1

On Theoretical and Practical Complexity of TAG Parsers

Carlos Gómez-Rodríguez, Miguel A. Alonso,
Manuel Vilares

Abstract

We present a system allowing the automatic transformation of parsing schemata to efficient executable implementations of their corresponding algorithms. This system can be used to easily prototype, test and compare different parsing algorithms. In this work, it has been used to generate several different parsers for Context Free Grammars and Tree Adjoining Grammars. By comparing their performance on different sized, artificially generated grammars, we can measure their empirical computational complexity. This allows us to evaluate the overhead caused by using Tree Adjoining Grammars to parse context-free languages, and the influence of string and grammar size on Tree Adjoining Grammars parsing.

Keywords Parsing Schemata, Computational Complexity, Tree Adjoining Grammars, Context Free Grammars

1.1 Introduction

The process of parsing, by which we obtain the structure of a sentence as a result of the application of grammatical rules, is a highly relevant step in the automatic analysis of natural languages. In the last decades, various parsing algorithms have been developed to accomplish this task. Although all of these algorithms essentially share the common goal of generating a tree structure describing the input sentence by means of a grammar, the approaches used to attain this result vary greatly between algorithms, so that different parsing algorithms are best suited to different situations.

FG-2006.
Organizing Committee:, Paola Monachesi, Gerald Penn, Giorgio Satta, Shuly Wintner.

Parsing schemata, introduced in (Sikkel, 1997), provide a formal, simple and uniform way to describe, analyze and compare different parsing algorithms. The notion of a parsing schema comes from considering parsing as a deduction process which generates intermediate results called *items*. An initial set of items is directly obtained from the input sentence, and the parsing process consists of the application of inference rules (called *deductive steps*) which produce new items from existing ones. Each item contains a piece of information about the sentence's structure, and a successful parsing process will produce at least one *final item* containing a full parse tree for the sentence or guaranteeing its existence.

Almost all known parsing algorithms may be described by a parsing schema (non-constructive parsers, such as those based on neural networks, are exceptions). This is done by identifying the kinds of items that are used by a given algorithm, defining a set of inference rules describing the legal ways of obtaining new items, and specifying the set of final items.

As an example, we introduce a CYK-based algorithm (Vijay-Shanker and Joshi 1985) for Tree Adjoining Grammars (TAG) (Joshi and Schabes 1997). Given a tree adjoining grammar $G = (V_T, V_N, S, I, A)$[1] and a sentence of length n which we denote by $a_1\ a_2\ \ldots\ a_n$[2], we denote by $P(G)$ the set of productions $\{N^\gamma \rightarrow N_1^\gamma N_2^\gamma \ldots N_r^\gamma\}$ such that N^γ is an inner node of a tree $\gamma \in (I \cup A)$, and $N_1^\gamma N_2^\gamma \ldots N_r^\gamma$ is the ordered sequence of direct children of N^γ.

The parsing schema for the TAG CYK-based algorithm is a function that maps such a grammar G to a deduction system whose domain is the set of items

$$\{[N^\gamma, i, j, p, q, adj]\}$$

verifying that N^γ is a tree node in an elementary tree $\gamma \in (I \cup A)$, i and j $(0 \leq i \leq j)$ are string positions, p and q may be undefined or instantiated to positions $i \leq p \leq q \leq j$ (the latter only when $\gamma \in A$), and $adj \in \{true, false\}$ indicates whether an adjunction has been performed on node N^γ.

The positions i and j indicate that a substring $a_{i+1} \ldots a_j$ of the string is being recognized, and positions p and q denote the substring dominated by γ's foot node. The final item set would be

[1]Where V_T denotes the set of terminal symbols, V_N the set of nonterminal symbols, S the axiom, I the set of initial trees and A the set of auxiliary trees.

