EFFICIENT IMPLEMENTATION OF THE CKY ALGORITHM

Nathan Bodenstab (bodenstab@gmail.com)

CS 506/606 Computational Linguistics, Final Project Paper, Fall 2009

ABSTRACT

When the CKY algorithm is presented in Natural Language Processing literature, it is often is described in high-level pseudo code. The implementation details of the CKY algorithm, despite being critical to efficiency, are rarely (if ever) discussed. In this paper I discuss multiple implementation approaches, and optimizations on these approaches to increase parsing time an order of magnitude when parsing with large grammars.

1. INTRODUCTION

The CKY algorithm is a popular dynamic programming algorithm that constructs the most likely syntactic parse tree given a binary grammar and an input sentence. The complexity of the algorithm is O(n^3|G|) where n is the length of the input sentence and |G| is the size of the grammar, or "grammar constant". This algorithm is often presented in Natural Language Processing (NLP) text books at a high level and no discussion is given about actual implementation details. For example, see the adaptation of Jurafsky and Martin's CYK algorithm below.

```
// CKY algorithm; Jurafsky & Martin, 2000
n=length(sentence)
for span in 2:n
 for beg in 1:n-span+1
  end=beg+span
  for mdpt in beg+1:end-1
   for A in non-terminals
    for B in non-terminals
     for C in non-terminals
      prob = chart[beg][mdpt][B]
           * chart[mdpt][end][C]
           * prob(A->B C)
      if prob > chart[beg][end][A]
       chart[beg][end][A] = prob
```

Although this algorithm will run correctly, decisions about data structures, early termination, caching, chart traversal order, and other implementation details can have a significant impact on performance. Even though the complexity of the algorithm remains at O(n^3|G|), we will see that an optimized implementation can reduce run time by an order of magnitude.

Another important factor when discussing the efficiency (and accuracy) of the CKY algorithm is the size and structure of the grammar. In general, an increase in the grammar size allows better modeling of relevant syntactic context, and improves the average accuracy of the maximum likelihood parse tree. But with the good also comes the bad. As seen in Jurafsky and Martin's CKY implementation above, an iteration over the entire space of the grammar (possible non-terminals cubed) is nested within the inner n^3 loop and has a very large impact on the run time of the algorithm. In the next section we will take a look at the accuracy/efficiency trade-offs of parsing with a large grammar under the CKY algorithm.

It should be noted that there are competing algorithms for finding a good syntactic parse tree given a binarized grammar. Agenda-based parsing [Klein & Manning] and A* parsing [Klein & Manning] traverse the space of possible parse trees in a best-first manor by assigning a Figure of Merit to each possible (beg,end,A->BC) tuple on the frontier of the search space. An agenda-based approach can find the maximum likelihood parse tree while intelligently skipping a large portion of the search space, but the overhead of maintaining a priority queue of possible frontier tuples (the agenda) must be considered.

Another recent syntactic parsing approach is Coarse-to-Fine [Charniak & Johnson; Petrov & Klein]. Coarse-to-fine parsing incrementally parses the input sentence with a larger and larger grammar, using previous parsing results to prune the subsequent search space. At each iteration, the Coarse-to-Fine algorithm uses the CKY algorithm for parsing, but only considers chart spans and non-terminals that pass a threshold in the previous (smaller grammar) round. As a result, improvements we consider in this paper for the CKY algorithm are directly applicable to the Coarse-to-Fine algorithm.

2. THE GRAMMAR CONSTANT

The most popular treebank used in the parsing community is the Penn Treebank: a 1 million word corpus with human-labeled parse tree annotations [UPenn]. The Penn Treebank uses a set of 26 phrase-level non-terminals and 36 word-level pre-terminals to classify constituents in the corpus. For instance NP is a Noun Phrase, VP is a Verb Phrase, NN is a

Noun, etc. It is straight forward to induce a Probabilistic Context Free Grammar (PCFG) from this treebank, and such a grammar is usually the baseline for parsing research.

