A Preprocessor Based Parsing System

Pradip Peter Dey, Mohammad N. Amin, and Thomas M. Gatton

School of Engineering and Technology
National University
11255 North Torrey Pines Road, La Jolla, CA 92037, U.S.A.
{pdey, tgatton, mamin}@nu.edu

Abstract. A preprocessor based parsing system for Tree Adjoining Grammars is presented. The preprocessor is used primarily for organizing the data structures required for parsing the grammar efficiently. The preprocessor works hard in order to reduce the runtime processing load so that the parser executes fast. At least one model of Tree-Adjoining Grammars allows transfer of significant processing load from the parser to the preprocessor since the adjoining process can be optionally applied before runtime on tree structures. A parallel parsing algorithm is presented that takes advantage of the preprocessor.

1 Introduction

A preprocessor based parsing system is described in this paper. A parsing system basically performs syntactic analyses in natural language processing [1, 2]. The parsing system described here is based on a grammatical formalism called Tree Adjoining Grammar (TAG) developed by Joshi [3, 4, 5]. TAGs seem to be appropriate for our purposes because: (a) they can be easily decomposed into independent modules which can be processed concurrently, (b) they can be developed incrementally since they represent information about the language in a highly modular fashion, (c) they seem to have the formal properties required for processing natural languages [5, 6] and most importantly, (d) the significant processing load of the system can be transferred from the runtime module to the preprocessor. This paper describes the parsing system with a model of TAGs that is appropriate for preprocessor based parsing strategies.

Two finite sets of trees are defined by a TAG along with a composition operation called an adjoin that recursively generates new trees by combining existing trees [3, 4]. The trees serve as structural descriptions of sentences or sentence fragments. The central module of the parser is a pattern matcher that finds every tree that matches the input string. If the input matches two or more trees, then it is structurally ambiguous. If there is no perfect match then the best available approximate match can be determined according to some heuristics. This paper is not concerned with this latter inexact match by which semi-grammatical sentences are parsed. It considers only grammatical sentences. The lexical categories of the input string are determined by a lexical search. These categories are matched with the leaves of the trees stored in TREE BANK or produced dynamically by the adjoin process as shown in Figure 1.

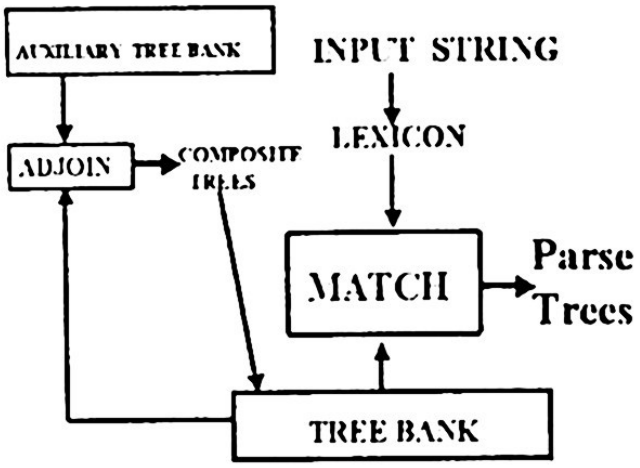

Figure 1: Major Components of a TAG Parser

The tree-bank has a set of trees that correspond to the structural descriptions of sentences of the language. Initially, it has a well-defined set of minimal trees called initial trees that correspond to "simple" sentences of the language. The auxiliary tree-bank has a set of trees called auxiliary trees. Auxiliary trees are used in the adjoining operation to account for recursion. They do not occur independently in the language. In order to improve the performance of the parser, three basic strategies are adopted: (a) Adjoin is applied before runtime through a preprocessor in order to obtain an inflated tree-bank which is sorted according to the length of the trees; (b) the lexicon is ordered so that a parallel binary search can be efficiently applied; (c) the match operation is parallelized so that millions of trees can be searched efficiently. Suppose the input is a string such as "The girl danced". The parser first searches the lexicon and finds that "The" is a determiner (D), "girl" is a noun (N), and "danced" is a verb (V). The category sequence, DNV, for this sentence is then matched with the leaves of the trees from the tree-bank. The tree given in Figure 2. matches the category sequence because it has exactly the same leaves. The result of the match is the top tree in Figure 3. For this tree and for future references, assume N = Noun, V = Verb, A = Adjective, NP = Noun Phrase, VP = Verb Phrase, R = Relative-Pronoun, and S = Sentence.

