A Modified Earley Parser for Huge Natural Language Grammars

Sinan Polat, Merve Selcuk-Simsek, and Ilyas Cicekli

Department of Computer Engineering
Hacettepe University
Ankara, Turkey
spolat@aselsan.com.tr {merveselcuk, ilyas}@cs.hacettepe.edu.tr

Abstract. For almost a half century Earley parser has been used in the parsing of context-free grammars and it is considered as a touch-stone algorithm in the history of parsing algorithms. On the other hand, it is also known for being expensive from its time requirement and memory usage perspectives. For huge context-free grammars, its performance is not good since its time complexity also depends on the number of rules in the grammar. The time complexity of the original Earley parser is $O(R^2N^3)$ where N is the string length, and R is the number of rules. In this paper, we aim to improve time and memory usage performances of Earley parser for grammars with a large number of rules. In our approach, we prefer radix tree representation for rules instead of list representation as in original Earley parser. We have tested our algorithm using different number of rule sets up to 200,000 which are all learned by an example-based machine translation system. According to our evaluation results, our modified parser has a time bound of $O(log(R)N^3)$, and it has 20% less memory usage regarding the original Earley parser.

Keywords: earley parser, computational linguistics, natural language processing

1 Introduction

Grammar parsers first revealed with programming languages. Until Earley's algorithm, the previous algorithms had been able only to parse unambiguous grammars, or subsets of context-free grammars. Revealing his general context-free grammar parser, Earley made a breakthrough. His algorithm became prior to others because it was able to parse left-recursive grammars, handle empty strings in grammars, and work in both ways as top-down and bottom-up. We analyze the details of his algorithm in Section 2.1. Although Earley parser looks good enough to parse every context-free grammar, an attempt to parse a natural language's grammar has not occurred even after years since his algorithm. Some only tried to parse a partial grammar. While parsing a huge grammar with thousands of rules, Earley's algorithm is ineffective due to following reasons:

– It is not good in memory usage:

  - Grouping the similar rules is not a part of the algorithm. The same rules may be stored over and over again unnecessarily resulting to consume the memory.

- It is slow in that it may take even days to parse with a big grammar:
  - The algorithm's mechanism for handling empty string rules in a grammar is not an ideal solution because it is too costly. Earley's solution required an extra data structure that needs to be dynamically-updated for this step.
  - Earley used linked-lists as data structures in his work. In linked-lists, time-complexity of getting an item equals to $O(n)$, where n is the number of total item count in the list. For such an expensive algorithm considering time bound, $O(n)$ is really huge. Our plan is to store hundreds of thousands of rules in our data structure. Considering this it would not be plausible to prefer lists here.

The performance evaluation of the Earley's algorithm is handled in Section 2.2.

As seen from the list above to parse a huge grammar, a modified Earley parser is needed. We put the choice on a regular basis:

- We do not use look-ahead tables in our algorithm to get performance. The details about not using look-ahead is examined in Section 3.1.
- For empty right-hand sides (empty strings) we prefer to use a slightly different version of Aycock and Horspool's well studied work[1] which is explained deeply in Section 3.2.
- As a data structure we prefer to use a modified radix tree over a list. All rules are carried in the tree, so we believe we build an easily comprehensible, memory efficient, and less expensive in time-complexity system as explained in Section 3.3.
- Two natural language grammars are used for evaluation: Turkish, and English. The rules that we use to parse are all coming from an example-based machine translation system [2, 3]. A variety of rules obtained from different data sets with example-based machine translation are parsed in tests, and we get encouraging results.

Throughout the paper, we assume that the reader is familiar with the basic information of language theory, e.g. context-free grammars (CFG), and context-free languages (CFL). The necessary information can be found in [4, 5]. Paper's flow is as it follows. In Section 2, we briefly describe the standard Earley parser, and its performance. We give the details of our modified parser including radix tree usage in the rule representation in Section 3. Section 4 presents the evaluation of the modified Earley parser, and in Section 5 we examined the related work before us.