Using sections 2-21 of the Penn Treebank for training, and section 24 for testing, this baseline grammar will contain approximately 64 thousand rules and achieve an F-score of 72.5 (F-score is the harmonic mean of precision and recall). Since the CKY algorithm finds the maximum likelihood solution, there are no search errors, and a change in the model (grammar) is required to improve accuracy.

There have been a number of publications on inducing more accurate grammars, but since grammar induction isn't the focus of this project, I will briefly describe the general concept, do some hand waving, and present relevant results.

The baseline grammar computes the probability for rule A -> B C as:

```
P(A -> B C)
    = P(B, C | A)
    = count(A -> B C) / count(A -> * *)
```

More accurate methods, such as Parent Annotation [Johnson] and lexical grammars [Charniak, Collins] include additional contextual conditioning information. For example, the parent annotated version of this rule would be (where X^Y represents non-terminal X with parent non-terminal Y in the context of a parse tree)

```
P(A^X -> B^A C^A)
    = P(B, A, C, A | A, X)
    = P(B, C | A, X)
    = count(A^X -> B^A C^A) / count(A^X -> * *)
```

As you can see with parent annotation, each non-terminal is annotated with its parent non-terminal, potentially squaring the number of non-terminals and the grammar size. Lexical annotation (the process of annotating non-terminals with the head word of the constituent) will increase the grammar size significantly more.

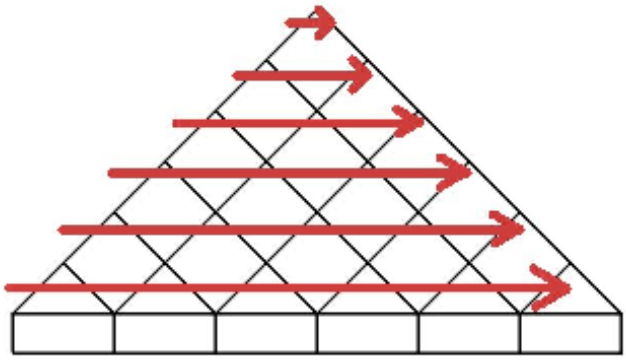
The table below contains three grammars induced from sections 2-21 of the Penn Treebank using the three methods discussed in this section.

	Gram Size	F-score	Parse Time Sec/Sent
Baseline	64k	72.5	0.3
Parent	812k	78.2	1.5
Lexical	4050k	86.1	44.3

As expected, with an increase in the size and complexity of the grammar, we see corresponding increases in accuracy and computational time.

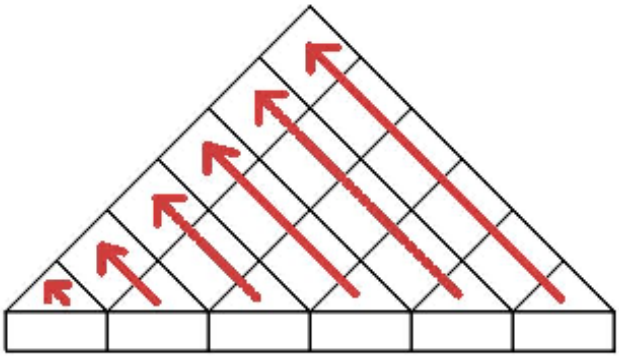
3. CHART TRAVERSAL

Chart traversal is the order in which chart cells are visited during the CKY algorithm. Before visiting a new chart cell, the necessary pre-condition is that all possible child cells have previously been visited. The most common traversal order (also the order in Jurafsky & Martin's implementation) is a bottom-to-top, left-to-right traversal:



At each "level" the span increases by one, meaning all chart cells that cover SPAN number of words are considered until SPAN equals the length of the sentence. The left-to-right aspect of this traversal is actually irrelevant. Going from right-to-left will give identical performance with no additional advantages or disadvantages.