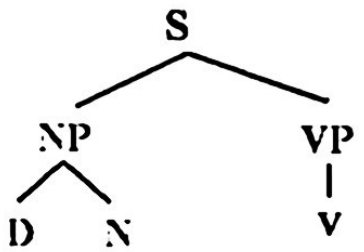

Figure 2. A tree that matches the string "The girl danced"

MATCH simply asks if the categories of the input string are equal to the leaf-nodes of a tree. In the proposed parser, a large number of trees are pre-generated by a preprocessor before runtime in order to reduce runtime processing loads. The TAG parser has at least two advantages over classical parsers: (i) the trees are not built at run time, so it is fast, (ii) the parser can be developed incrementally by adding new trees to the tree-bank. Joshi and Vijay-Shankar described a sequential algorithm that takes O (n^6) time, where n is the length of the input string [6]. We describe an algorithm that takes O(n) time. We present a parallel version of the algorithm that achieves almost linear speed-up. TAGs can be processed very fast with appropriate combination of parallel processing, preprocessing of certain information and heuristic search. It is suggested that natural language systems should be designed to process short sentences very fast, because, in ordinary interactions, a sentence will be approximately a dozen words long. The proposed parser does not fail to process long strings, but their processing is slower because additional runtime processing is required for them where aids from the preprocessor are minimal. Specifically, it has to apply adjoin in runtime to generate large trees for long input.

2 TAGs For Natural Language Parsing

Natural language parsing systems are usually difficult to build because they are expected to achieve computational efficiency and linguistic adequacy. TAGs seem to be appropriate for natural language parsing because they help in achieving these two goals. TAGs have two types of elementary trees: initial trees (IT) and auxiliary trees (AT). All elementary trees are minimal in the sense that they do not exhibit recursion. Recursion is factored by the composition operation adjoin that produces composite trees by combining auxiliary trees with initial or composite trees. The trees are stored in the parser without words or lexical materials. If an input string matches a tree then that tree is returned as an output of the parser after inserting the words of the input string into the tree. A tree, IT_1, is an initial tree if its root is labeled S and no other node is labeled S. A tree, AT_1, is an auxiliary tree if its root and one of the leaf-nodes (frontier nodes) are labeled by the same non-terminal symbol, not necessarily S. The leaf-node that has the same label as the root is called the hook node of AT_1. The adjoining operation can only adjoin an auxiliary tree to a non-auxiliary tree (initial or composite) that has at least one node whose label is the same as that of the root of the auxiliary. This matching node in the initial or composite tree is called the target node.

An example of adjoining from English is given in Figures 3-5 for revealing its application. An initial tree that matches English sentences like "The girl danced" is given at the top of Figure 3. An auxiliary tree that corresponds to an embedded clause like "who ate fish" is given at the bottom of Figure 3. The auxiliary tree of Figure 3 can be adjoined at the NP node of the initial tree of Figure 3, because it matches with the root of the auxiliary tree.

The ultimate result of this adjoining is the composite tree given in Figure 5 which corresponds to the English sentence "The girl who ate fish danced". However, adjoining is a two step process in which the first step is shown in Figure 4. In the first step of adjoining, the sub-trees below the target NP node of the initial tree are

moved to the hook-node NP of the auxiliary tree resulting in the trees of Figure 4. In the next step of adjoining, the root of the auxiliary tree is pasted on the target node of the initial tree. The resulting composite tree is given in Figure 5. The lexical items are inserted into each of the trees to illustrate the correspondence between trees and strings. The composition operation is actually applied on skeletal trees without lexical items or words.