[2]From now on, we will follow the usual conventions by which nonterminal symbols are represented by uppercase letters (A, $B \ldots$), and terminals by lowercase letters (a, $b \ldots$). Greek letters (α, β...) will be used to represent trees, N^γ a node in the tree γ, and R^γ the root node of the tree γ.

$$\{[R^\alpha, 0, n, -, -, adj] \mid \alpha \in I\}$$

for the presence of such an item would indicate that there exists a valid parse tree with yield $a_1\ a_2\ \ldots\ a_n$ and rooted at R^α, the root of an initial tree; and therefore there exists a complete parse tree for the sentence.

A *deductive step* $\frac{\eta_1 \ldots \eta_m}{\xi}\ \Phi$ allows us to infer the item specified by its consequent ξ from those in its antecedents $\eta_1 \ldots \eta_m$. *Side conditions* (Φ) specify the valid values for the variables appearing in the antecedents and consequent, and may refer to grammar rules or specify other constraints that must be verified in order to infer the consequent. An example of one of the schema's deductive steps would be the following, where the operation $p \cup p'$ returns p if p is defined, and p' otherwise:

$$\text{CYK Binary: } \frac{\begin{array}{c}[O_1^\gamma, i, j', p, q, adj1] \\ [O_2^\gamma, j', j, p', q', adj2]\end{array}}{[M^\gamma, i, j, p \cup p', q \cup q', false]}\ M^\gamma \rightarrow O_1^\gamma O_2^\gamma \in P(G)$$

This deductive step represents the bottom-up parsing operation which joins two subtrees into one, and is analogous to one of the deductive steps of the CYK parser for Context-Free Grammars (Kasami 1965, Younger 1967). The full TAG CYK parsing schema has six deductive steps (or seven, if we work with TAGs supporting the substitution operation) and can be found at (Alonso et al., 1999). However, this sample deductive step is an example of how parsing schemata convey the fundamental semantics of parsing algorithms in simple, high-level descriptions. A parsing schema defines a set of possible intermediate results and allowed operations on them, but doesn't specify data structures for storing the results or an order for the operations to be executed.

1.2 Compilation of parsing schemata

Their simplicity and abstraction of low-level details makes parsing schemata very useful, allowing us to define parsers in a simple and straightforward way. Comparing parsers, or considering aspects such as their correction and completeness or their computational complexity, also becomes easier if we think in terms of schemata.

However, the problem with parsing schemata is that, although they are very useful when designing and comparing parsers with pencil and paper, they cannot be executed directly in a computer. In order to execute the parsers and analyze their results and performance they must be implemented in a programming language, making it necessary to abandon the high abstraction level and focus on the implementation details in order to obtain a functional and efficient implementation.

In order to bridge this gap between theory and practice, we have designed and implemented a compiler able to automatically transform parsing schemata into efficient Java implementations of their corresponding algorithms. The input to this system is a simple and declarative representation of a parsing schema, which is practically equal to the formal notation that we used previously. For example, this is the CYK deductive step we have seen as an example in a format readable by our compiler:

```
@step CYKBinary
[ Node1 , i , j' , p , q , adj1 ]
[ Node2 , j' , j , p' , q' , adj2 ]
----------------------------------------------------- Node3 -> Node1 Node2
[ Node3 , i , j , Union(p;p') , Union(q;q') , false ]
```

The parsing schemata compilation technique behind our system is based on the following fundamental ideas:

- Each deductive step is compiled to a Java class containing code to match and search for antecedent items and generate the corresponding conclusions from the consequent.
- The generated implementation will create an instance of this class for each possible set of values satisfying the side conditions that refer to production rules. For example, a distinct instance of the CYK Binary step will be created for each grammar rule of the form $M^\gamma \rightarrow O_1^\gamma O_2^\gamma \in P(G)$, as specified in the step's side condition.
- The step instances are coordinated by a deductive parsing engine, as the one described in (Shieber et al., 1995). This algorithm ensures a sound and complete deduction process, guaranteeing that all items that can be generated from the initial items will be obtained. It is a generic, schema-independent algorithm, so its implementation is the same for any parsing schema. The engine works with the set of all items that have been generated and an *agenda*, implemented as a queue, holding the items we have not yet tried to trigger new deductions with.
- In order to attain efficiency, an automatic analysis of the schema is performed in order to create indexes allowing fast access to items. Two kinds of index structures are generated: *existence indexes* are used by the parsing engine to check whether a given item exists in the item set, while *search indexes* are used to search for all items conforming to a given specification. As each different parsing schema needs to perform different searches for antecedent items, the index