The second traversal approach is left-to-right, maximal span; also known as a left-corner traversal. As the figure below shows, the algorithm proceeds from the left-most word index to the right-most, and at each index, all possible left parent cells are visited. Note that with this traversal, we are still guaranteed that for every cell we visit, all possible child cells will have previously been visited.



Brian Roark claims that there are advantages to the left-corner traversal by way of caching, but I was unable to verify his results in this work. I hope to investigate the advantages and disadvantages of these traversal methods more in the future.

4. CELL POPULATION

Upon visitation of a new chart cell, the CKY algorithm computes the most likely parse tree rooted at that cell for each non-terminal (there may be no valid parse for some non-terminals). Although I have found nothing in the literature discussing various cell population approaches, I would classify implementations I've found on-line and talking with colleagues into two groups: Grammar Loop and Cell Cross-Product.

In an effort to save space and increase readability, the pseudo code for both approaches will use the following function, which adds a grammar rule to a chart cell and updates the inside probability and back-pointer only if it is better than the previous entry.

```
def addEdge(beg,mdpt,end,A->B C)
 prob = chart[beg][mdpt][B]
      * chart[mdpt][end][C]
      * prob(A->B C)
 if prob > chart[beg][end][A]
  chart[beg][end][A] = prob
  bkpt[beg][end][A] = (B,C,mdpt)
```

4.1. Grammar Loop Cell Population

The Grammar Loop approach iterates over the entire grammar when visiting each cell. It is probably the most intuitive of the two approaches and is similar to the Jurafsky & Martin implementation. The pseudo code is as follows:

```
// Grammar Loop Cell Population
def populateCell(beg,end)
 for A->B C in grammar
  for mdpt in beg+1:end-1
   if B in chart[beg][mdpt]
    if C in chart[mdpt][end]
     addEdge(beg,mdpt,end,A->B C)
```

First of all, this implementation of the Grammar Loop approach will be much faster than Jurafsky & Martin's because they consider every grammar rule in the set |non-terminals|^3. Unless the grammar is smoothed, the grammar size is often much smaller than |non-terminals|^3 and iterating over the grammar rules directly is a more efficient approach.

4.2. Cell Cross-Product Cell Population

The Cell Cross-Product strategy takes all child non-terminals in the left and right cells, computes a cross product of those sets, and then applies all grammar rules with a left-hand-side that match an entry in the set of cross-products. The pseudo code is as follows:

```
// Cell Cross-Product Cell Population
def populateCell(beg,end)
 for mdpt in beg+1:end-1
  for B in chart[beg][mdpt]
   for C in chart[mdpt][end]
    for A->B C in grammar.find(B,C)
     addEdge(beg,mdpt,end,A->B C)
```

Assuming the grammar.find(B,C) function has complexity O(1), the Cell Cross-Product method will always run faster than the Grammar Loop method since, in the worst case, Cell Cross-Product will consider every grammar rule and in the average case, it will consider significantly fewer. But as we will see in the next section, each strategy allows different optimizations, and because these inner loops of the CKY algorithm can run billions of times over the course of the section 24 test set, small choices in data structures and early bailout decisions can have large effects on the overall computational time.

5. CELL POPULATION SPEED-UPS

5.1. Grammar Loop Speed-Ups

There are two important improvements that can be added to the Grammar Loop algorithm to increase its efficiency. Pseudo code for these speed-ups are below

```
// Fast Grammar Loop
def populateCell(beg,end)
 for A in grammar.possLHS()
  for A->B C in grammar.byLHS(A)
   minMP, maxMP = filter(A->B C,beg,end)
   for mdpt in minMP:maxMP
    if B in chart[beg][mdpt]
     if C in chart[mdpt][end]
      addEdge(beg,mdpt,end,A->B C)
  updateFilter(maxEdge[A],beg,end)
```

The most important improvement here is the grammar rule filter. When lexical productions are added to the chart (not shown above) the updateFilter() function is also called. This function keeps track of the max and min span for each non-terminal at each sentence index. Using this information, when the filter() function is called, it can compute the possible range of midpoints for a grammar rule A -> B C given the previously placed B and C non-terminals in the chart. For many of these filter() calls, no possible midpoints are valid and the entire inside loop is skipped for a given grammar rule.

