Iterative CKY parsing for Probabilistic Context-Free Grammars

Yoshimasa Tsuruoka‡† **and Jun'ichi Tsujii**†‡
†Department of Computer Science, University of Tokyo
Hongo 7-3-1, Bunkyo-ku, Tokyo 113-0033
‡CREST, JST (Japan Science and Technology Agency)
Honcho 4-1-8, Kawaguchi-shi, Saitama 332-0012
{tsuruoka,tsujii}@is.s.u-tokyo.ac.jp

Abstract

This paper presents an iterative CKY parsing algorithm for probabilistic context-free grammars (PCFG). This algorithm enables us to prune unnecessary edges produced during parsing, which results in more efficient parsing. Since pruning is done by using the edge's inside Viterbi probability and the upper-bound of the outside Viterbi probability, this algorithm guarantees to output the exact Viterbi parse, unlike beam-search or best-first strategies. Experimental results using the Penn Treebank II corpus show that the iterative CKY achieved more than 60% reduction of edges compared with the conventional CKY algorithm and the run-time overhead is very small. Our algorithm is general enough to incorporate a more sophisticated estimation function, which should lead to more efficient parsing.

1 Introduction

There are several well-established $O(n^3)$ algorithms for finding the best parse for a sentence using probabilistic context-free grammars (PCFG). However, when the size of the grammar is large, the computational cost of $O(n^3)$ is quite burdensome. For example, the PCFG learned from the Penn Treebank II corpus has around 15,000 rules and the number of edges produced for parsing a 40-word sentence sometimes exceeds one million.

Many research efforts have been devoted to reducing the computational cost of PCFG parsing. One way to prune the edges produced during parsing is to use a beam search strategy, in which only the best n parses are tracked. Roark (2001) and Ratnaparkhi (1999) applied this technique to PCFG parsing. The advantage of this method is that one can incorporate beam search strategies into existing parsing algorithms without any significant additional processing cost. However, it has a major drawback: the Viterbi (highest probability) parse may be pruned during parsing, so the *optimality* of the output is not guaranteed.

Another way to reduce the number of edges produced is to use best-first or A* search strategies (Charniak et al., 1998; Caraballo and Charniak, 1998; Klein and Manning, 2003). Best-first strategies produce edges that are most likely to lead to a successful parse at each moment. While best-first strategies do not guarantee to output the Viterbi parse, an A* search always outputs the Viterbi parse by making use of an estimation function that gives an upper bound of the score to build a successful parse.

A* parsing is very promising because it can prune unnecessary edges during parsing, while keeping the optimality of the output parse. From an implementation point of view, however, A* parsing has a serious difficulty. It maintains an agenda to keep edges to be processed. The edges in the agenda are properly scored so that an appropriate edge can be retrieved from the agenda at any moment. One of the most efficient ways to implement such an agenda is to use *priority queues*, which requires a computational cost

of $O(log(n))$ at each action (Cormen et al., 2001), where n is the number of edges in the agenda. Since the process of retrieving and storing edges is conducted in the innermost loop of the A* algorithm, the cost of $O(log(n))$ makes it difficult to build a fast parser by using the A* algorithm.

In this paper, we propose an alternative way of pruning unnecessary edges while keeping the optimality of the output parse. We call this algorithm *the iterative CKY algorithm*. This algorithm is an extension of the well-established CKY parsing algorithm. It conducts repetitively CKY parsing with a probability threshold until the successful parse is found. It is easy to implement and the runtime overhead is quite small. We verified its effectiveness through experiments using the Penn Treebank II corpus.

This paper is organized as follows. Section 2 explains the iterative CKY parsing algorithm along with its pseudocode. Section 3 presents experimental results using the Penn Treebank II corpus. Section 4 offers some concluding remarks.

2 Iterative CKY parsing

The CKY algorithm is a well-known $O(n^3)$ algorithm for PCFG parsing (Ney, 1991; Jurafsky and Martin, 2000). It is essentially a bottom-up parser using a dynamic programming table. It fills out the probability table by induction.

The iterative CKY algorithm, which we present in this paper, is an extension of the conventional CKY algorithm. It repetitively conducts CKY parsing with a threshold until the successful parse is found. The threshold allows the parser to prune edges during parsing, which results in efficient parsing. The reason why we need to execute parsing repetitively is that CKY parsing with a threshold does not necessarily return a successful parse[1]. In such cases, we need to relax the threshold and conduct parsing again. When CKY does return a successful parse, it is guaranteed to be optimal.

The details of the algorithm are described in the following section.