structures that we generate are schema-specific. Each deductive step is analyzed in order to keep track of which variables will be instantiated to a concrete value when a search must be performed. This information is known at schema compilation time and allows us to create indexes by the elements corresponding to instantiated variables. In this way, we guarantee constant-time access to items so that the computational complexity of our generated implementations is never above the theoretical complexity of the parsing algorithms.

- *Deductive step indexes* are also generated to provide efficient access to the set of deductive step instances which can be applicable to a given item. Step instances that are known not to match the item are filtered out by these indexes, so less time is spent on unsuccessful item matching.
- Since parsing schemata have an open notation, for any mathematical object can potentially appear inside items, the system includes an extensibility mechanism which can be used to define new kinds of objects to use in schemata. The code generator can deal with these user-defined objects as long as some simple and well-defined guidelines are followed in their specification.

A more detailed description of this system, including a more thorough explanation of automatic index generation, can be found at (Gómez-Rodríguez et al., 2006b).

1.3 Parsing natural language CFG's

Although our main focus in this paper is on performance of TAG parsing algorithms, we will briefly outline the results of some experiments on Context-Free Grammars (CFG), described in further detail in (Gómez-Rodríguez et al., 2006b), in order to be able to contrast TAG and CFG parsing.

Our compilation technique was used to generate parsers for the CYK (Kasami 1965, Younger 1967), Earley (Earley 1970) and Left-Corner (Rosenkrantz and Lewis II 1970) algorithms for context-free grammars, and these parsers were tested on automatically-generated sentences from three different natural language grammars: Susanne (Sampson 1994), Alvey (Carroll 1993) and Deltra (Schoorl and Belder 1990). The runtimes for all the algorithms and grammars showed an empirical computational complexity far below the theoretical worst-case bound of $O(n^3)$, where n denotes the length of the input string. In the case of the Susanne grammar, the measurements were close to being linear with string size. In the other grammars, the runtimes grew faster, approximately $O(n^2)$, still far below the cubic worst-case bound.

Another interesting result was that the CYK algorithm performed better than the Earley-type algorithms in all cases, despite being generally considered slower. The reason is that these considerations are based on time complexity relative to string length, and do not take into account time complexity relative to grammar size, which is $O(|P|)$ for CYK and $O(|P|)^2$ for the Earley-type algorithms. This factor is not very important when working with small grammars, such as the ones used for programming languages, but it becomes fundamental when we work with natural language grammars, where we use thousands of rules (more than 17,000 in the case of Susanne) to parse relatively small sentences. When comparing the results from the three context-free grammars, we observed that the performance gap between CYK and Earley was bigger when working with larger grammars.

1.4 Parsing artificial TAG's

In this section, we make a comparison of four different TAG parsing algorithms: the CYK-based algorithm used as an example in section 1.1, an Earley-based algorithm without the valid prefix property (described in Alonso et al. 1999, inspired in the one in Schabes 1994), an Earley-based algorithm with the valid prefix property (Alonso et al. 1999) and Nederhof's algorithm (Nederhof 1999). These parsers are compared on artificially generated grammars, by using our schema compiler to generate implementations and measuring their execution times with several grammars and sentences.

Note that the advantage of using artificially generated grammars is that we can easily see the influence of grammar size on performance. If we test the algorithms on grammars from real-life natural language corpora, as we did with the CFG parsers, we don't get a very precise idea of how the size of the grammar affects performance. Since our experience with CFG's showed this to be an important factor, and existing TAG parser performance comparisons (e.g. Díaz and Alonso 2000) work with a fixed (and small) grammar, we decided to use artificial grammars in order to be able to adjust both string size and grammar size in our experiments and see the influence of both factors.

