# Speeding Up Full Syntactic Parsing by Leveraging Partial Parsing Decisions

**Elliot Glaysher** and **Dan Moldovan**
Language Computer Corporation
1701 N. Collins Blvd. Suite 2000
Richardson, TX 75080
{eglaysher,moldovan}@languagecomputer.com

## Abstract

Parsing is a computationally intensive task due to the combinatorial explosion seen in chart parsing algorithms that explore possible parse trees. In this paper, we propose a method to limit the combinatorial explosion by restricting the CYK chart parsing algorithm based on the output of a chunk parser. When tested on the three parsers presented in (Collins, 1999), we observed an approximate three–fold speedup with only an average decrease of 0.17% in both precision and recall.

## 1 Introduction

### 1.1 Motivation

Syntactic parsing is a computationally intensive and slow task. The cost of parsing quickly becomes prohibitively expensive as the amount of text to parse grows. Even worse, syntactic parsing is a prerequisite for many natural language processing tasks. These costs make it impossible to work with large collections of documents in any reasonable amount of time.

We started looking into methods and improvements that would speed up syntactic parsing. These are divided into simple software engineering solutions, which are only touched on briefly, and an optimization to the CYK parsing algorithm, which is the main topic of this paper.

While we made large speed gains through simple software engineering improvements, such as internal symbolization, optimizing critical areas, optimization of the training data format, et cetera, the largest *individual* gain in speed was made by modifying the CYK parsing algorithm to leverage the decisions of a syntactic chunk parser so that it avoided combinations that conflicted with the output of the chunk parser.

### 1.2 Previous Work

Chart parsing is a method of building a parse tree that systematically explores combinations based on a set of grammatical rules, while using a chart to store partial results. The general CYK algorithm is a bottom-up parsing algorithm that will generate all possible parse trees that are accepted by a formal grammar. Michael Collins, first in (1996), and then in his PhD thesis (1999), describes a modification to the standard CYK chart parse for natural languages which uses probabilities instead of simple context free grammars.

The CYK algorithm considers all possible combinations. In Figure 1, we present a CYK chart graph for the sentence "The red balloon flew away." The algorithm will search the pyramid, from left to right, from the bottom to the top. Each box contains a pair of numbers that we will refer to as the span, which represent the sequence of words currently being considered. Calculating each "box" in the chart means trying all combinations of the lower parts of the box's sub-pyramid to form possible sub-parse trees. For example, one calculates the results for the span $(1, 4)$ by trying to combine the results in $(1, 1)$ and $(2, 4)$, $(1, 2)$ and $(3, 4)$, and $(1, 3)$ and $(4, 4)$.

In (Collins, 1999), Collins describes three new parsers. The Model 2 gives the best output, parsing section 23 at 88.26% precision and 88.05% recall in 40 minutes. The Model 1 is by far the fastest of the three, parsing section 23 of Treebank (Marcus et al., 1994) at 87.75% precision and 87.45% recall in 26 minutes.

Syntactic Chunking is the partial parsing process of segmenting a sentence into non-

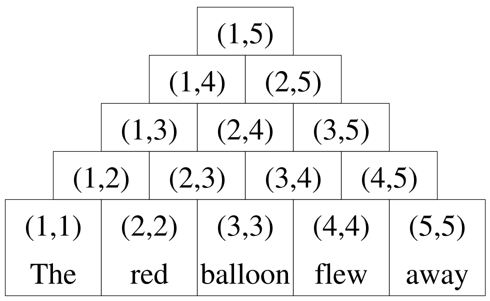

Figure 1: The CYK parse visualized as a pyramid. CYK will search from the left to right, bottom to top.

overlapping "chunks" of syntactically connected words. (Tjong Kim Sang and Buchholz, 2000) Unlike a parse tree, a set of syntactic chunks has no hierarchical information on how sequences of words relate to each other. The only information given is an additional label describing the chunk.

We use the YamCha (Kudo and Matsumoto, 2003; Kudo and Matsumoto, 2001) chunker for our text chunking. When trained on all of Penn Treebank , except for section 23 and tested on section 23, the model had a precision of 95.96% and a recall of 96.08%. YamCha parses section 23 of Treebank in 36 seconds.

Clause Identification is the partial parsing process of annotating the hierarchical structure of clauses—groupings of words that contain a subject and a predicate (Tjong Kim Sang and Déjean, 2001). Our clause identifier is an implementation of (Carreras et al., 2002), except that we use C5.0 as the machine learning method instead of Carreras' own TreeBoost algorithm (Carreras and Márquez, 2001). When trained and scored on the CoNLL 2001 shared task data[1] with the results of our chunker, our clause identifier performs at 90.73% precision, 73.72% recall on the development set and 88.85% precision, 70.22% recall on the test set.

