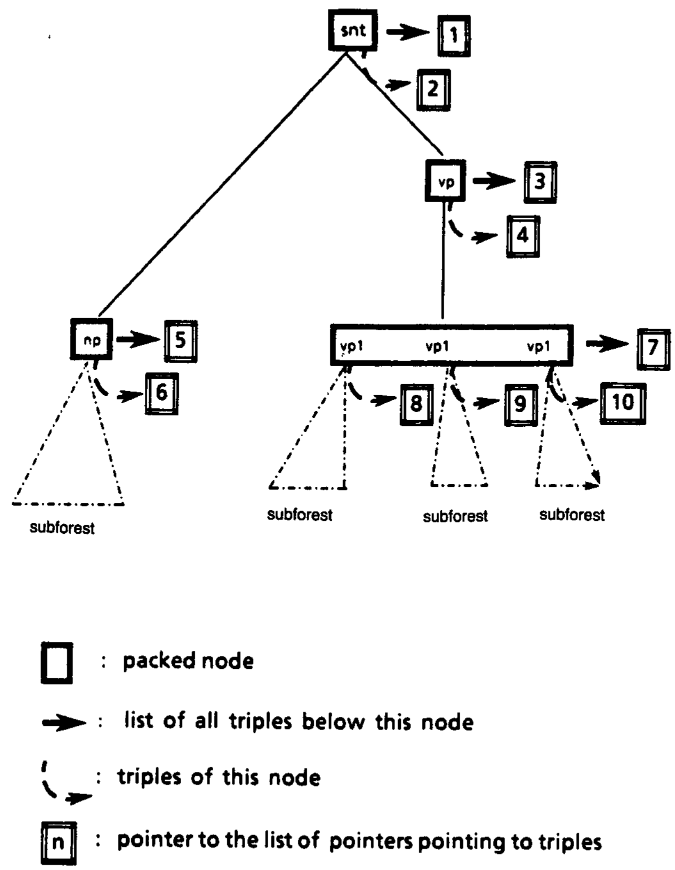

Figure 8 Parse Forest Augmented with Triples.

Figure 8 shows the upper part of the parse forest in Figure 5 after collecting triples. A hooked arrow of each nonterminal node points to the list of the indices of the triples that were added to the node in parsing. For example, pointer 2 contains the indices of the triples added to the node snt by the grammar rule:

$$\text{snt} \rightarrow \text{np} + \text{vp}$$

A simple arrow for each packed node points to the list of all indices of the triples in the forest of which it is the root. This list is generated and recorded after the processor collects all indices of triples in its subnodes. Therefore the arrow of the root node of the whole forest, Pointer 1, contains the list of all indices of the triples in the whole forest.

Since several indices may represent the same triple, after collecting all the indices of the triples in the parse forest, the parser removes duplicating triples in the final representation of the syntactic graph of a sentence.

Collecting pointers to triples in the subforest of a packed node and constructing the Ematrix is done recursively as follows: First, Ematrix(i,j) is initialized to 1, which means all arcs are marked exclusive of each other. Later, if any two arcs indexed with i and j

```
function collect_triple(Packed_node)
  if Packed_node. Collected
                          % if the indices of triples are already collected
    then return(Packed_node. TripleIndex)  % collected then, return
                                            % the collected indices
    else Packed_node. TripleIndex := collectl(Packed_node)
                                            % else collected them
        Packed_node. Collected := true     % set flag Collected.
        return(Packed_node. TripleIndex)   % return collected indices.

function collectl(Packed_node)
  Triple_Indices := { }
  for each Node in Packed_node do
    TRiple_Indices := merge(Node.TripleIndex, Triple_Indices)
    case Node.Child_node_num
     0 : (do nothing)
     1 : Temp := collect_triple(Node.Child_node)
         Triple_Indices := merge(Temp, Triple_Indices)
         co-occuri(Node.TripleIndex, Temp)
     2 : Templ := collect_triple(Node.Left_child)
         Temp2 := collect_triple(Node.Right_child)
         Triple_Indices := merge(Templ, Temp2, Triple_Indices)
         co-occur2(Node.TripleIndex, Templ, Temp2)
  end-do
  return(Triples)
```