For this purpose, given an integer $k > 0$, we define the tree-adjoining grammar G_k to be the grammar $G_k = (V_T, V_N, S, I, A)$ where $V_T = \{a_j | 0 \leq j \leq k\}$, $V_N = \{S, B\}$, and

$I = \{S(B(a_0))\}$[3],

$A = \{B(B(B^* \; a_j)) | 1 \leq j \leq k\}$.

[3] Where trees are written in bracketed notation, and * is used to denote the foot node.

Therefore, for a given k, G_k is a grammar with one initial tree and k auxiliary trees, which parses a language over an alphabet with $k+1$ terminal symbols. The actual language defined by G_k is the regular language $L_k = a_0(a_1|a_2|..|a_k)^*$. [4] We shall note that although the languages L_k are trivial, the grammars G_k are built in such a way that any of the auxiliary trees may adjoin into any other. Therefore these grammars are suitable if we want to make an empyrical analysis of worst-case complexity.

Table 1 shows the execution time in milliseconds[5] of four TAG parsers with the grammars G_k, for different values of string length (n) and grammar size (k).

From this results, we can observe that both factors (string length and grammar size) have an influence on runtime, and they interact between themselves: the growth rates with respect to one factor are influenced by the other factor, so it is hard to give precise estimates of empirical computational complexity. However, we can get rough estimates by focusing on cases where one of the factors takes high values and the other one takes low values (since in these cases the constant factors affecting complexity will be smaller) and test them by checking whether the sequence $T(n,k)/f(n)$ seems to converge to a positive constant for each fixed k (if $f(n)$ is an estimation of complexity with respect to string length) or whether $T(n,k)/f(k)$ seems to converge to a positive constant for each fixed n (if $f(k)$ is an estimation of complexity with respect to grammar size).

By applying these principles, we find that the empirical time complexity with respect to string length is in the range between $O(n^{2.8})$ and $O(n^3)$ for the CYK-based and Nederhof algorithms, and between $O(n^{2.6})$ and $O(n^3)$ for the Earley-based algorithms with and without the valid prefix property (VPP). Therefore, the practical time complexity we obtain is far below the theoretical worst-case bounds for these algorithms, which are $O(n^6)$ (except for the Earley-based algorithm with the VPP, which is $O(n^7)$).

Although for space reasons we don't include tables with the number of items generated in each case, our results show that the empirical

[4] Also, it is easy to prove that the grammar G_k is one of the minimal tree adjoining grammars (in terms of number of trees) whose associated language is L_k. Note that we need at least a tree containing a_0 as its only terminal in order to parse the sentence a_0, and for each $1 \leq i \leq k$, we need at least a tree containing a_i and no other a_j ($j > 0$) in order to parse the sentence $a_0 a_i$. Therefore, any TAG for the language L_k must have at least $k+1$ elementary trees.

[5] The machine used for all the tests was an Intel Pentium 4 3.40 GHz, with 1 GB RAM and Sun Java Hotspot virtual machine (version 1.4.2_01-b06) running on Windows XP.

Runtimes in ms: Earley-based without the VPP					
String Size (n)	Grammar Size (k)				
	1	8	64	512	4096
2	∼0	16	15	**1,156**	**109,843**
4	∼0	31	63	**2,578**	**256,094**
8	16	31	**172**	**6,891**	**589,578**
16	31	172	**625**	**18,735**	**1,508,609**
32	110	609	**3,219**	**69,406**	
64	485	2,953	**22,453**	**289,984**	
128	2,031	**13,875**	**234,594**		
256	10,000	**101,219**			
512	61,266				

Runtimes in ms: CYK-based					
String Size (n)	Grammar Size (k)				
	1	8	64	512	4096
2	∼0	∼0	16	1,344	125,750
4	∼0	∼0	63	4,109	290,187
8	16	31	234	15,891	777,968
16	**15**	**62**	782	44,188	2,247,156
32	**94**	**312**	3,781	170,609	
64	**266**	**2,063**	25,094	550,016	
128	**1,187**	14,516	269,047		
256	**6,781**	108,297			
512	**52,000**				