2 Earley Parser

2.1 Earley's Algorithm

Earley's algorithm is a dynamic programming algorithm and a chart parser which resembles Knuth's $LR(k)$ algorithm [6]. Given an input string $S_1, S_2, \ldots, S_n$ (n refers to the length of string), the algorithm scans it from left to right. After each symbol S_i is scanned, a state set E_i is generated for it. These sets are mostly called as *Earley sets*, so we named it as E. The state of the recognition process is monitored from state sets. On each set E_i, there are productions to show which rule of the grammar is scanned. Each production has one dot ($\bullet$) to mark the place that is recognized so far from the right-side of the production. Also, productions include a state reminder j to identify from which state this production is composed. Earley's algorithm has three main phases:

SCANNER. This operation takes place if there is a terminal after dot in a state, e.g. $[T \rightarrow \ldots \bullet a \ldots,\ j]$

PREDICTOR. This operation starts when there is a nonterminal after dot in the state, e.g. $[T \rightarrow \ldots \bullet P \ldots,\ j]$

COMPLETER. The COMPLETER is applicable if in a state the dot is at the end of the production, e.g. $[T \rightarrow \ldots aP \bullet,\ j]$

2.2 Performance Evaluation of the Standard Earley Parser

Earley's algorithm has a time bound of $O(N^3)$ for ambiguous grammars in worst-case where N is the number of characters in the input string [7]. Another important criterion to determine time complexity, however, is the size of the grammar, i.e. the number of rules in the grammar. Earley states that there is a constant C depending on the size of the grammar, and not depending on n [7]. This information leads to the time complexity proportional to CN^3.

Although Earley does not discuss the properties of the constant C in his work any further, our experiments in Fig. 1 shows that C results a cost of $O(R^2)$ where R is the number of rules in the grammar. That makes the overall complexity of Earley's algorithm $O(R^2N^3)$. In other words, when we double the size of the grammar, the time required to parse a string is multiplied by a factor of four. Our point here is for very large grammars, e.g. a natural language's grammar, using the standard version of Earley parser is not feasible because parsing operation would take very long time.

3 Modified Earley Parser

We examined Earley's parser from three points of view. First two points cover algorithmic approach, and third one is a structural approach. Our goal is both to accelerate the algorithm, and to use the memory more efficiently.

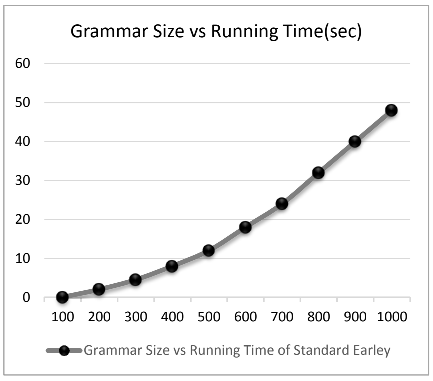


Fig. 1. The running time of the standard Earley parser relating to the number of rules in the grammar

3.1 Look-ahead

Look-ahead principle was one of the features Earley derived from Knuth's work on LR(k) grammars [6, 7]. Earley suggested using look-ahead on COMPLETER step of his algorithm [7]. In time there arose controversy about look-ahead's usage. A research conducted by Bouckaert et al. showed that look-ahead best fit to PREDICTOR part [8]. McLean and Horspool indicated look-ahead slowed down the Earley parser [9]. Later Aycock and Horspool stated look-ahead's necessity was controversial [10]. Because we seek for speed, we decide not to use to look-ahead which defines our parser as an $LR(0)$ parser.

3.2 Empty Sequence (ϵ) Rules

In formal language theory, the empty sequence is a sequence with length zero, and represented by ϵ. If a grammar includes rules with empty strings, parsing the grammar using Earley's algorithm is inefficient. Since Earley parser process items in a particular set of order, encountering an ϵ rule creates problems.

Consider $A \rightarrow \epsilon$ as a rule in the grammar. When A is called in state E_i, it remains without consuming any terminals in E_i. Its staying the same delivers problems, because at this situation COMPLETER will assume $i = k$. However, COMPLETER needs to evaluate the full set of E_k to parse the input correctly. Fortunately, this is a solvable problem. There was some suggestions to fix it including Earley himself [11, 7, 4, 12], yet being either so expensive in time-complexity, or not memory friendly. We prefer to apply Aycock and Horspool's method [1], this simple but effective method explained in Fig. 2.