Figure 9 An Algorithm to Collect Triples.

```
function cooccurl(Tripl, Trip2)
  fully_cooccur(Tripl)
  co-occur3(Tripl, Trip2)

function cooccur2(Tripl, Trip2, Trip3)
  fully_cooccur(Tripl)
  cooccur3(Tripl, Trip2)
  cooccur3(Tripl, Trip3)
  cooccur3(Trip2, Trip3)

function cooccur3(Tripl, Trip2)
  for each index i in Tripl do
    for each index j in Trip2 do
      Ematrix(i, j) := 0
      Ematrix(j, i) := 0   /* Ematrix is symmetric */

function fully_cooccur(Triples)
  for each pair of indices i and j in Triples do
    Ematrix(i, j) := 0
```

Figure 10 An Algorithm to Construct the Exclusion Matrix.

co-occur in some parse tree, then the value of Ematrix, E(i,j), is set to 0. For each nonterminal node in a packed node, the parser collects every index appearing below the nonterminal node—i.e., the index of the triples of its subnodes. If a subnode of the nonterminal node was previously visited, and its indices were already collected, then the subnode already has the pointer to the list of collected indices. Therefore the parser does not need to navigate the same subforests again, but it takes the indices using the pointer. The algorithm in pseudo-PASCAL code is in Figure 9.

After the parser collects the indices of the triples from the subnodes of the nonterminal node, it adjusts the values in the exclusion matrix according to the following cases:

1. If the nonterminal node has one child node, its own triples can co-occur with each other, and with every collected triple from its subforest.
2. If the nonterminal node has two child nodes, its own triples can co-occur with each other and with the triples collected from both left and right child nodes, and the triples from the left child node can co-occur with the triples from the right one.

This algorithm is described in Figure 10.

For example, the process starts to collect the indices of the triples from SNT node in Figure 8. Then, it collects the indices in the left subforest whose root is np. After all indices of triples in the subforest of np are collected, those indices and the indices of the triples of the node in 6 are recorded in 5. Similarly all indices in 7 and 4 are recorded in 3 as the indices of the triples in the right subforest of the snt node. The indices in 5 and 3 and the indices in 2 are recorded in 1 as the indices of the triples of the whole parse forest. In packed nodes with more than one nonterminal node, like vp1, all indices of the triples in the three subforests of vp1 and the indices in 8, 9, and 10 are collected and recorded in 7.

By the first case in the above rule, every triple represented by the indices in 4 can co-occur with each other, and every triple represented by the indices in 4 can co-occur with every triple represented by the indices in 7. One example of the second case is that every triple represented by the indices in 2 can co-occur with each other, and every triple represented by the indices in 2 can co-occur with every triple represented by the indices in 5 and 3. Every triple represented by the indices in 5 can co-occur with the triples represented by the indices in 3. Whenever the process finds a pair of co-occurring triples it adjusts the value of Ematrix appropriately.

6 Completeness and Soundness of the Syntactic Graph

In this section, we will discuss **completeness** and **soundness** of a syntactic graph with an exclusion matrix as an alternative for tree representation of syntactic information of a sentence.

Definition 6: *A syntactic graph of a sentence is* ***complete*** *and* ***sound*** *compared to the parse forest of the sentence* ***iff*** *there is an algorithm that enumerates syntactic readings from the syntactic graph of the sentence and satisfies the following conditions:*
1. For every parse tree in the forest, there is a syntactic reading from the syntactic graph that is structurally equivalent to that parse tree. ***(completeness)***
2. For every syntactic reading from the syntactic graph, there is a parse tree in the forest that is structurally equivalent to that syntactic reading. ***(soundness)***