[1]"Successful" means that the resulting parse contains S at the root of the tree.

```
function iterativeCKY(words, grammar, step)
{
  threshold = 0
  until CKY'() returns success
  {
    CKY'(words, grammar, threshold)
    threshold = threshold - step
  }
}

function CKY'(words, grammar, threshold)
{
  Create and clear π[]

  # diagonal
  for j = 2 to num_words
    for A = 1 to num_nonterminals
      if A → w_i is in grammar then
        π[i, i, A] = log(P(A → w_i))

  # the rest of the matrix
  for j = 2 to num_words
    for i = 1 to num_words-j+1
      for k = 1 to j-1
        for A = 1 to num_nonterminals
        for B = 1 to num_nonterminals
        for C = 1 to num_nonterminals
          α = π[i, k, B] + π[i + k, j − k, C]
                         +log(P(A → BC))
          if (α > π[i, j, A]) then
            β = outside(A, i, j)
            if (α + β ≥ threshold) then
              π[i, j, A] = α

  if π[1, num_words,S] has a value then
    return success
  else
    return failure
}
```

Figure 1: Pseudocode of the iterative CKY parsing algorithm. Probabilities are expressed in logarithmic form. "S" is the non-terminal symbol corresponding to a whole sentence. Outside(...) is the function that returns the upper bound of the outside Viterbi log-probability (see Section 2.2).

2.1 Algorithm

Figure 1 shows the pseudo-code of the entire algorithm of the iterative CKY parsing algorithm. Note that probabilities are expressed in logarithmic form.

The main function **iterativeCKY(...)** repetitively calls the function **CKY'(...)** giving a threshold to it until it returns a successful parse. The threshold starts with zero and is decreased by a predefined *step* at each iteration.

The function **CKY'(...)** is almost the same as the conventional CKY algorithm. The only difference is that it is given a threshold of log-probability and it prunes edges that do not satisfy the condition

$$\alpha_e + \beta_e \geq \text{threshold}, \quad (1)$$

where α_e is the inside Viterbi log-probability[2] of the edge, which is calculated in a bottom-up manner, and β_e is the upper bound of the outside Viterbi log-probability of the edge. Therefore, the sum of α_e and β_e is the highest log-probability among those of all possible successful parse trees that contain the edge. In other words, the sum is the most optimistic estimate. How to calculate β_e is described in Section 2.2.

After filling the dynamic programming table, the algorithm checks whether the non-terminal symbol "S" is in the cell in the upper right corner of the matrix. If it is, it returns *success*; otherwise, it returns *failure*.

It should be noted that if this function returns *success*, the obtained parse is optimal because pruning is done on the most optimistic estimate. The algorithm prunes only the edges that would not lead to a successful parse within the given threshold. The edges needed to build the optimal parse are never pruned.

When this function returns *failure*, it is called again with a relaxed threshold.

It is important that the value of β_e can be computed off-line. Therefore, what we need to do in runtime is just retrieve the information from the precomputed table. The runtime overhead is very small.

[2]In this paper, we use the term "inside (or outside) Viterbi probability" to refer to the probability of the Viterbi parse inside (or outside) an edge. Do not confuse it with the *inside (or outside) probability* which refers to the sum of the probabilities of all possible parses inside (or outside) an edge.

Table 1: Example of log-probability threshold and the number of edges. The log-probability of the sentence is -62.34.

Log-probability threshold	Number of edges	Parse result
0	0	failure
-10	0	failure
-20	0	failure
-30	22	failure
-40	594	failure
-50	4,371	failure
-60	16,080	failure
-70	38,201	success
Total	59,268	
Normal CKY	110,441	success

It might seem wasteful to iteratively perform CKY parsing until a successful parse is found. Somewhat counterintuitively, however, the waste is not very problematic. Although the first few attempts come to nothing, the total number of edges produced to give a successful parse is smaller than for normal CKY in most cases. Table 1 shows an example of iterative CKY parsing. Each row shows the number of edges and the parsing result. The parser conducted seven iterations until it finally obtained a successful parse. Thus the edges produced during the first six trials were wasted. However, since the number of edges increases exponentially as the threshold decreases in a constant step, the number of the wasted edges is relatively small.

The point is that we do not know the log-probability of the successful parse of a sentence in advance. If we did, setting the threshold to that log-probability would result in maximum pruning, because the tighter the threshold is, the more edges we can prune. In preliminary experiments, we tried to estimate the log-probability of a sentence using its length in order to reduce the number of wasted edges, but we did not have much success. In this paper, therefore, we take the simple strategy of decreasing by a constant step from the threshold at each iteration.