Although the algorithm in Fig. 2 looks expensive especially considering its first part, it is not a complex operation. Aycock and Horspool didn't present a specific algorithm for this part in their work [1]. Our algorithm for handling the first part (determining all nullable nonterminals in the grammar), doesn't involve

i Determine all nullable nonterminals in grammar G, e.g. for a nonterminal A: $A \Rightarrow^* \epsilon$
ii Modify the algorithm of the PREDICTOR as follows:

```
for each item $[A \rightarrow \ldots \bullet\ B \ldots,\ j]$ in $E_i$
 for each rule $[B \rightarrow \beta]$
  add $[B \rightarrow \bullet\beta,\ i]$ to $E_i$
 if $B$ is nullable
  add $A \rightarrow \ldots\ B \bullet \ldots,\ j$ to $E_i$
```

Fig. 2. Aycock and Horspool's algorithm to deal with empty sequences

a loop checking every symbol in the grammar, both terminals, and nonterminals. Because terminals doesn't have a chance to be empty right-sided, we skip all terminals in a grammar while searching *epsilon* rules. This approach not only decreases the count of the loop, but also reduces the time-complexity of this part. Additionally, the first part is run only for once at the beginning of the flow as a preprocessing step.

3.3 Modified Radix Tree

In our application, we prefer to use radix trees which are slightly different forms of PATRICIA trees [13]. In fact, a radix tree is a special PATRICIA tree with a radix value of 2. Our reasons to use radix trees are:

– *Memory efficiency.* While working with so many rules and huge grammars, efficiently used memory becomes a priority. The space-complexity of a radix tree is $O(CN)$ in that C is the average length of rules, and N is the number of rules in a grammar. One of the best properties of radix trees is its ability to group similar rules. This is an important feature for both memory usage, and compactness of the application.
– *Fast prefix search.* Radix trees' search time-complexity is $O(N)$.
– *Low insertion complexity.* The insertion complexity of a radix tree is $O(\log(N))$. If it were more expensive than that, we may not be able to construct a structure for hundreds of thousands rules.

The evaluation proof of the radix tree's performance can be seen in Section 4. We do not cover the deletion of strings from the radix tree here since it is not used by our application. Note that the radix tree structure is generated only once, when the grammar rules are loaded to the parser. After the rules are loaded to the parser, no rule is deleted, so we do not need the deletion operation.

The developed EBMT (Example-Based Machine Translation) system [2, 3] learns translation rules from translation examples between English and Turkish as in Table 1. The learned translation rules can be seen as two natural language grammars. We use modified radix tree to store and benefit our rules learned with our EBMT system. Because we use modified Earley parser in a machine translation environment, radix tree comes in handy both parsing the examples, and matching them for appropriate templates.

Table 1. Learning phase of translation rules

Consider the following translation pairs, in this example there are similar parts in both languages (car and araba, respectively), and also there are differing parts (black-siyah, and red-kırmızı).

black car ⇔ siyah araba
red car ⇔ kırmızı araba

Our EBMT system replaces differing parts with variables to create translation template below:

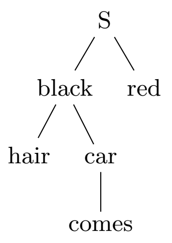
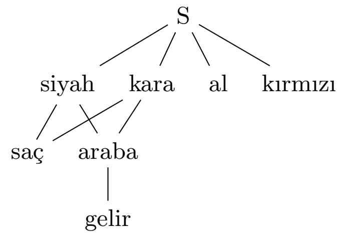
$$X^1 \text{ car} \Leftrightarrow Y^1 \text{ araba}$$
$$\text{if } X^1 \Leftrightarrow Y^1$$

Using translation template above, the EBMT system learns the templates below:

black ⇔ siyah
red ⇔ kırmızı

A sample context-free grammar and radix trees between English and Turkish languages are given in Table 2. The specified context-free grammar is composed of learned translation rules. The words between brackets (e.g. S→ black *[siyah]*) refers to the translation of the source sentence. Since we concern context-free grammars, the presentation involves only a single nonterminal (*S*) on the left-hand side of all rules. There might be terminals and/or nonterminals on the right-hand sides of the rules. No type information is used in Table 2 to make it more intelligible.