To show the completeness and soundness of the syntactic graph representation, we present the algorithm that enumerates all possible syntactic readings from a syntactic graph using an exclusion matrix. This algo-

```
The following data are initial input.
Partition_I = a list of triples which have the I-th word as a modifier.
Sen_length  = the position of the last word in a sentence.
RootList    = a list of root triples.

gen_subgraph(RootList, Sen_length, Graphs, All_readings) :-
  ( RootList = []                  % if all root triple in RootList had been tried
    -> All_readings = Graphs       % then return Graphs as all readings,
                                   % otherwise, find all readings with a RootTriple
   ; RootList = [RootTriple| RootList1],
     gen_subgraph1(RootTriple, Sen_length, Sub_graphs),
     append(Graphs, Sub_graphs, Graphs1),
     gen_subgraph(RootList1, Graphs1, All_readings) ).

gen_subgraph1(RootTriple, Sen_length, Sub_graphs) :-
   Rh = Position of the head node in RootTriple   % i.e., position of the root node
   Rm = Position of the modifier node in RootTriple,
   Wlist = [RootTriple],
   setof(Graph, gen_sub1(Rh, Rm, Sen_length, Wlist, Graph, 0), Sub_graphs).

gen_sub1(Rh, Rm, Sen_length, Wlist, Graph, N) :-
  (N > Sen_length      % if it take a triple from all partitions
   -> Graph = Wlist    % then return Wlist as one reading of a syntactic graph,
                       % otherwise, pick one triple from partition N.
  ; ( ( Rh = N         % Don't pick up any triple from root node position.
      ; Rm = N)        % A triple from partition Rm is already picked in Wlist.
    -> true
   ; get_triple(N, Triple), % take a triple(in fact, an index of the triple) from partition N.
     not_exclusive(Wlist, Triple), % check exclusiveness of Triple with other triples in Wlist.
     N1 is N + 1,                  % go to the next partition.
     gen_sub1(Rh, Rm, [Triple| Wlist], Graph, N1) )).
```

Figure 11 Algorithm that Generates All and Only Readings from an SG.

rithm constructs subgraphs of the syntactic graph, one at a time. Each of these subgraphs is equivalent to one reading of the syntactic graph. Since no node can modify itself, each of these subgraphs is a **directed acyclic graph** (DAG). Furthermore, since every node in each of these subgraphs can have no more than one in-arc, the DAG subgraph is actually a tree.

Before going into detail, we give an intuitive description of the algorithm. The algorithm has two lists of triples as input: a list of triples of a syntactic graph and a list of root triples. A root triple is a triple that represents the highest level constituent in a parse—i.e., snt (sentence) in the grammar in Figure 2. The head node of a root triple is usually the head verb of a sentence reading.

According to Property 3 in Section 4, one reading of a syntactic graph must include one and only one node from every position, except the position of the root node, as a modifier node. This is a necessary requirement for any subgraph of a syntactic graph to be one reading of the graph. One of the simplest ways to make a subgraph of a syntactic graph that satisfies this requirement is:

- Make partitions among triples according to the position of the modifier node of the triples, e.g., triples in Partition 0 have the first word in a sentence as the modifier nodes. Then take one triple from each partition. Here, the algorithm must know the position of the root node so that it can exclude the partition in which triples have the root node as a modifier. When it chooses a triple, it also must check the exclusion matrix. If a triple from a partition is exclusive with any of the triples already chosen, the triple cannot be included in that reading. The algorithm must try another triple in that partition. Since the exclusion matrix is based on the indices of the triples, when it chooses a triple, it actually chooses an index in the list of indices of the triple.

Note that any subgraphs produced in this way satisfy Property 3, and all triples in each subgraph are inclusive with each other according to the exclusion matrix. The top level procedures of the algorithm in Prolog are shown in Figure 11.[13]

We do not have a rigorous proof of the correctness of the algorithm, but we present an informal discussion about how this algorithm can generate all and only the correct syntactic readings from a syntactic graph.

Since the syntactic graph of a sentence is explicitly constructed as a union of all parse trees in the parse forest of the sentence, the triples of the syntactic graph imply all the parse trees. This fact is due to the algorithm that constructs a syntactic graph from a parse forest. Therefore, if we can extract all possible syntactic readings from the graph, these readings will include all possible (and more) parse trees in the forest. Intuitively, the set of all subgraphs of a syntactic graph includes all syntactic readings of a syntactic graph.