The second Grammar Loop efficiency is iterating over the grammar rules ordered by the non-terminal of the rule's left hand side (LHS), also known as the rule head, or parent non-terminal. Considering grammar rules in this order guarantees that once all productions with the same left hand side have been processed, we know we have found the maximum likelihood entry for the left hand side non-

terminal in that cell. We can push off updating the filter and recording the back-pointer to after this left-hand-side loop is complete.

5.2. Cell Cross-Product Speed-Ups

The largest possible improvement to the Cell Cross-Product approach is to create a matrix of possible left child non-terminals by possible right child non-terminals, which points to a linked list of grammar rules containing these two non-terminals as children. The tradeoff of this data structure is that it wastes quite a bit of memory, especially if the grammar is sparse.

[side note: I've been told that this matrix-by-children isn't a good way to go because of memory requirements, but I just did a quick napkin calculation from the high-accuracy Berkeley grammar I have sitting around and it doesn't seem so bad. There are 1800 possible left non-terminals and 895 right non-terminals. The memory requirements for this matrix should be 1800*895*8 byte pointer (on a 64 bit machine) * 1 KB / 1024 bytes * 1 MB / 1024 KB ~= 12.3MB … and that's not too bad at all. Although, it's also very late right now, so I could also be way off. Anyhow, I didn't implement this approach because I assumed it wasn't memory efficient.]

I implemented two separate algorithms for the Cell Cross-Product strategy. The first initially hashes the grammar by (leftChild, rightChild), and then looks up this list of possible grammar rules in the inner loop.

```
// Cell Cross-Product Hash
def populateCell(beg,end)
 for mdpt in beg+1:end-1
  for B in chart[beg][mdpt].possLeft()
   for C in chart[mdpt][end].possRight()
    for A->B C in grammar.hash(B,C)
     addEdge(beg,mdpt,end,A->B C)
```

The advantage of a hash table is that it is space efficient (assuming it's implemented correctly) and also has O(1) look-up time. I used the default Java utility HashMap in my code, and the results were not impressive. In fact, the look-up time for the hash was quite slow. I didn't spend any time writing my own hash function, but I would assume that a custom hash function would decrease the run time of this algorithm significantly.

The second Cell Cross-Product implementation, also known as the Brian Roark CKY implementation, is as follows (although I believe Brian doesn't' separate possible left and right children in each cell).

```
// Cell Cross-Product Intersect
def populateCell(beg,end)
 for mdpt in beg+1:end-1
  for B in chart[beg][mdpt].leftNTs()
   gramR = grammar[B]
   cellR = chart[mdpt][end].rightNTs()
   for A->B C in intersect(gramR,cellR)
    addEdge(beg,mdpt,end,A->B C)
```

The first improvement is to only consider non-terminals in the left cell that participate as a left child in at least one of the grammar rule, and similarly for possible right child non-terminals. Depending on the grammar factorization, the number of possible left non-terminals and right non-terminals can be quite different.

Second, we store the grammar in an array by left non-terminal. Each entry in this list points to a linked list of possible right child non-terminals, and each of the entries in that list is another list of grammar productions that match the (left-child, right-child) entries. Once the set of possible right cell children (cellR) and right grammar children (gramR) have been collected, we need to find the intersection of these two sets, which will then point to the possible grammar rules that can be placed in the current cell.

Brian Roark has tried multiple intersection methods (merge sort, binary merge, etc) but concluded that a simple iteration through the grammar list, with a O(1) look-up into the right cell to see if the non-terminal exists, is the simplest and most efficient. Because of this, I chose to implement his method, and as the results in the next section will show, it works well.