Table 2. Example context-free grammars and radix trees of English and Turkish phrases

English to Turkish	Turkish to English
S→ black [siyah]	S→ siyah [black]
S→ black [kara]	S→ kara [black]
S→ red [kırmızı]	S→ kırmızı [red]
S→ red [al]	S→ al [red]
S→ black hair [siyah saç]	S→ siyah saç [black hair]
S→ black hair [kara saç]	S→ kara saç [black hair]
S→ black car [siyah araba]	S→ siyah araba [black car]
S→ black car [kara araba]	S→ kara araba [black car]
S→ black car comes [siyah araba gelir]	S→ siyah araba gelir [black car comes]
S→ black car comes [kara araba gelir]	S→ kara araba gelir [black car comes]
	 S siyah kara al kırmızı saç araba gelir

Input: black car

$\boxed{\text{Chart 0}}$

State 0 = • black car

(1) &→ • S	(radix pointer 0)
	(starter rule)
(2) S→ • black	(radix pointer 1)
	(predicted from 1)
(3) S→ • red	(radix pointer 2)
	(predicted from 1)

$\boxed{\text{Chart 1}}$

State 1 = black • car

(4) S→ black •	(radix pointer 1)
	(scanner from chart 0 item 2)
(5) S→ • hair	(radix pointer 3)
	(completed from chart 0 item 2 by 4)
(6) S→ • car	(radix pointer 4)
	(completed from chart 0 item 2 by 4)

$\boxed{\text{Chart 2}}$

State 2 = black car •

(7) S→ car •	(radix pointer 4)
	(scanner from chart 1 item 6)
(8) S→ • comes	(radix pointer 5)
	(completed from chart 1 item 6)
(9) &→S •	(radix pointer 0)
	(completed from chart 0 item 1 by 7)

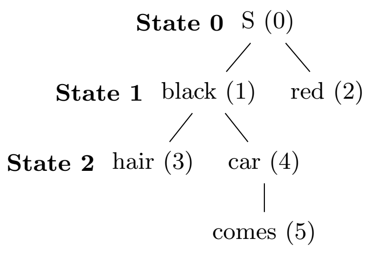


Fig. 3. Modified Earley parser states and radix tree pointers with assigned integer values

The significant points of the modified Earley parser are:

– For each node of the tree we use a vector to carry all the rules belonging to the specific node to be used on translation.

– When building radix trees of grammars, a list of a radix tree pointer is generated for the nodes of tree. Radix tree pointers are used to access a specified node, and its child nodes. Also, integer values are assigned to radix tree pointers to represent these pointers in Earley charts.

– When parsing algorithm gets to PREDICTOR, SCANNER or COMPLETER steps, standard Earley parser adds all rules to the chart according to dot position, yet we don't add rules having the same prefix. Instead of adding all rules to the chart, we add only one rule which contains radix tree pointer relating to the dot position.

– When parsing algorithm gets to PREDICTOR step, radix tree returns the set of rules to make the parsing process faster.

– When parsing algorithm gets to COMPLETER step, the radix tree pointers help to find child nodes to add chart for next dot position.

Similar to the standard Earley parser, we also have parser states and state items in our modified parser. Its internal structure is, however, slightly different compared to the standard Earley parser. In Fig. 3 modified Earley parser's states, and radix tree pointers with assigned integer values are shown. Integer values of the radix tree pointers are represented in brackets. Parsing the input *black car* is demonstrated according to the given grammar in Table 2.

4 Evaluation

To build various sizes of context-free grammars for evaluation purposes we benefited translation databases that were created by Informatics Association of Turkey (TBD). These databases include about 40,000 of translation pairs. We used different amount of translation pairs from these databases to extract translation templates, and we obtained 200,000 translation templates when all translation pairs are used in the learning phase. The prepared English and Turkish grammars are created from these extracted translation templates. Each grammar includes 200,000 rules and there are approximately 1,080,000 words in each grammar. The average length of the rules for each grammer is 5.4 words and the number of words in a rule is between 1 and 17 words.

The sizes of these grammars vary from 10 rules to 200,000 rules. All the strings used for evaluation are in a language defined by the context-free grammar, i.e. Turkish and English. In order to translate these strings, we parse them by using the standard Earley parser as well as the modified Earley parser. We evaluated average parsing time and memory usage of parsers with randomly selected strings in these datasets. Note that our parser's code is written in Java 1.8, and our evaluation results were taken on Intel i7 processor with 16 GB RAM.