Runtimes in ms: Nederhof's Algorithm					
String Size (n)	Grammar Size (k)				
	1	8	64	512	4096
2	∼0	∼0	47	1,875	151,532
4	∼0	15	187	4,563	390,468
8	15	31	469	12,531	998,594
16	46	188	1,500	40,093	2,579,578
32	219	953	6,235	157,063	
64	1,078	4,735	35,860	620,047	
128	5,703	25,703	302,766		
256	37,125	159,609			
512	291,141				

Runtimes in ms: Earley-based with the VPP					
String Size (n)	Grammar Size (k)				
	1	8	64	512	4096
2	∼0	∼0	31	1,937	194,047
4	∼0	16	78	4,078	453,203
8	15	31	234	10,922	781,141
16	31	188	875	27,125	1,787,140
32	125	750	4,141	98,829	
64	578	3,547	28,640	350,218	
128	2,453	20,766	264,500		
256	12,187	122,797			
512	74,046				

TABLE 1 Execution times of four different TAG parsers for artificially-generated grammars G_k. Best results are shown in boldface.

space complexity with respect to string length is approximately $O(n^2)$ for all the algorithms, also far below the worst-case bounds ($O(n^4)$ and $O(n^5)$).

With respect to the size of the grammar, we obtain a time complexity of approximately $O(|I \cup A|^2)$ for all the algorithms. This matches the theoretical worst-case bound, which is $O(|I \cup A|^2)$ due to the adjunction steps, which work with pairs of trees. In the case of our artificially generated grammar, any auxiliary tree can adjoin into any other, so it's logical that our times grow quadratically. Note, however, that real-life grammars such as the XTAG English grammar (XTAG Research Group 2001) have relatively few different nonterminals in relation to their amount of trees, so many pairs of trees are susceptible of adjunction and we can't expect their behavior to be much better than this.

Space complexity with respect to grammar size is approximately $O(|I \cup A|)$ for all the algorithms. This is an expected result, since each generated item is associated to a given tree node.

Practical applications of TAG in natural language processing usually fall in the range of values for n and k covered in our experiments (grammars with hundreds or a few thousands of trees are used to parse sentences of several dozens of words). Within these ranges, both string length and grammar size take significant values and have an important influence on execution times, as we can see from the results in the tables. This leads us to note that traditional complexity analysis based on a single factor (string length or grammar size) can be misleading for practical applications, since it can lead us to an incomplete idea of real complexity. For example, if we are working with a grammar with thousands of trees, the size of the grammar is the most influential factor, and the use of filtering techniques (Schabes and Joshi 1991) to reduce the amount of trees used in parsing is essential in order to achieve good performance. The influence of string length in these cases, on the other hand, is mitigated by the huge constant factors related to grammar size. For instance, in the times shown in the tables for the grammar G_{4096}, we can see that parsing times are multiplied by a factor less than 3 when the length of the input string is duplicated, although the rest of the results have lead us to conclude that the practical asymptotic complexity with respect to string length is at least $O(n^{2.6})$. These interactions between both factors must be taken into account when analyzing performance in terms of computational complexity.

Earley-based algorithms achieve better execution times than the CYK-based algorithm for large grammars, although they are worse for small grammars. This contrasts with the results for context-free grammars, where CYK works better for large grammars: when working with

CFG's, CYK has a better computational complexity than Earley (linear with respect to grammar size, see section 1.3), but the TAG variant of the CYK algorithm is quadratic with respect to grammar size and does not have this advantage.

CYK generates fewer items than the Earley-based algorithms when working with large grammars and short strings, and the opposite happens when working with small grammars and long strings.

The Earley-based algorithm with the VPP generates the same number of items than the one without this property, and has worse execution times. The reason is that no partial parses violating this property are generated by any of both algorithms in the particular case of this grammar, so guaranteeing the valid prefix property does not prevent any items from being generated. Therefore, the fact that the variant without the VPP works better in this particular case cannot be extrapolated to other grammars. However, the differences in times between these two algorithms illustrates the overhead caused by the extra checks needed to guarantee the valid prefix property in a particularly bad case.