6. RESULTS

I implemented the naïve version of both the Grammar Loop and Cell Cross-Product CKY algorithms, as well as the speedups mentioned in section 5. The code is written in Java and executed on the OGI kiwi machines, which have 8 core Intel Xeon 3.0 GHz processors and 16 GB of memory. The time reported is the average number of seconds to parse one sentence, averaged over three iterations of the Penn Treebank section 24, which contains 1346 sentences.

	Baseline 64k | 72.5	Parent 812k | 78.2	Lexical 4050k | 86.1
Gram Loop	0.81 sec	5.77 sec	xxx
Gram Loop ***Fast***	0.29 sec	2.03 sec	44.3 sec
Cell Cross-Prod	0.89 sec	10.96 sec	xxx
Cell Cross-Prod ***Fast***	0.33 sec	1.53 sec	83.9 sec

My assumption going into this experiment was that the Fast Cell Cross-Product method would perform significantly better in all experiments. The only case I imagined the Grammar Loop algorithm could possibly win

would be if a smoothed grammar was used (all non-terminals are possible in every cell). In this case, both algorithms would effectively enumerate through the entire grammar in every cell, and the straight-forwardness of the Grammar Loop algorithm to do this may give it a slight advantage.

The actual results seem to be the complete opposite. The parent annotated grammar here is actually a *smoothed* parent annotated grammar, so the one case in which Grammar Loop should win, it doesn't. I find these results very interesting and think more investigation would lead to a more satisfactory explanation of the conflict between my intuition and the empirical results. Another data point is that the Berkeley parser, which is freely available online, implements the Fast Grammar Loop for their inner loop of the CKY algorithm. Because the NLP group at Berkeley has been working on parsing for some time now, I would assume that they choose this implementation for good reason. But maybe my assumptions are misplaced.

7. CONCLUSION

We have discussed the CKY algorithm and various implementation strategies in detail. We have also shown what impact the grammar has on the CKY algorithm, in terms of accuracy and efficiency. Speed-ups for two different implementations were discussed and show to improve the efficiency of the CKY algorithm by a significant factor. Although both "fast" implementations of the Grammar Loop and Cell Cross-Product decreased the average parsing time of a sentence, the intuition behind why the Fast Grammar Loop algorithm is significantly quicker than the Fast Cell Cross-Product algorithm with the largest grammar is still a mystery. This and a comparison of chart traversal methods will be addressed in future research.

8. REFERENCES

[1] Eugene Charniak and Mark Johnson. 2005. Coarse-to-fine n-best parsing and MaxEnt discriminative reranking. In Proceedings of the 43rd Annual Meeting of the Association for Computational Linguistics (ACL'05), pages 173-180, Ann Arbor, Michigan, June. Association for Computational Linguistics.

[2] Dan Klein and Chris Manning. 2001. An O(n3) agenda-based chart parser for arbitrary probabilistic context-free grammars. Technical Report dbpubs/2001-16, Stanford University, Stanford, CA.

[3] Dan Klein and Chris Manning. 2003. A* parsing: Fast exact viterbi parse selection. In Proceedings of HLT-NAACL 2003.

[4] Slav Petrov and Dan Klein. 2007. Learning and Inference for Hierarchically Split PCFGs. In proceedings of AAAI (Nectar Track)

[5] University of Pennsylvania. http://www.cis.upenn.edu/~treebank/

[6] Eugene Charniak. 1997. Statistical parsing with a context-free grammar and word statistics. In Proceedings of the 14th National Conference on Artificial Intelligence, pp. 598-603.

[7] Mike Collins. 1999. Head-Driven Statistical Models for Natural Language Parsing. Ph.D. thesis, Univ. of Pennsylvania.

[8] Mark Johnson. 1998. PCFG models of linguistic tree representations. Computational Linguistics, 24:613-632.