In this paper, we describe modifications to the version of the CYK algorithm described in (Collins, 1999) and experiment with the modifications to both our proprietary parser and the (Collins, 1999) parser.

[1]http://www.cnts.ua.ac.be/conll2001/clauses/clauses.tgz

## 2 Methods

### 2.1 Software Optimizations

While each of the following optimizations, individually, had a smaller effect on our parser's speed than the CYK restrictions, collectively, simple software engineering improvements resulted in the largest speed increase to our syntactic parser. In the experiments section, we will refer to this as the "Optimized" version.

**Optimization of the training data and internal symbolization**: We discovered that our parser was bound by the number of probability hash-table lookups. We changed the format for our training data/hash keys so that they were as short as possible, eliminating deliminators and using integers to represent a closed set of POS tags that were seen in the training data, reducing the two to four byte POS tags such as "VP" or "ADJP" down to single byte integers. In the most extreme cases, this reduces the length of non-word characters in a hash from 28 characters to 6. The training data takes up less space, hashes faster, and many string comparisons are reduced to simple integer comparisons.

**Optimization of our hash-table implementation**: The majority of look ups in the hash-table at runtime were for non-existent keys. We put a bloomfilter on each hash bucket so that such lookups would often be trivially rejected, instead of having to compare the lookup key with every key in the bucket. We also switched to the Fowler/Noll/Vo (Noll, 2005) hash function, which is faster and has less collisions then our previous hash function.

**Optimization of critical areas**: There were several areas in our code that were optimized after profiling our parser.

**Rules based pre/post-processing**: We were able to get very minor increases in precision, recall and speed by adding hard coded rules to our parser that handle things that are handled poorly, specifically parenthetical phrases and quotations.

### 2.2 CYK restrictions

In this section, we describe modifications that restrict the chart search based on the output of a partial parser (in this case, a chunker) that marks groups of constituents.

First, we define a span to be a pair $c = (s, t)$, where $s$ is the index of the first word in the span and $t$ is the index of the last word in the span. We then define a set $S$, where $S$ is the set of spans

$c_1, \ldots, c_n$ that represent the restrictions placed on the CYK parse. We say that $c_1$ and $c_2$ overlap iff $s_1 < s_2 \leq t_1 < t_2$ or $s_2 < s_1 \leq t_2 < t_1$, and we note it as $c_1 \sim c_2$.[2]

When using the output of a chunker, $S$ is the set of spans that describe the non-VP, non-PP chunks where $t_i - s_i > 0$.

During the CYK parse, after a span's start and end points are selected, but before iterating across all splits of that span and their generative rules, we propose that the span in question be checked to make sure that it does not overlap with any span in set $S$. We give the pseudocode in Algorithm 1, which is a modification of the parse() function given in Appendix B of (Collins, 1999).

**Algorithm 1** The modified parse() function

```
initialize()
for span = 2 to $n$ do
  for start = 1 to $n -$ span $+ 1$ do
    end $\leftarrow$ start + span $- 1$
    if $\forall x \in S (x \not\sim (\text{start}, \text{end}))$ then
      complete(start, end)
    end if
  end for
end for
$X \leftarrow$ edge in chart[1,n,TOP] with highest probability
return $X$
```

For example, given the chunk parse:

[The red balloon]$_{NP}$ [flew]$_{VP}$ [away]$_{ADVP}$,

$S = \{(1,3)\}$ because there is only one chunk with a length greater than 1.

Suppose we are analyzing the span $(3,4)$ on the example sentence above. This span will be rejected, as it overlaps with the chunk $(1,3)$; the leaf nodes "balloon" and "flew" are not going to be children of the same parsetree parent node. Thus, this method does not compute the generative rules for all the splits of the spans $\{(2,4),(2,5),(3,4),(3,5)\}$. This will also reduce the number of calculations done when calculating higher spans. When computing $(1,4)$ in this example, time will be saved since the spans $(2,4)$ and $(3,4)$ were not considered. This example is visualized in Figure 2.

A more complex, real–world example from section 23 of Treebank is visualized in Figure 3, using the sentence "Under an agreement signed by the Big Board and the Chicago Mercantile Exchange, trading was temporarily halted in Chicago." This sentence has three usable chunks, [an agreement]$_{NP}$, [the Big Board]$_{NP}$, and [the Chicago Mercantile Exchange]$_{NP}$. This example shows the effects of the above algorithm on a longer sentence with multiple chunks.