Nederhof's algorithm has slower execution times than the other Earley variants. Despite the fact that Nederhof's algorithm is an improvement over the other Earley-based algorithm with the VPP in terms of computational complexity, the extra deductive steps it contains makes it slower in practice.

1.5 Parsing the XTAG English grammar

In order to complement our performance comparison of the four algorithms on artificial grammars, we have also studied the behavior of the parsers when working with a real-life, large-scale TAG: the XTAG English grammar (XTAG Research Group 2001).

The obtained execution times are in the ranges that we could expect given the artificial grammar results, i.e. they approximately match the times in the tables for the corresponding grammar sizes and input string lengths. The most noticeable difference is that the Earley-like algorithm verifying the valid prefix property generates fewer items that the variant without the VPP in the XTAG grammar, and this causes its runtimes to be faster. But this difference is not surprising, as explained in the previous section.

Note that, as the XTAG English grammar has over a thousand elementary trees, execution times are very large (over 100 seconds) when working with the full grammar, even with short sentences. However, when a tree selection filter is applied in order to work with only a subset of the grammar in function of the input string, the grammar size is reduced to one or two hundred trees and our parsers process short sentences in less than 5 seconds. Sarkar's XTAG distribution parser written in C[6] applies further filtering techniques and has specific optimizations for this grammar, obtaining better times for the XTAG than our generic parsers.

Table 2 contains a summary of the execution times obtained by our parsers for some sample sentences from the XTAG distribution. Note that the generated implementations used for these executions apply the mentioned tree filtering technique, so that the effective grammar size is different for each sentence, hence the high variability in execution times. More detailed information on these experiments with the XTAG English grammar can be found at (Gómez-Rodríguez et al., 2006a).

Sentence	Runtimes in milliseconds			
	CYK	Ear. no VPP	Ear. VPP	Neder.
He was a cow	2985	**750**	**750**	2719
He loved himself	3109	1562	**1219**	6421
Go to your room	4078	1547	**1406**	6828
He is a real man	4266	1563	**1407**	4703
He was a real man	4234	1921	**1421**	4766
Who was at the door	4485	1813	**1562**	7782
He loved all cows	5469	2359	**2344**	11469
He called up her	7828	4906	**3563**	15532
He wanted to go to the city	10047	4422	**4016**	18969
That woman in the city contributed to this article	13641	**6515**	7172	31828
That people are not really amateurs at intelectual duelling	16500	**7781**	15235	56265
The index is intended to measure future economic performance	16875	17109	**9985**	39132
They expect him to cut costs throughout the organization	25859	**12000**	20828	63641
He will continue to place a huge burden on the city workers	54578	**35829**	57422	178875
He could have been simply being a jerk	**62157**	113532	109062	133515
A few fast food outlets are giving it a try	**269187**	3122860	3315359	

TABLE 2 Runtimes obtained by applying different XTAG parsers to several sentences. Best results for each sentence are shown in boldface.

[6] Downloadable at: ftp://ftp.cis.upenn.edu/pub/xtag/lem/

1.6 Overhead of TAG parsing over CFG parsing

The languages L_k that we parsed in section 1.4 were regular languages, so in practice we don't need tree adjoining grammars to parse them, although it was convenient to use them in our comparison. This can lead us to wonder how large is the overhead caused by using the TAG formalism to parse context-free languages.

Given the regular language $L_k = a_0(a_1|a_2|..|a_k)^*$, a context-free grammar that parses it is $G'_k = (N, \Sigma, P, S)$ with $N = \{S\}$ and

$$P = \{S \rightarrow a_0\} \cup \{S \rightarrow Sa_i | 1 \leq i \leq k\}$$

This grammar minimizes the number of rules needed to parse L_k (k+1 rules), but has left recursion. If we want to eliminate left recursion, we can use the grammar $G''_k = (N, \Sigma, P, S)$ with $N = \{S, A\}$ and

$$P = \{S \rightarrow a_0 A\} \cup \{A \rightarrow a_i A | 1 \leq i \leq k\} \cup \{A \rightarrow \epsilon\}$$

which has $k + 2$ production rules.