In fact, this algorithm generates all possible subgraphs of a syntactic graph that meets the necessary conditions imposed by Property 3. The predicate gen_sub1 generates one reading with a given root triple. All readings with a root triple are exhaustively collected by the predicate gen_subgraph1 using the setof predicate—a meta predicate in Prolog. All readings of a syntactic graph are produced by the predicate gen_subgraph, which calls the predicate gen_subgraph1 for each root triple in RootList. Therefore, this algorithm generates all subgraphs of a syntactic graph that satisfy Property 3 and that are consistent with the exclusion matrix. Hence, the set of subgraphs generated by the algorithm includes all parse trees in the forest.

The above algorithm checks the exclusion matrix when it generates subgraphs from the syntactic graph, so all triples in each subgraph generated by the algorithm are guaranteed to co-occur with each other in the exclusion matrix. Unfortunately, it does not appear possible to prove that if triples, say T1 and T2, T2 and T3, and T1 and T3, all co-occur in pairs, that they must all three co-occur in the same tree! So, although empirically all of our experiments have generated only trees from the forest, the exclusion matrix does not provide mathematical assurance of soundness.

If subsequent experience with our present statistically satisfactory, but unsound exclusion matrix requires it, we can produce, instead, an inclusion matrix that guarantees soundness. The columns of this matrix are I.D. numbers for each parse tree; the rows are triples. The following procedure constructs the matrix.

1. Navigate the parse forest to extract a parse tree, I, and collect triples appearing in that parse tree.
2. Mark matrix(Tindex, I) = 1, for each triple with the index Tindex appearing in the I-th parse tree. Backtrack to step 1 to extract another possible parse tree until all parse trees are exhausted.

Then, given a column number i, all triples marked in that column co-occur in the i-th parse tree. Since this algorithm must navigate all possible parse trees one by one, it is less efficient than the algorithm for constructing the exclusion matrix. But if our present system eventually proves unsound, this inclusion matrix guarantees that we can test any set of constituents to determine unequivocally if they occur in a single parse tree from the forest.

Therefore, we claim that syntactic graphs enable us to enumerate all and only the syntactic readings given in a parse forest, and that syntactic graph representation is complete and sound compared to tree representations of the syntactic structure of a sentence.

7 Related Works

Several researchers have proposed variant representations for syntactic structure. Most of them, however, concentrated on how to use the new structure in the parsing process. Syntactic graph representation in this work does not affect any parsing strategy, but is constructed after the syntactic processor finishes generating a parse forest using any all path parser.

Marcus et. al. (1983) propose a parsing representation that is also different from tree representation. They use the new representation for a syntactic structure of a sentence to preserve information, while modifying the structure during parsing, so that they can solve the problems of a deterministic parser (Marcus 1980)—i.e., parsing garden path sentences. Marcus's representation consists of dominator-modifier relationships between two nodes. It is, however, doubtful that a correct parse tree can be derived from the final structure, which consists of only domination relationships. They do not represent all possible syntactic readings in one structure.

Barton and Berwick (1985) also discuss the possibility of a different representation, an "assertion set", as an alternative for trees, and show various advantages expected from the new structure. As in Marcus's work, they use the assertion set to preserve information as parsing progresses, so that they can make a deterministic parser to be partially noncommittal, when the parser handles ambiguous phrases. Their representation consists of sets of assertions. Each assertion that represents a constituent is a triple that has the category name and the range of terminals that the constituent spans. It is unclear how to represent dominance relationships between constituents with assertion sets, and whether the final structure represents all possible parses or parts of the parses.

Rich et. al. (1987) also propose a syntactic representation in which all syntactic ambiguities are kept. In this work, the ambiguous points are represented as one modifier with many possible dominators. Since, however, this work also does not consider possible problems of exclusive attachments, their representation loses some information present in a parse forest.