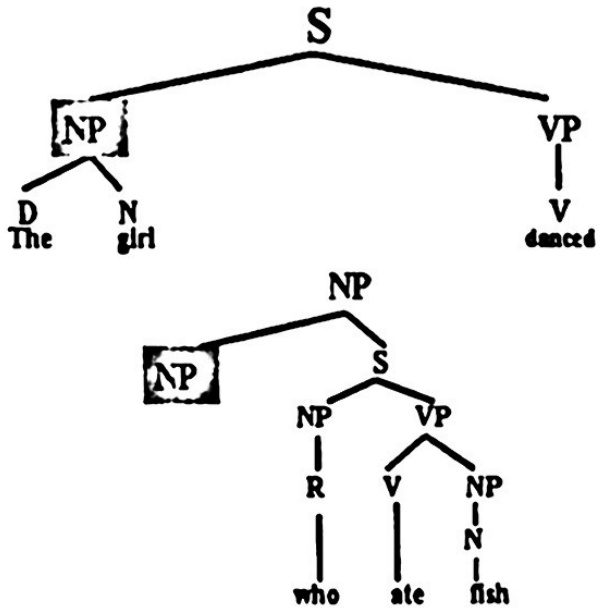

Figure 3. An initial tree at the top and an auxiliary tree at the bottom

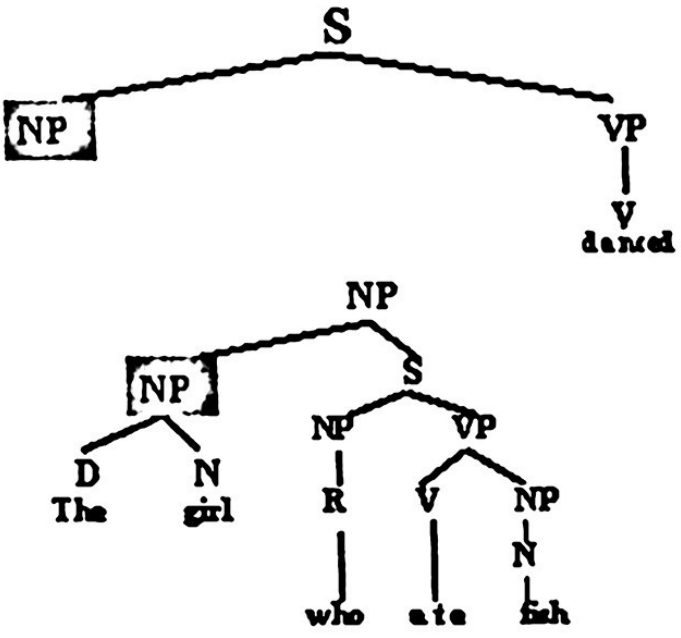

Figure 4. Sub-trees below the target node of the non-auxiliary tree are moved to the hook node of the auxiliary tree

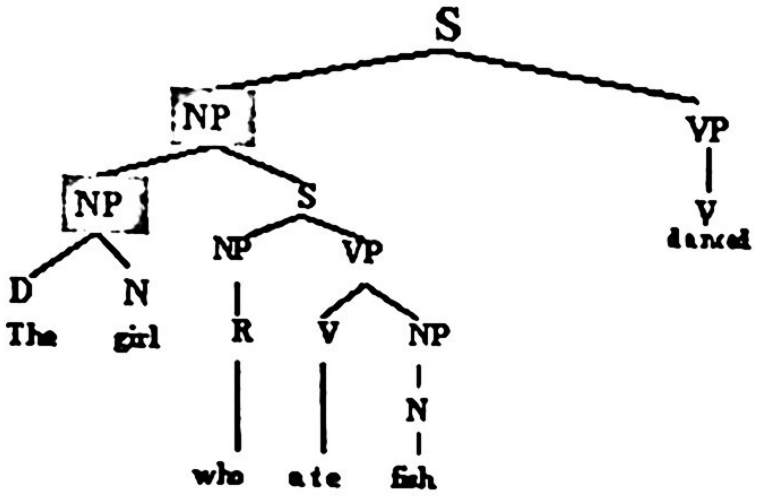

Figure 5: The composite tree produced from the adjoining process

The lexicon is a database in which each word is a key and the lexical category of the word is its main entry. Some words belong to more than one lexical categories resulting lexical ambiguity [7, 8, 9].