The number of items generated by the Earley algorithm for context-free grammars when parsing a sentence of length n from the language L_k by using the grammar G'_k is $(k+2)n$. In the case of the grammar G''_k, the same algorithm generates $(k+4)n + \frac{n(n-1)}{2} + 1$ items. In both cases the amount of items generated is linear with respect to grammar size, as in TAG parsers. With respect to string size, the amount of items is $O(n)$ for G'_k and $O(n^2)$ for G''_k, and it was approximately $O(n^2)$ for the TAG G_k. Note, however, that the constant factors behind complexity are much greater when working with G_k than with G''_k, and this reflects on the actual number of items generated (for example, the Earley algorithm generates 16,833 items when working with G''_{64} and a string of length $n = 128$, while the TAG variant of Earley without the valid prefix property generated 1,152,834 items).

The execution times for both algorithms appear in table 3. From the obtained times, we can deduce that the empirical time complexity is linear with respect to string length and quadratic with respect to grammar size in the case of G'_k; and quadratic with respect to string length and linear with respect to grammar size in the case of G''_k. So this example shows that, when parsing a context-free language using a tree-adjoining grammar, we get an overhead both in constant factors (more complex items, more deductive steps, etc.) and in asymptotic behavior, so actual execution times can be several orders of magnitude larger. Note that the way grammars are designed also has an influence, but our tree adjoining grammars G_k are the simplest TAGs able to parse the languages L_k by using adjunction (an alternative would be to write a grammar using the substitution operation to combine trees).

n	Grammar Size (k), grammar G'_k				
	1	8	64	512	4096
2	∼0	∼0	∼0	31	2,062
4	∼0	∼0	∼0	62	4,110
8	∼0	∼0	∼0	125	8,265
16	∼0	∼0	∼0	217	15,390
32	∼0	∼0	15	563	29,344
64	∼0	∼0	31	1,062	61,875
128	∼0	∼0	109	2,083	122,875
256	∼0	15	188	4,266	236,688
512	15	31	328	8,406	484,859
n	Grammar Size (k), grammar G''_k				
	1	8	64	512	4096
2	∼0	∼0	∼0	∼0	47
4	∼0	∼0	∼0	15	94
8	∼0	∼0	∼0	16	203
16	∼0	∼0	∼0	46	688
32	∼0	∼0	15	203	1,735
64	31	31	93	516	4,812
128	156	156	328	1,500	13,406
256	484	547	984	5,078	45,172
512	1,765	2,047	3,734	18,078	

TABLE 3 Runtimes obtained by applying the Earley parser for context-free grammars to sentences in L_k.

1.7 Conclusions

In this paper, we have presented a system that compiles parsing schemata to executable implementations of parsers, and used it to evaluate the performance of several TAG parsing algorithms, establishing comparisons both between themselves and with CFG parsers.

The results show that both string length and grammar size can be important factors in performance, and the interactions between them sometimes make their influence hard to quantify. The influence of string length in practical cases is usually below the theoretical worst-case bounds (between $O(n)$ and $O(n^2)$ in our tests for CFG's, and slightly below $O(n^3)$ for TAG's). Grammar size becomes the dominating factor in large TAG's, making tree filtering techniques advisable in order to achieve faster execution times.

Using TAG's to parse context-free languages causes an overhead both in constant factors and in practical computational complexity, thus increasing execution times by several orders of magnitude with respect to CFG parsing.

Acknowledgements

The work reported in this article has been supported in part by Ministerio de Educación y Ciencia and FEDER (TIN2004-07246-C03-01, TIN2004-07246-C03-02), Xunta de Galicia (PGIDIT05PXIC30501PN, PGIDIT05PXIC10501PN, PGIDIT05SIN044E and PGIDIT05SIN059E), and Programa de becas FPU (Ministerio de Educación y Ciencia).