Tomita (1985) also suggests a disambiguation process, using his shared, packed-parse forest, in which all possible syntactic ambiguities are stored. The disambiguation process navigates a parse forest, and asks a user whenever it meets an ambiguous packed node. It does a "shaving-a-forest" operation, which traverses the parse forest to delete ambiguous branches. Deleting one arc accomplishes the "shave" in the syntactic graph representation. Furthermore, in a parse forest, the ambiguous points can be checked only by navigating the forest and are not explicit.

Since a parse forest does not allow direct access to its internal structure, a semantic processor would have to traverse the forest whenever it needed to check internal relations to generate case relations and disambiguate without a user's guidance. Syntactic graph representation provides a more concise and efficient structure for higher level processes.

8 Conclusion

In this paper, we propose the syntactic graph with an exclusion matrix as a new representation of the surface syntactic structure of a sentence. Several properties of syntactic graphs are examined. An algorithm that enumerates all and only the correct syntactic readings from a syntactic graph is also presented. Therefore, we claim that syntactic graph representation provides a concise way to represent all possible syntactic readings in one structure without losing any useful information contained in the tree structured representation.

To further justify that syntactic graph representation is a suitable formalism for an output format of syntactic processes, we need to investigate methods for using syntactic graphs to make correct decisions in higher level processes. The exclusion matrix is an efficient tool to help semantic processes make correct choices.

Because of its conciseness, the syntactic graph makes it possible to store temporarily the syntactic structure of sentences that already have been processed. A text understanding process is very likely to find contradicting evidence between a current sentence and the context of the previous sentences. If we did not keep alternative analyses of previous sentences the only thing we could do is backtracking, which is computationally too expensive. Furthermore, since the search space of the syntactic processor is different from that of the semantic processor, it is very important for the syntactic process to commit to a final result. We are currently investigating how to use syntactic graphs of previous sentences to maintain a continuous context whose ambiguity is successively reduced by additional incoming sentences.

Acknowledgments

This work is sponsored by the Army Research Office under contract DAAG29-84-K-0060. The authors are grateful to Olivier Winghart for his critical review of an earlier draft of this paper.

REFERENCES

Aho, A. V. and Ullman, J. D. 1972 *The Theory of Parsing, Translation and Compiling* 1. Prentice-Hall, Englewood Cliffs, NJ.

Barton, G. E. and Berwick, R. C. 1985 "Parsing with Assertion Sets and Information Monotonicity." In *Proceedings of International Joint Conference on Artificial Intelligence-85* (IJCAI-85): 769–771.

Birnbaum, L. and Selfridge, M. 1981 "Conceptual analysis of natural language." In R. Schank and C. Riesbeck, eds., *Inside Computer Understanding*. Lawrence Erlbaum, Hillsdale, NJ.

Chester, D. 1980 "A Parsing Algorithm that extends Phrases." *American Journal of Computational Linguistics* 6 (2): 87–96.

Chomsky, N. 1970 "Remarks on nominalization." In R. Jacobs and P. S. Rosenbaum, Eds., *Readings in English Transformational Grammar*. Waltham, MA: Ginn & Co.

Chomsky, N. 1981 Lectures on Government and Binding. Foris, Dordrecht, Holland.

Early, J. 1970 "An Efficient Context-free Parsing algorithm." *Comm* ACM 13, (2): 94–102.

Frazier, L. and Fodor, J. 1979 "The Sausage Machine: A New Two-Stage Parsing Model." *Cognition* 6: 41–58.

Kay, M. 1980 "Algorithm Schemata and Data Structures in Syntactic Processing." Xerox Corporation, Technical Report Number CSL-80-12, Palo Alto, CA.

Lytinen, S. L. 1986 "Dynamically Combining syntax and semantics in natural language processing." In *Proceedings of The American Association for Artificial Intelligence-86*(AAAI-86): 574–578.

Marcus, M. P. 1980 *A Theory of Syntactic Recognition for Natural Language*. MIT Press, Cambridge, MA.