The lexical items are inserted into each of the trees to illustrate the correspondence between trees and strings. The composition operation is actually applied on skeletal trees without lexical items or words. The lexicon is a database in which each word is a key and the lexical category of the word is its main entry. Some words belong to more than one lexical categories resulting lexical ambiguity [7, 8, 9].

3 Parsing Algorithms

In order to increase the speed of processing sentences of reasonable length, we maintain a sorted tree-bank with initial and pre-generated composite trees. That is, all trees a maximum of 7 adjoining are pre-generated before runtime and stored in the tree-bank. The trees are ordered in the tree-bank according to their length. All trees of the same length are stored in the same sub-tree-bank marked by the length of the trees. The length of a tree is the number of leaves it has. Depending on the length of the input string, only one of the sub-tree-banks is selected for an exhaustive search. The length of each tree in that sub-tree-bank must be equal to the input length. This strategy along with some heuristics allows efficient searching of the tree-bank. The tree bank (TB) is managed by the procedure PARALLEL-PREGENERATE. It pre-generates composite trees with a maximum of M adjoining and inserts them in the sorted tree bank, where M is an integer. We initially set M to 7. The ADJOIN procedure follows the steps outlined informally in section 2. It takes an auxiliary tree and a non-auxiliary (initial or composite) tree as arguments and returns z composite trees by adjoining the auxiliary tree at z occurrences of the root of the auxiliary tree in the non-auxiliary tree. In order to pre-generate a large number of composite trees and maintain them in a sorted tree-bank, can also be used during runtime. It is an asynchronous algorithm.

Procedure PARALLEL-PREGENERATE(FRONT, AT, TB, M)

FRONT = {I_1, I_2, ..., I_c }, a finite set of non-auxiliary trees. Initially FRONT has only initial trees. AT = {AT_1, AT_2, ..., AT_b }, a finite set of auxiliary trees. TB = (T_1, T_2, ..., T_x), an ordered set of non-auxiliary trees. Initially we present the PARALLEL-PREGENERATE procedure, which is usually applied before runtime although it TB has only initial trees. M = The maximal number of adjoining desired in a tree.

```
begin
1  OPEN ← FRONT // FRONT is copied to OPEN
2  For d = 1 to M do
     begin
3        FRONT ← ( )
4        For j = 1 to | OPEN | create process j
                        // create a process for each tree in OPEN
5        In each process j :
5.1              r ← OPEN_j
5.2              For h = 1 to | AT | Create process h
5.3              In each process h:
5.3.1                    ct ← ADJOIN(AT_h, r)
5.3.2                    For i = 1 to | ct | Create process i
5.3.3                    In each process i :
5.3.3.1                          TB ← UPDATE(ct_i, TB, FRONT)
                                 | if ct_i is not in TB then TB ← INSERT(ct_i,
                                 | TB) & FRONT ← FRONT U ct_i
6        Synchronize
7        OPEN ← FRONT
     end
8  Return TB
end
```

Parallelism in the PARALLEL-PREGENERATE procedure is achieved mainly by data partitioning. It will generate parse trees with desired number of adjoining before runtime. For efficient runtime execution of the parser the PARALLELPARSE procedure is given below.

Procedure PARALLELPARSE(FRONT, AT, TB, W, L, M)

FRONT = {I_1, I_2, ..., I_c }, a finite set of non-auxiliary trees. AT = {AT_1, AT_2, ..., AT_b }, a finite set of auxiliary trees. TB = (T_1, T_2, ..., T_x), an ordered set of non-auxiliary trees. FRONT is a subset of TB. W = {W_1, W_2 ..., W_n }, the input string. L = {L_1, L_2 ..., L_y } an ordered set of words called lexicon, M = The maximal number of adjoining desired in a tree.