References

Alonso, Miguel A., David Cabrero, Eric de la Clergerie, and Manuel Vilares. 1999. Tabular algorithms for TAG parsing. In *Proc. of EACL'99, Ninth Conference of the European Chapter of the Association for Computational Linguistics*, pages 150–157. ACL, Bergen, Norway.

Carroll, J. 1993. Practical unification-based parsing of natural language. PhD thesis. Tech. Rep. 314, Computer Laboratory, University of Cambridge, Cambridge, UK.

Díaz, Víctor J. and Miguel A. Alonso. 2000. Comparing tabular parsers for tree adjoining grammars. In D. S. Warren, M. Vilares, L. Rodríguez Liñares, and M. A. Alonso, eds., *Proc. of Tabulation in Parsing and Deduction (TAPD 2000)*, pages 91–100. Vigo, Spain.

Earley, J. 1970. An efficient context-free parsing algorithm. *Communications of the ACM* 13(2):94–102.

Gómez-Rodríguez, Carlos, Miguel A. Alonso, and Manuel Vilares. 2006a. Generating XTAG parsers from algebraic specifications. In *Proc. of TAG+8, the Eighth International Workshop on Tree Adjoining Grammar and Related Formalisms*. Sydney, Australia.

Gómez-Rodríguez, Carlos, Jesús Vilares, and Miguel A. Alonso. 2006b. Automatic generation of natural language parsers from declarative specifications. In *Proc. of STAIRS 2006*. Riva del Garda, Italy. Long version available at http://www.grupocole.org/GomVilAlo2006a_long.pdf.

Joshi, Aravind K. and Yves Schabes. 1997. Tree-adjoining grammars. In G. Rozenberg and A. Salomaa, eds., *Handbook of Formal Languages. Vol 3: Beyond Words*, chap. 2, pages 69–123. Berlin/Heidelberg/New York: Springer-Verlag.

Kasami, T. 1965. An efficient recognition and syntax algorithm for context-free languages. Scientific Report AFCRL-65-758, Air Force Cambridge Research Lab., Bedford, Massachussetts.

Nederhof, Mark-Jan. 1999. The computational complexity of the correct-prefix property for TAGs. *Computational Linguistics* 25(3):345–360.

Rosenkrantz, D. J. and P. M. Lewis II. 1970. Deterministic Left Corner parsing. In *Conference Record of 1970 Eleventh Annual Meeting on Switching and Automata Theory*, pages 139–152. IEEE, Santa Monica, CA, USA.

Sampson, G. 1994. The Susanne corpus, release 3.

Schabes, Yves. 1994. Left to right parsing of lexicalized tree-adjoining grammars. *Computational Intelligence* 10(4):506–515.

Schabes, Yves and Aravind K. Joshi. 1991. Parsing with lexicalized tree adjoining grammar. In M. Tomita, ed., *Current Issues in Parsing Technologies*, chap. 3, pages 25–47. Norwell, MA, USA: Kluwer Academic Publishers. ISBN 0-7923-9131-4.

Schoorl, J. J. and S. Belder. 1990. Computational linguistics at Delft: A status report, Report WTM/TT 90–09.

Shieber, Stuart M., Yves Schabes, and Fernando C. N. Pereira. 1995. Principles and implementation of deductive parsing. *Journal of Logic Programming* 24(1–2):3–36.

Sikkel, Klaas. 1997. *Parsing Schemata — A Framework for Specification and Analysis of Parsing Algorithms*. Texts in Theoretical Computer Science — An EATCS Series. Berlin/Heidelberg/New York: Springer-Verlag. ISBN 3-540-61650-0.

Vijay-Shanker, K. and Aravind K. Joshi. 1985. Some computational properties of tree adjoining grammars. In *23rd Annual Meeting of the Association for Computational Linguistics*, pages 82–93. ACL, Chicago, IL, USA.

XTAG Research Group. 2001. A lexicalized tree adjoining grammar for English. Tech. Rep. IRCS-01-03, IRCS, University of Pennsylvania.

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