Marcus, M. P.; Hindle, D.; and Fleck, M. M. 1983 "D-Theory: Talking about Talking about Trees." In *Proceedings of 21st Annual Meeting of the Association for Computational Linguistics*: 129–136.

Pereira, F. C. N. and Warren, D. H. 1980 "Definite Clause Grammars — A survey of the formalism and a Comparison with Augmented Transition Network." *Artificial Intelligence*, 13: 231–278.

Rich, A.; Barnett, J.; Wittenburg, K.; and Wroblewski, D. 1987 "Ambiguity Procrastination." In *Proceedings of AAAI–87*: 571–576.

Shubert, L. K. 1984 "On Parsing Preferences." In *Proceedings of the Conference on Computational Linguistics 84* Stanford, CA: 247–250.

Shubert, L. K. 1986 "Are There Preference Trade-Offs in Attachment Decision?" In *Proceedings of AAAI–86*: 601–605.

Tomita, M. 1985 *Efficient Parsing for Natural Language*. Kluwer Academic Publishers, Boston, MA.

Tsukada, D. 1987 "Using Dominator-Modifier Relations to Disambiguate a Sentence" (master's thesis), Department of Computer Sciences, University of Texas at Austin.

Waltz, D. L. 1982 "The State of the Art in Natural Language Understanding." In W. Lehnert and M. Ringle (eds.), *Strategies for Natural Language Processing*, Lawrence Erlbaum Associates, Inc., Hillsdale, NJ.

Wilks, Y.; Huang, X.; and Fass, D. 1985 "Syntax, Preference and Right Attachment." In *Proceedings of International Joint Conference on Artificial Intelligence–85* (IJCAI–85): 779–784.

Winghart, O. J. 1986 "A Processing Model for Recognition of Discourse Coherence Relations" (unpublished Ph.D proposal), Department of Computer Sciences, University of Texas at Austin.

Notes

1. Since we are discussing the syntactic representation, we use the term "semantic processor" for all higher level processors including the semantic, coherence, and discourse processors.
2. By "correct" is meant semantically correct. Here, semantics has a broad meaning including pragmatics.
3. Borrowing a term from Tomita's (1985) system. Although we present an example of a shared, packed-parse forest in Section 3, we refer readers to (Tomita 1985) for more detailed discussion and examples.
4. There are different views of text processing in which syntactic and semantic processors are integrated (Birnbaum and Selfridge 1981; Lytinen 1986; Winghart 1986). However detailed discussion of other control flows is beyond the scope of this work.
5. For the sentence, "It is transmitted by eating shellfish such as oysters living in infected waters, or by drinking infected water, or by dirt from soiled fingers", there are 1433 parses from our context-free grammar.
6. In our experience, a graph representing several hundred parse trees may take less than three times the number of triples as one representing a single interpretation graph for the sentence.
7. In a syntactic graph, however, we call both modifier and specifier nodes **modifier nodes.**
8. From now on, we will use the terms **node** and **word,** as well as **arc** and **triple,** interchangeably.
9. A unique index can be generated using the special function **gensym** which returns a unique symbol whenever it is called.

10. Due to the complexity of the diagram, some of the details are omitted.
11. Not all different readings of a syntactic graph have different root nodes. In this example, [0,time,v] is the root node of two different readings of the graph with simple grammar rules. The equivalent parse trees of the two readings are:

    [snt,[vp,[verb,[time]],[np,[np,[noun,[flies]]],[pp,[prep,[like]],
    [np,[det,[an]] ,[noun,[arrow]]]]]]]]
    [snt,[vp,[vp,[verb,[time]],[np,[noun,[flies]]]],[pp,[prep,[like]],
    [np,[det,[an]],[noun,[arrow]]]]]]]
12. The node in a forest is different from the node in a syntactic graph. A non-terminal node in a forest with two children nodes has one head-modifier relation, and hence the non-terminal with two children in a forest represents one arc in a syntactic graph.
13. We use the syntax of Quintus–Prolog version 2 on SUN systems. The special predicate, (Cond → Then ; Else), in the algorithm can be interpreted as; if Cond is true, then call Then. Otherwise, call Else.