[2]This notation was originally used in (Carreras et al., 2002).

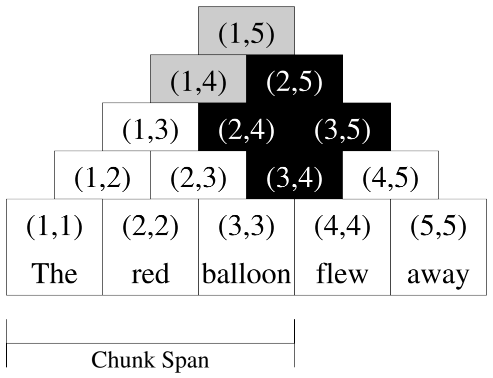

Figure 2: The same CYK example as in Figure 1. Blacked out box spans will not be calculated, while half toned box spans do not have to calculate as many possibilities because they depend on an uncalculated span.

## 3 Experiments & Results

### 3.1 Our parser with chunks

Our parser uses a simplified version of the model presented in (Collins, 1996). For this experiment,we tested four versions of our internal parser:

- Our original parser. No optimizations or chunking information.
- Our original parser with chunking information.
- Our optimized parser without chunking information.
- Our optimized parser with chunking information.

For parsers that use chunking information, the runtime of the chunk parsing is included in the parser's runtime, to show that total gains in runtime offset the cost of running the chunker.

We trained the chunk parser on all of Treebank except for section 23, which will be used as the test set. We trained our parser on all of Treebank except for section 23. Scoring of the parse trees