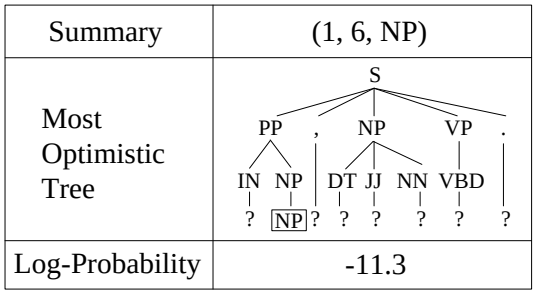

Figure 2: Context summary estimate (Klein and Manning, 2003)

2.2 Upper bound of outside Viterbi probability

There are many ways to calculate the upper bound of the outside Viterbi probability of the edge, depending on how much contextual information we specify. In this work, we use *context summary estimates* proposed by Klein (2003).

Figure 2 shows an example, where the edge is NP. In this case, the *context summary* is that NP has one word on the left and six on the right. The tree is the most optimistic tree, meaning that it has the highest probability among all possible trees that conform to this context summary. This estimate can be calculated efficiently in a recursive manner. For details of how to compute this estimation, see (Klein and Manning, 2003).

The estimates for all possible combinations of *lspan* (the number of words on the left), *rspan* (the number of words in the right), and symbols in the grammar are computed in advance. The memory size required for storing this information is

$$(\text{number of symbols}) \times \frac{(\text{max sentence length})^2}{2}. \quad (2)$$

For instance, the number of symbols in the set of binarized rules learned from the Penn Treebank is 12,946. If we parse maximum-40-word sentences, the memory required is about 80 MB (assuming that the size of each entry is 8 bytes).

We can use richer contexts for the upper bound of outside Viterbi probabilities. It is a trade-off between time and space. By using a richer context, we can obtain tighter upper bounds, which lead to more pruning. However, more space is required to store the estimates.

3 Experiment

To evaluate the effectiveness of the proposed algorithm, we conducted experiments using the Penn Treebank II corpus (Marcus et al., 1994), which is a syntactically annotated corpus in English.

3.1 Binarization

Since all rules in the grammar must be either unary or binary in CKY parsing[3], we binarized the rules that have more than two symbols on the right side in the following way.

- Create a new symbol which corresponds to the first two symbols on the right.
- The probability of the newly created rule is 1.0

This process is repeated until no rule has more than two symbols on the right side.

For example, the rule

NP $\rightarrow$ DT JJ NN (0.3)

is decomposed into the following two rules.

NP $\rightarrow$ X_{DTJJ} NN (0.3)

$\mathrm{X}_{DTJJ} \rightarrow$ DT JJ (1.0)

The probability distribution over the transformed grammar is equivalent to the original grammar. It is easy to convert a parse tree in the transformed grammar into the parse tree in the original grammar.

3.2 Corpus and grammar

Following (Klein and Manning, 2003), we parsed sentences of length 18-26 in section 22. The grammar used for parsing was learned from section 2 to 21.

We discarded all functional tags attached to nonterminals and traces, which are labeled "-NONE-", in the Treebank. The grammar learned had 14,891 rules in the original form. We binarized them and obtained 27,854 rules.

3.3 Step of threshold decrease

We first conducted experiments to estimate the optimal step of threshold decrease using a held-out set. The held-out set was created from the first 10% of section 22. The remaining sentences in the section were reserved for the test set.

[3]Strictly speaking, the original CKY algorithm requires all the rules to be binary. We used a slightly extended version of the CKY algorithm which can deal with unary rules.

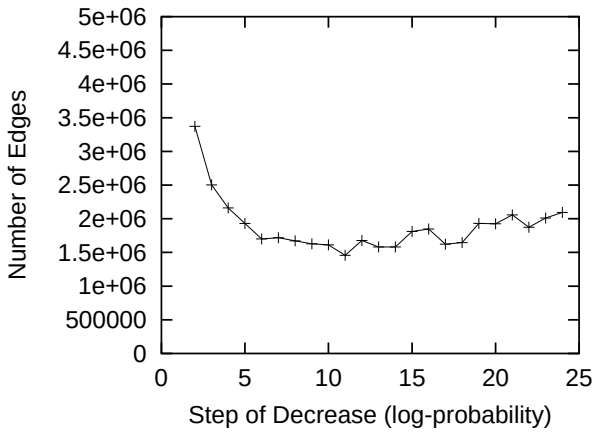

Figure 3: The step of decrease in log-probability threshold and the total number of edges.

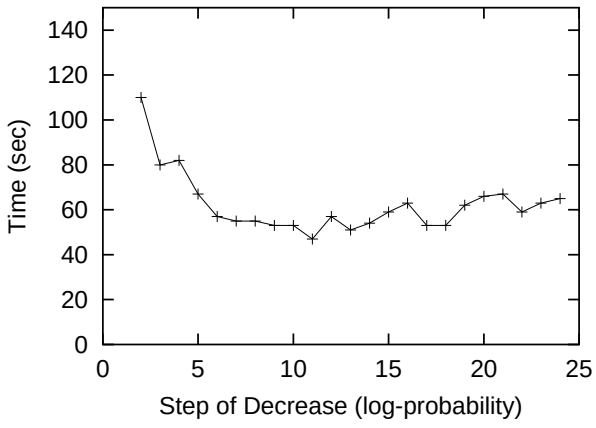

Figure 4: The step of decrease in log-probability threshold and the total time.