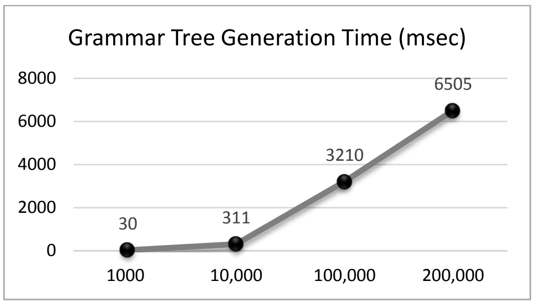
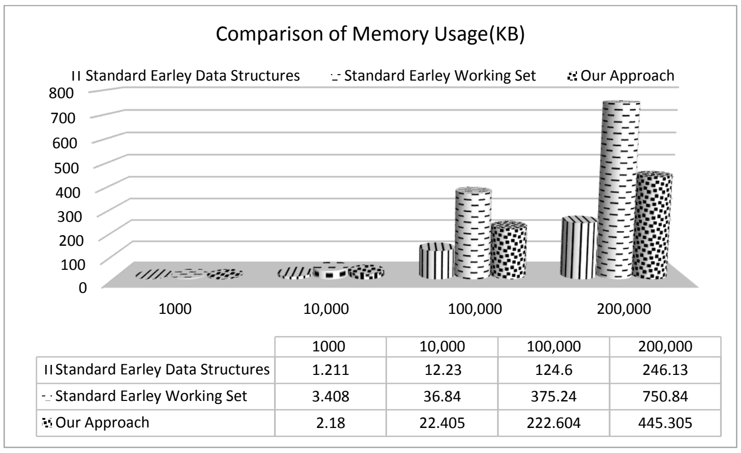


Fig. 4. Complexity of the radix tree generation in relation to the number of rules in the grammar

First, our proposed method requires the generation of the radix tree corresponding to a given context-free grammar. The time-complexity of the radix tree generation for the grammar is $O(N)$ as it is shown in Fig. 4. Generation of the radix tree for a grammar of 10,000 rules takes 311 milliseconds whereas the generation of the radix tree for a 10 times larger grammar of 100,000 rules takes about 10 times longer, 3210 milliseconds. The generation of the radix tree is required to be performed only once and furthermore even for a grammar with 200,000 rules it takes slightly longer than 6 seconds. It is definitely feasible.

Another important concern is memory efficiency. In order to evaluate memory efficiency of the approaches, we measured total memory usage of the approaches for different size of grammars. According to the results, standard Earley parser is quite conservative in terms of the memory space required for the data structures since all it does is to store the grammar rules in a list. Therefore, in terms of the size of the data structures, the standard Earley parser looks like the most efficient one in our data set as it is shown in Fig. 5. During parsing, however, the standard Earley parser seems to allocate more memory space than our approach, and uses about 64% more memory space than the modified Earley parser. In Fig. 5, standard Earley working set refers to the space requirement of the standard Earley parser for storage of rules, and charts at run time. In the figure the storage requirement of the modified Earley parser is given for both storage of rules and charts. Our approach, the modified Earley parser, uses about twice as much memory as the standard Earley parser to store the rules of the grammar. During parsing our approach uses about 39.25% less memory space than the standard Earley parser. Generally, we think it is fair to say that ours is more efficient in terms of memory usage, compared to the standard Earley parser since we are generally interested in the memory usage patterns during parsing.



