Great Algorithms: CYK

Aaron Gorenstein

September 1, 2013

Abstract

The CYK algorithm, named after Cocke, Younger, and Kasami, is an algorithm for deciding if a string is in a context-free language. In particular: given a grammar G in Chomsky Normal Form and a string s, the CYK algorithm returns true exactly when $s \in L(G)$. This exposition is just to introduce the algorithm and show how it works. For brevity, we will assume you are comfortable with context-free grammars (CFG) and CFGs in Chomsky Normal Form (CNF).

1 Introduction

The CYK algorithm has a number of fascinating qualities. First and foremost, context-free languages are an important part of the Chomsky hierarchy, and this algorithm shows that deciding if $s \in L(G)$ is in P for *any* CNF CFG G. Second, the CYK algorithm is classified as a *dynamic programming* algorithm: one builds sub-solutions, and combines them to build to the answer. However, while most dynamic programming algorithms are *optimization* problems (e.g., find the *shortest* path, find the *longest* increasing subsequence, and so on), this algorithm is a *decision* problem—only returning true or false at the end. This is a very unusual quality, and I have only encountered one other such algorithm. Third and lastly, the CYK algorithm is a really clever algorithm that subtly exploits some of the structure of CNF grammars, and I think that cleverness is worth sharing.

2 The Algorithm

The key insight of the CYK algorithm is this: if a string $s \in L(G)$, then there exists a production in the CNF grammar G such that $A \to BC$, where B is a nonterminal that generates some prefix of s, and C is a nonterminal that generates the rest. To restate that in the bottom-up fashion befitting of a dynamic programming algorithm: given a string s, consider any "split" ℓ, r such that $\ell + r = s$. (So, if $s = 10001$, then one such split is $\ell = 10$ and $r = 001$.) We assume we know all the nonterminals that can generate ℓ and all the nonterminals that can generate r. To determine the nonterminals that generate s, see if there is a nonterminal called B that generates ℓ, a nonterminal C that generates r, and a nonterminal A that generates BC. In that case, we know that A generates s.

Please take care to understand the above exposition: that is the heart of the algorithm, and it packs a lot of information in few words of text. Enough dilly-dallying, let's build the algorithm! The full algorithm is in algorithm 1, and as we build our intuition for the algorithm we will reference the psuedocode for concreteness.

Clearly, for each terminal in s, $s[i]$ (the i-th symbol of the string s) there must be at least one nonterminal with the production $T \to s[i]$, otherwise s has a character not in the alphabet! That is the base case, for substrings of s that are length 1. The main nested **for** loops do the following. We consider all substrings of increasing length—so we start by considering all substrings of length 2. We let ℓ on line 4 denote the length of the substrings being considered. Note that as an artifact of the indexing system, to denote substrings of length 2 we set $\ell = 1$ (and in general, $\ell = k$ means we are considering substrings of length $k + 1$). Explicitly, we use the "slice" syntax to mean substring here, so $s[i : j]$ is the substring of s that starts with the i-th symbol and ends at the j-th symbol.

Algorithm 1: The psuedocode for the CYK algorithm

```
   Input: A string s = s[0], ..., s[|s| − 1] and CFG G in CNF form
   Output: A Boolean value indicating if s ∈ L(G)
 1 Initialize M[|s|, |s|] to a zero-indexed, two-dimensional array of empty sets
 2 for i = 0, ..., |s| − 1 do
 3 |   ∀T → s[i], add T to the set M[i, i]
 4 for ℓ = 1, ..., |s| − 1 do                                  /* every substring length */
 5 |   for r = 0, ..., |s| − ℓ − 1 do   /* every starting location for a substring of length ℓ */
 6 |   |   for t = 0, ..., ℓ − 1 do              /* every split of the substring at s[r : r + ℓ] */
 7 |   |   |   Let L = M[r, r + t], all nonterminals generating s[r : r + t]
 8 |   |   |   Let R = M[r + t + 1, r + ℓ]
 9 |   |   |   forall the pairs B ∈ L, C ∈ R do
10 |   |   |   |   ∀A → BC add A to M[r, r + ℓ]
11 if S ∈ M[0, |s| − 1] then
12 |   return true
13 else
14 |   return false
```