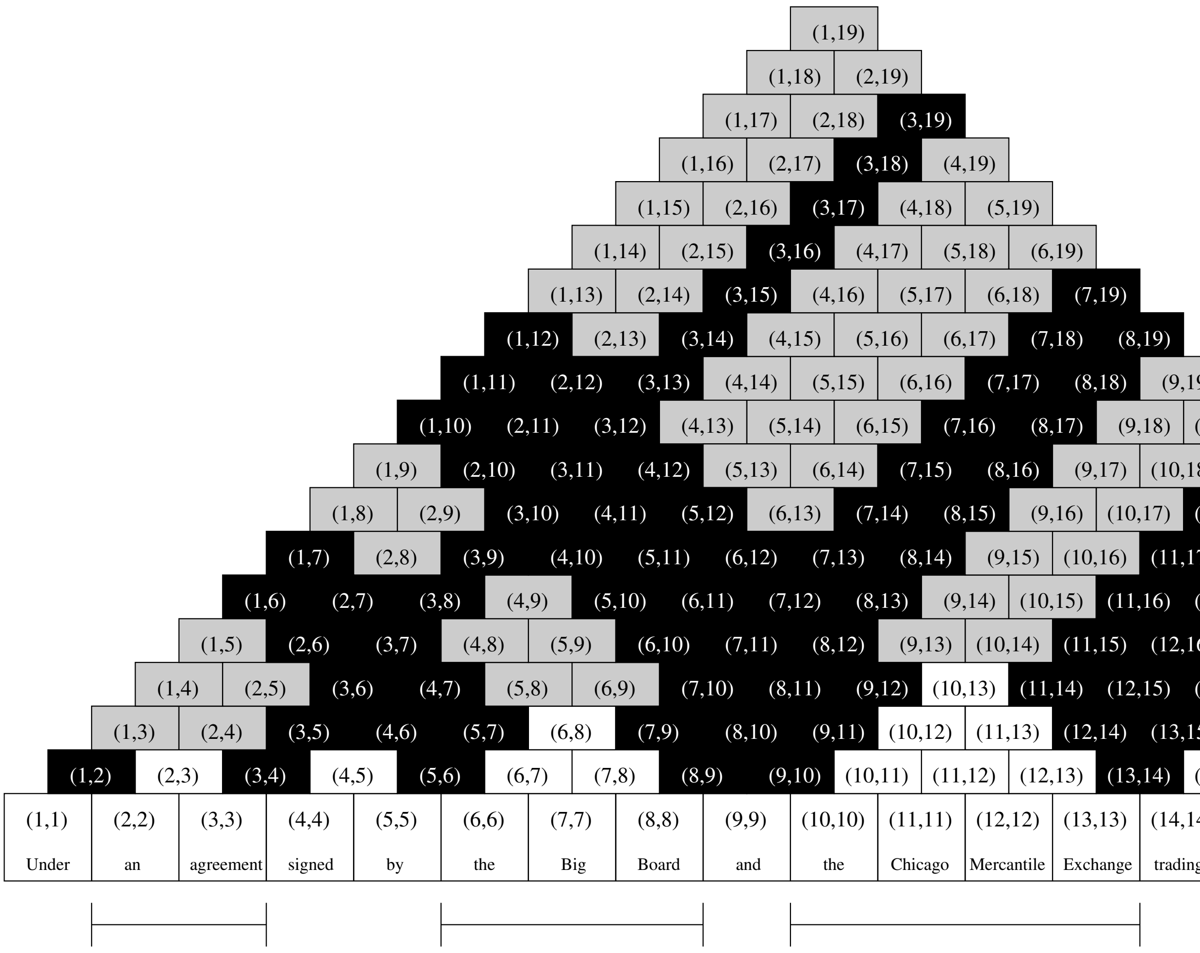

Figure 3: The CYK chart for the chunk parsed sentence "[Under]$_{PP}$ [an agreement]$_{NP}$ [signed]$_{VP}$ [by]$_{PP}$ [the Big Board]$_{NP}$ [and]$_{NP}$ [the Chicago Mercantile Exchange]$_{NP}$, [trading]$_{NP}$ [was temporarily halted]$_{VP}$ [in]$_{PP}$ [Chicago]$_{NP}$." The color coding scheme is the same as in Figure 2.

| | Precision | Recall | Time |
|---|---|---|---|
| Original | 82.79% | 83.19% | 25'45" |
| With chunks | 84.40% | 83.74% | 7'37" |
| Optimized | 83.86% | 83.24% | 4'28" |
| With chunks | 84.42% | 84.06% | 1'35" |

Table 1: Results from our parser on Section 23

| | Precision | Recall | Time |
|---|---|---|---|
| Model 1 | 87.75% | 87.45% | 26'18" |
| With chunks | 87.63% | 87.27% | 8'54" |
| Model 2 | 88.26% | 88.05% | 40'00" |
| With chunks | 88.04% | 87.87% | 13'47" |
| Model 3 | 88.25% | 88.04% | 42'24" |
| With chunks | 88.10% | 87.89% | 14'58" |

Table 2: Results from the Collins parsers on Section 23 with chunking information

was done using the EVALB package that was used to score the (Collins, 1999) parser. The numbers represent the labeled bracketing of all sentences; not just those with 40 words or less.

The experiment was run on a dual Pentium 4, 3.20Ghz machine with two gigabytes of memory.

The results are presented in Table 1.

The most notable result is the greatly reduced time to parse when chunking information was added. Both versions of our parser saw an average three–fold increase in speed by leveraging chunking decisions. We also saw small increases in both precision and recall.

### 3.2 Collins Parsers with chunks

To show that this method is general and does not exploit weaknesses in the lexical model of our parser, we repeated the previous experiments with the three models of parsers presented in the (Collins, 1999). We made sure to use the exact same chunk post-processing rules in the Collins parser code to make sure that the same chunk information was being used. We used Collins' training data. We did not retrain the parser in any way to optimize for chunked input. We only modified the parsing algorithm.

Once again, the chunk parser was trained on all of Treebank except for section 23, the trees are evaluated with EVALB, and these experiments were run on the same dual Pentium 4 machine.

These results are presented in Table 2.

Like our parser, each Collins parser saw a slightly under three fold increase in speed. But unlike our parser, all three models of the Collins parser saw slight decreases in accuracy, averaging at -0.17% for both precision and recall. We theorize that this is because the errors in our lexical model are more severe than the errors in the chunks, but the Collins parser models make fewer errors in word grouping at the leaf node level than the chunker does. We theorize that a more accurate chunker would result in an increase in the precision and recall of the Collins parsers, while preserving the substantial speed gains.

| | Precision | Recall | Time |
|---|---|---|---|
| *Optimized* | *83.86%* | *83.24%* | *4'28"* |
| *With chunks* | *84.42%* | *84.06%* | *1'35"* |
| With clauses | 83.66% | 83.06% | 5'02" |
| With both | 84.20% | 83.84% | 2'26" |

Table 3: Results from our parser on Section 23 with clause identification information. Data copied from the first experiment has been italicized for comparison.

### 3.3 Clause Identification

Encouraged by the improvement brought by using chunking as a source of restrictions, we used the data from our clause identifier.