	1000	10,000	100,000	200,000
▥ Standard Earley Data Structures	1.211	12.23	124.6	246.13
▤ Standard Earley Working Set	3.408	36.84	375.24	750.84
▩ Our Approach	2.18	22.405	222.604	445.305

Fig. 5. Comparison of the memory usage of the standard Earley parser's data structures, standard Earley parser's working set on running, and our modified Earley parser

The last crucial subject is the running time of parsing. To evaluate running times of the approaches, we measured average running time of the approaches for different size of grammars. According to the evaluated results, the modified Earley parser is significantly faster than the standard Earley parser for very large grammars. We have shown that the time-complexity of the standard Earley parser is $O(R^2N^3)$ where R is the number of rules in the grammar and N is the string size. The time-complexity of the modified Earley parser, however, is $O(log(R)N^3)$ where R is the number of rules in the grammar (Fig. 6). Therefore, the modified Earley parser is asymptotically faster than the standard Earley parser. Even if the number of rules in a given grammar are too large, the modified Earley parser can still parse strings quickly. The average time for parsing a string in a grammar with 10,000 rules is about 1.4 seconds, and for a 100,000 rules this increases to 2.2 seconds.

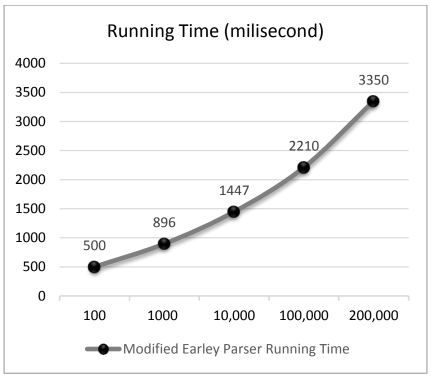


Fig. 6. The average time required to parse a single string in terms of milliseconds, in relation to the number of rules in the grammar

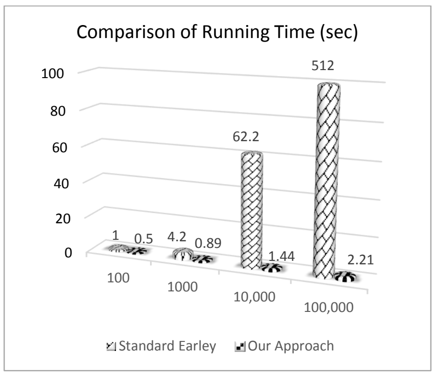


Fig. 7. Comparison between the time-complexities of the standard, and modified Earley parser algorithms in relation to the number of rules in the grammar

The comparison of our parser and the standard Earley parser showed that the modified Earley parser is asymptotically faster than the Standard Earley parser. In fact, as the language gets larger, the gap between the algorithms gets much larger. The comparison for the time-complexities of both parsers is given in Fig. 7. When there are 10,000 rules in the grammar, it takes about 62.2 seconds for the standard Earley parser to parse one string, whereas the modified parser can parse a string in 1.44 seconds. If we then increase the number of rules to 100,000, it takes about 512.0 seconds for the standard Earley parser while the modified parser can parse a string only in 2.21 seconds.

5 Related Work

Graham et al. studied a derivation of Earley's algorithm in their well detailed work [14]. They made a data structure change in the original algorithm of Earley's [7] as we do. They preferred using matrix over list. At the end of their paper, they made a conclusion that their algorithm was less complex than Earley's, and it had worst-case time complexity of $\Omega(n^3)$ where n is the input size.

Tomita [15] examined the base idea of Earley's algorithm, LR parsing algorithm. He generalized the method by precomputing an LR shift-reduce parsing table from a given grammar, and used DAG (Directed Acyclic Graph) as a data structure. He concluded that his algorithm was faster than Earley's algorithm with a five to tenfold speed advantage. Tomita was the first one that testing his system with a natural language grammar. He used a sample English grammar consisting of 220 context-free rules, and 40 sentences.

A faster earley parser was presented by McLean and Horspool [9]. They went for an algorithmic change, used a hybrid approach, and combined Earley's algorithm with LR(k) method (they named it LRE). They made an evaluation using Roskind's *ANSI C* grammar. Their study resulted 10 to 15 times faster recognition, and less than half storage requirements comparing to Earley parser.

Aycock and Horspool conducted a study about zooming in the performance of deterministic and general parsers [10]. They constructed an Earley parser which had speed comparable to deterministic parsers. Radix trees were used in this work to carry parent pointers of items having the same core, but different parents. Trees were constructed if and only if there were two or more Earley items which own the same core with varied parents. In their nodes, trees only carried either 1, or 0. They used *Java* 1.1 grammar with 350 rules to evaluate their systems. Although this study is not directly related to ours, and our tree usage principle is different than theirs as we explained in Section 3.3, we would like to point out that to our knowledge, they were the only ones to use radix trees except us in a study to build faster parsers.