The variable r on line 5 marks each substring by its starting index in s: there is the length-2 substring $s[0:1]$, and the next length-2 substring $s[1:2]$, and so on up to $s[|s|-2, |s|-1]$. So the r-th substring of length ℓ is $s[r : r+\ell]$. Then we consider every split of our substring by setting t to every possible split-point (see line 6:

$$s[r : r+\ell] = s[r : r+t] + s[r+t+1 : r+\ell]. \tag{1}$$

Thus we have our two substrings, call them s_1 and s_2, defined by $s_1 = s[r : r+t]$ and $s_2 = s[r+t+1 : r+\ell]$ These s_1, s_2 are redefined for each split-point t. For each such split, we see what nonterminals $\mathcal{L} = \{B_1, B_2, \ldots\}$ generate s_1 and what nonterminals $\mathcal{R} = \{C_1, C_2, \ldots\}$ generate s_2. For every pair B_i, C_j we see if there is a production $A_k \to B_i C_j$. Then, by definition, that A_k generates $s[r : r+\ell]$, and so we store that. So we compute *all* nonterminals that can compute a given slice of s.

Ultimately, if the full string $s[1, |s|]$ is generated by the start symbol S of our grammar, we know we're done.

To compute all this efficiently, we keep a $|s| \times |s|$-sized table M. Conceptually, it is indexed by substring. So, $M[r, r+\ell]$ stores all of the nonterminals generating the substring $s[r : r+\ell]$. Clearly this is a $O(|s|^2)$-entry table.

3 Discussion

This is a really clever algorithm, and I think it is worth mulling over how it works. It really exploits the fact that CNF productions always have exactly two nonterminals: in fact in sort of exploits the converse, that if a string is generated by our CNF CFG, then $S \to AB$ where A generates some first chunk of the string, and B generates the last chunk. Here's a "discussion question" (and you thought that was only for English class!): does the algorithm *use* this structure, or rather is the algorithm simply a natural extension of this structure? This may be an algorithmic variant of the "is math invented, or discovered?" question.

Does there exist a more general version of this algorithm? In a private correspondence Frank Ferraro (`http://cs.jhu.edu/~ferraro/`) pointed me to the thesis "Semiring Parsing" by Joshua Goodman (1998), and it seems to me that really flexes this algorithm to its fullest extent. Are there other problems, like maybe determining some characteristic of a binary tree, which can also be solved in this method? It is also interested that while tree-based problems are usually $O(n)$, this algorithm (with an implicit tree) is $O(n^3)$.

Lastly, there is exactly one other natural problem I can think of which uses dynamic programming for a decision problem: the gerrymandering problem presented in Tardos and Kleinberg. Are there more?

4 Further reading

As the name suggests, there were three authors who independently developed this algorithm. Here is one such paper: `http://dx.doi.org/10.1016/S0019-9958(67)80007-X`.

The Wikipedia article on this algorithm (`http://en.wikipedia.org/wiki/CYK_algorithm`) presents a different form of the algorithm, and of course has many related links and discussions. Without criticizing their work, I found the algorithm variant I presented here much easier to understand.

Speaking of the algorithm I chose, it is from the textbook "The Design and Analysis of Computer Algorithms", by Aho, Hopcroft, and Ullman.[1] Whenever any subset of those authors wrote a book, it became a fixture of the field. That book was kindly lent to me, and my questions were patiently answered, by Professor Joel Seiferas from the University of Rochester (`http://www.cs.rochester.edu/u/joel/`).

There are code implementations of this algorithm: two examples I found (with no particular endorsement on my part): `http://patricklerner.com/2012/05/CYK-Algorithm_in_newLISP.html`, `www.cs.helsinki.fi/u/andrews/` (which links to `http://www2.tcs.ifi.lmu.de/SeeYK/`). I also have decided to implement this algorithm for myself, in C++, source code available upon request.

[1] Correction: this algorithm is an *exercise* in that book (oops!) and is actually presented in Hocroft and Ullman's Introduction to Automata Theory, Languages, and Computation 1979 edition, around page 139.