Again, our clause identifier was derived from (Carreras et al., 2002), using boosted C5.0 decision trees instead of their boosted binary decision tree method, which performs below their numbers: 88.85% precision, 70.22% recall on the CoNLL 2001 shared task test set.

These results are presented in Table 3.

Adding clause detection information hurt performance in every category. The increases in runtime are caused by the clause identifier's runtime complexity of over $O(n^3)$. The time to identify clauses is greater then the speed increases gained by using the output as restrictions.

In terms of the drop in precision and recall, we believe that errors from the clause detector are grouping words together that are not all constituents of the same parent node. While errors in a chunk parse are relatively localized, errors in the hierarchical structure of clauses can affect the entire parse tree, preventing the parser from exploring the correct high-level structure of the sentence.

## 4 Future Work

While the modification given in section 2.2 is specific to CYK parsing, we believe that placing restrictions based on the output of a chunk parser is general enough to be applied to any generative, statistical parser, such as the Charniak parser (2000), or a Lexical Tree Adjoining Grammar based parser (Sarkar, 2000). Restrictions can be placed where the parser would explore possible trees that would violate the boundaries determined by the chunk parser, pruning paths that will not yield the correct parse tree.

## 5 Conclusion

Using decisions from partial parsing greatly reduces the time to perform full syntactic parses, and we have presented a method to apply the information from partial parsing to full syntactic parsers that use a variant of the CYK algorithm. We have shown that this method is not specific to the implementation of our parser and causes a negligible effect on precision and recall, while decreasing the time to parse by an approximate factor of three.

## References

Xavier Carreras and Lluís Márquez. 2001. Boosting trees for anti-spam email filtering. In *Proceedings of RANLP-01, 4th International Conference on Recent Advances in Natural Language Processing*, Tzigov Chark, BG.

Xavier Carreras, Lluís Màrquez, Vasin Punyakanok, and Dan Roth. 2002. Learning and inference for clause identification. In *ECML '02: Proceedings of the 13th European Conference on Machine Learning*, pages 35–47, London, UK. Springer-Verlag.

Eugene Charniak. 2000. A maximum-entropy-inspired parser. In *Proceedings of the first conference on North American chapter of the Association for Computational Linguistics*, pages 132–139, San Francisco, CA, USA. Morgan Kaufmann Publishers Inc.

Michael John Collins. 1996. A new statistical parser based on bigram lexical dependencies. In Arivind Joshi and Martha Palmer, editors, *Proceedings of the Thirty-Fourth Annual Meeting of the Association for Computational Linguistics*, pages 184–191, San Francisco. Morgan Kaufmann Publishers.

Michael John Collins. 1999. *Head-driven statistical models for natural language parsing*. Ph.D. thesis, University of Pennsylvania. Supervisor-Mitchell P. Marcus.

Taku Kudo and Yuji Matsumoto. 2001. Chunking with support vector machines. In *NAACL '01: Second meeting of the North American Chapter of the Association for Computational Linguistics on Language technologies 2001*, pages 1–8, Morristown, NJ, USA. Association for Computational Linguistics.

Taku Kudo and Yuji Matsumoto. 2003. Fast methods for kernel-based text analysis. In *ACL '03: Proceedings of the 41st Annual Meeting on Association for Computational Linguistics*, pages 24–31, Morristown, NJ, USA. Association for Computational Linguistics.

Mitchell P. Marcus, Beatrice Santorini, and Mary Ann Marcinkiewicz. 1994. Building a large annotated corpus of english: The penn treebank. *Computational Linguistics*, 19(2):313–330.

Landon C. Noll. 2005. Fnv hash. http://www.isthe.com/chongo/tech/comp/fnv/.

Anoop Sarkar. 2000. Practical experiments in parsing using tree adjoining grammars. In *Proceedings of the Fifth International Workshop on Tree Adjoining Grammars*, Paris, France.

Erik F. Tjong Kim Sang and Sabine Buchholz. 2000. Introduction to the conll-2000 shared task: Chunking. In Claire Cardie, Walter Daelemans, Claire Nedellec, and Erik Tjong Kim Sang, editors, *Proceedings of CoNLL-2000 and LLL-2000*, pages 127–132. Lisbon, Portugal.

Erik F. Tjong Kim Sang and Hervé Déjean. 2001. Introduction to the conll-2001 shared task: Clause identification. In Walter Daelemans and Rémi Zajac, editors, *Proceedings of CoNLL-2001*, pages 53–57. Toulouse, France.