Later Aycock and Horspool proposed a solution to RHS (right-hand side) grammar rules in their inspiring work [1]. They analyzed that Earley's algorithm tackled empty RHS grammar rules, and produced a solution by changing the PREDICTOR step in Earley's algorithm. They tested their parser using programming languages' files, i.e. *python* consisting of 735 files of source code, and

3234 source files from $Java - JDK$ 1.2.2. With this solution they made Earley parser 2x speed up.

Horák and Kadlec also used a natural language, Czech language, to evaluate their parsing system, *synt* [16]. In *synt*, a grammar is represented with meta-rule sets for Czech language, while we use learned rules between English and Turkish from our EBMT system. For testing they used 10,000 sentences, and 191,034 words overall; yet we use 200,000 rules for each of our grammars and they contain approximately 1,080,000 words.

6 Conclusion

In this work, we intend to build a faster, and more memory-friendly general context-free grammar parser comparing with the previous parsers, and we introduced our modified Earley parser for this purpose. We profit our parser in an EBMT application and the grammars that we use are induced natural language grammars by our EBMT system, Turkish and English in this context. We run through ϵ rules, and look-ahead tables in Earley's algorithm. We prefer radix tree as a data structure to carry our learned rules coming from EBMT. 100,000 grammar rules are used to evaluate our work. Tests showed that the bigger a grammar gets, the higher a parser's time complexity. Our modified Earley parser has a time bound of $O(log(R)N^3)$ where R is the number of rules in the grammar and N is the string size, and 20% less memory usage. It is asymptotically more efficient than the standard Earley parser in terms of time-complexity, and still slightly more efficient regarding memory usage.

References

1. Aycock, J., Horspool, R.N.: Practical earley parsing. Computer Journal **45**(6) (2002) 620–630
2. Cicekli, I., Güvenir, H.A.: Learning translation templates from bilingual translation examples. Applied Intelligence **15**(1) (2001) 57–76
3. Cicekli, I.: Inducing translation templates with type constraints. Machine translation **19**(3-4) (2005) 283–299
4. Aho, A.V., Ullman, J.D.: The Theory of Parsing, Translation, and Compiling: Volume I: Parsing. Prentice-Hall, Incorporated (1972)
5. Harrison, M.A.: Introduction to formal language theory. Addison-Wesley Longman Publishing Co., Inc. (1978)
6. Knuth, D.E.: On the translation of languages from left to right. Information and Control **8**(6) (dec 1965) 607–639
7. Earley, J.: An efficient context-free parsing algorithm. Communications of the ACM **13**(2) (1970) 94–102
8. Bouckaert, M., Pirotte, A., Snelling, M.: Efficient parsing algorithms for general context-free parsers. Information Sciences **8**(1) (1975) 1–26
9. McLean, P., Horspool, R.N.: A Faster Earley Parser. Lecture Notes in Computer Science **1060** (1996) 281–293
10. Aycock, J., Horspool, N.: Directly-executable Earley parsing. Compiler Construction (2001) 229–243

11. Earley, J.: An efficient Context-Free Parsing Algorithm. PhD thesis (1968)
12. Jacobs, C., Grune, D.: Parsing techniques: A practical guide. Department of Mathematics and Computer Science, Vrije Universiteit, Amsterdam, The Netherlands (1990)
13. Morrison, D.R.: Patricia—practical algorithm to retrieve information coded in alphanumeric. Journal of the ACM (JACM) **15**(4) (1968) 514–534
14. Graham, S.L., Harrison, M.a., Ruzzo, W.L.: An Improved Context-Free Recognizer. ACM Transactions on Programming Languages and Systems **2**(3) (1980) 415–462
15. Tomita, M.: An efficient augmented-context-free parsing algorithm. Computational linguistics **13**(1-2) (1987) 31–46
16. Horák, A., Kadlec, V.: New Meta-grammar Constructs in Czech Language Parser synt. In: Text, Speech and Dialogue: 8th International Conference, TSD 2005, Karlovy Vary, Czech Republic, September 12-15, 2005. Proceedings. Springer Berlin Heidelberg, Berlin, Heidelberg (2005) 85–92