Table 2: The number of edges produced for parsing first several sentences in the test set.

Sentence length	Normal CKY	Iterative CKY	Ratio
19	68,366	16,343	0.24
19	69,979	4,194	0.06
24	96,185	45,232	0.47
21	110,926	15,220	0.14
23	115,296	85,634	0.74
23	102,797	18,393	0.18
25	165,564	109,002	0.66
20	86,454	21,622	0.25
22	71,797	33,819	0.47
26	127,250	62,834	0.49
:	:	:	:

Figure 3 shows the relationship between the step of threshold decrease and the total number of edges for parsing the held-out set. The best efficiency was achieved when the step was 11. The curve in the figure indicates that the efficiency is not very sensitive to the step when it is around the optimal point.

Figure 4 shows the relationship between the step and the time taken for parsing the held-out set [4]. The best setting was again 11.

3.4 Evaluation

We evaluated the efficiency of the parser on the test set. The step of threshold decrease was set to 11, which was determined by using the held-out set as described above.

Table 2 shows the number of edges produced for parsing the first several sentences in the test set. There were a few sentences for which the iterative CKY produced more edges than the normal CKY. In most cases, however, the number of edges produced by the iterative CKY was significantly smaller than that produced by the normal CKY.

Table 3 shows the the total number of edges and the total time required for parsing the entire test set. The number of edges produced by the iterative CKY was 39% of that by the normal CKY. This is a significant reduction in computational cost. As for the parsing time, the iterative CKY is almost twice as

[4]The experiments were conducted on a server having a Xeon 3.06 GHz processor and 1 GB of memory

Table 3: Performance on the test set.

	Number of edges	Time (sec)
Normal CKY	45,406,084	1,164
Iterative CKY	17,520,427	613

Table 4: Simulating ideal cases.

	Number of edges	Time (sec)
Ideal CKY	7,371,359	260

fast as the normal CKY. This result indicates that the run-time overhead of iterative CKY is quite small.

3.5 Simulating ideal cases

Klein et al. (2003) reported more than 80% reduction of edges using the same estimate under the A* search framework. Our reduction ratio is not as good as theirs, probably because our algorithm has to make several attempts that fail and the edges produced during those attempts are wasted.

In this work, we used a simple strategy of decreasing the threshold by a constant step at each iteration. It would be interesting to know how much we can further improve the efficiency by sophisticating the way of giving the threshold to the parser.

We simulated the ideal cases, where we knew the sentence probability in advance, by giving the parser the threshold that is exactly equal to the sentence log-probability. The number of edges and the total time for parsing the entire test set by the ideal parser are shown in Table 4. The results suggest that developing a method for estimating the probability of a sentence in advance should further improve the efficiency of the iterative CKY.

4 Conclusion

This paper presented an efficient and easy-to-implement iterative CKY parsing algorithm for PCFG. This algorithm enables us to prune unnecessary edges produced during parsing, which results in more efficient parsing. Since the run-time overhead of our algorithm is very small, it runs faster than the conventional CKY algorithm in an actual implementation.

Our algorithm is general enough to incorporate more sophisticated estimates of outside Viterbi probabilities, which should lead to more efficient parsing.

4.1 Future work

We used the simplest context summary estimate as the upper bound of outside Viterbi probabilities in this paper. Since tighter bounds would lead to more reduction of edges, it is worth investigating the use of other estimation methods.

References

Sharon A. Caraballo and Eugene Charniak. 1998. New figures of merit for best-first probabilistic chart parsing. *Computational Linguistics*, 24(2):275–298.

Eugene Charniak, Sharon Goldwater, and Mark Johnson. 1998. Edge-based best-first chart parsing. In *Proceedings of the Sixth Workshop on Very Large Corpora*.

Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2001. *Introduction to Algorithms*. The MIT Press.

Dainiel Jurafsky and James H. Martin. 2000. *Speech and Language Processing*. Prentice Hall.

Dan Klein and Christopher D. Manning. 2003. A* parsing: Fast exact viterbi parse selection. In *Proceedings of the HLT-NAACL*, pages 119–126.

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.

H. Ney. 1991. Dynamic programming parsing for context-free grammars in continuous speech recognition. *IEEE Transactions on Signal Processing*, 39(2):336–340.

Adwait Ratnaparkhi. 1999. Learning to parse natural language with maximum entropy models. *Machine Learning*, 34(1-3):151–175.

Brian Roark. 2001. Probabilistic top-down parsing and language modeling. *Computational Linguistics*, 27(2):249–276.