Output: AS = Ambiguity set, i.e., a set of trees that match W

```
begin
1  CAT  ← CATEGORIES(W, L)
        // CATEGORIES returns a sequence or lexical categories
2  n  ←  LENGTH(W)  // The input length is assigned to n
3  SUBTB ← BINARY(TB, n)
        // BINARY returns a subset of TB containing trees of length  n
4  AS ←  MATCH(SUBTB, CAT)
        // MATCH returns a subset of  SUBTB that matches CAT
5  DYN ← PRUNE(FRONT, CAT)
        // PRUNE returns a subset of FRONT appropriate for CAT
6  ATB ← PRUNE(AT, CAT)
        // PRUNE returns a subset of AT
7    If  DYN and ATB are both non-empty then do:
     begin
8      LOOP:
9        TEMP  ← ( )            // TEMP is initially empty
10       Create a process j for each tree in DYN
11       In each process j :
11.1             Create a processes h for each tree in ATB
11.2             In each process h:
11.2.1                   CT  ← ADJOIN(AT_h, DYNj)
11.2.2                   CTS ← MATCH(CT, CAT)
11.2.3                   AS  ← AS U CTS
11.2.4                   CT  ← CT - CTS
11.2.5                   TEMP ← TEMP U REVISE(CT, CAT)
                                 // REVISE returns a subset or CT
12   Synchronize
13   DYN ← TEMP
14   if(DYN  != nil) then  Go to LOOP
        //  If DYN  is non-empty,  then  go to  LOOP
     end
15   return AS
end
```

For input strings of reasonable length, steps 8 through 14 are not executed. For these strings, step 4 determines the trees that match them and these trees are returned in step 15. Steps 8 through 14 are used for dynamic adjoin which is avoided as much as possible, because dynamic adjoin is not efficient. However, if dynamic adjoin is used, then it stops at step 14 when DYN becomes empty. If the dynamic adjoin is successfully avoided then the worst case computational time is proportional to the number of trees in the tree-bank. It is expected that an implementation of PARALLELPARSE will achieve almost linear speed up. This expectation is based on the assumption that the synchronization and inter-process communication overheads can be kept at a minimal level, since each process can perform its work independently.

4 Concluding Remarks

Performance of natural language parsing systems can be improved if the input string length is restricted to a reasonable number. If the input string length is not restricted, then the proposed parser will take additional time for processing. For any reasonable application, parsing natural language demands both computational efficiency and linguistic adequacy. Computational efficiency can be achieved by parallel processing of TAGs with pre-runtime processing of certain information, and heuristic strategies. Linguistic adequacy can be achieved by a TAG with its formalized structure generating mechanism and incremental development of language based structures. Syntactic structures or trees can be added incrementally to the tree-bank of the TAG parser anytime. Recent advances in semantic processing of TAGs suggest that TAGs present a viable alternative for application development using natural languages such as English [3].

Acknowledgement:

We are grateful to Juan Espana, Jose Contreras and many others for their help, comments and encouragements.

References

[1] Jurafsky, D. et al, *Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics and Speech Recognition*, Prentice Hall, 2000.

[2] Allen, J. *Natural Language Understanding* (2nd Ed.), Addison-Wesley, 1995.

[3] Abeillé, A., Rambow, O., *Tree Adjoining Grammars*, University of Chicago Press, 2001.

[4] Joshi, A. K., Levy, L. S. "Tree Adjoining Grammars", *Journal of the Computer and System Sciences*, 1975, 10: 136-163.

[5] Joshi, A. K. "Tree Adjoining Grammars: How Much Context Sensitivity is Required to Provide Reasonable Structural Descriptions?", 1985, in Dowty et al (eds)., *Natural Language Parsing*.

[6] Vijay-Shankar, K, Joshi, A. K. "Some Computational Properties of Tree Adjoining Grammars", *Proceedings of the 23rd Annual Meeting of the ACL*, 1985, 82-93.

[7] Dey, P., Bryant, B., Takaoka, T. "Lexical Ambiguity in Tree Adjoining Grammars", *Information Processing Letters*, 34, 1990, 65-69

[8] Schabes, Yves, Waters, Richard C. : Tree Insertion Grammar: Cubic-Time, Parsable Formalism that Lexicalizes Context-Free Grammar without Changing the Trees Produced. Computational Linguistics 21(4), 1995, : 479-513

[9] Joshi, Aravind K., Schabes, Yves: Tree-adjoining grammars and lexicalized grammars.