

CS 273

Lecture 27: Wrap up

29 April 2008

"Then you must begin a reading program immediately so that you man understand the crises of our age," Ignatius said solemnly. "Begin with the late Romans, including Boethius, of course. Then you should dip rather extensively into early Medieval. You may skip the Renaissance and the Enlightenment. That is mostly dangerous propaganda. Now, that I think about of it, you had better skip the Romantics and the Victorians, too. For the contemporary period, you should study some selected comic books."

"You're fantastic."

"I recommend Batman especially, for he tends to transcend the abysmal society in which he's found himself. His morality is rather rigid, also. I rather respect Batman."

– A confederacy of Dunces, John Kennedy Toole

This lecture looks forward to the classes that follow 273. It's short, because we are doing ICES forms at the end.

Next year, you'll probably be taking Algorithms (CS 473) and Programming Languages and Compilers (CS 421). You might also be taking elective courses in Artificial Intelligence (CS 440) or Computational Linguistics (Ling 406). Notice also that Julia Hockenmaier (new faculty) is teaching a CS 498 on parsing next fall (which you all might want to take).

In this lecture, we'll do a quick introduction to NP-completeness, which is an important topic in Algorithms with close connections to undecidability. We will also see the CYK parsing algorithm, as a lead-in to practical algorithms for parsing programming languages and human language.

1 NP completeness

The question governing computer science is mostly the development of efficient algorithms. Hopefully, what is an algorithm is a well understood concept. But what is an efficient algorithm? A natural answer (but not the only one!) is an algorithm that runs quickly.

What do we mean by quickly? Well, we would like our algorithm to:

1. Scale with input size. That is, it should be able to handle large and hopefully huge inputs.
2. Low level implementation details should not matter, since they correspond to small improvements in performance. Since faster CPUs keep appearing it follows that such improvements would (usually) be taken care of by hardware.

3. What we will really care about are asymptotic running time. Explicitly, polynomial time.

In our discussion, we will consider the input size to be n , and we would like to bound the overall running time by a function of n which is asymptotically as small as possible. An algorithm with better asymptotic running time would be considered to be *better*.

Example 1.1 It is illuminating to consider a concrete example. So assume we have an algorithm for a problem that needs to perform $c2^n$ operations to handle an input of size n , where c is a small constant (say 10). Let assume that we have a CPU that can do 10^9 operations a second. (A somewhat conservative assumption, as currently [Jan 2006]¹, the blue-gene supercomputer can do about $3 \cdot 10^{14}$ floating-point operations a second. Since this super computer has about 131,072 CPUs, it is not something you would have on your desktop any time soon.) Since $2^{10} \approx 10^3$, you have that our (cheap) computer can solve in (roughly) 10 seconds a problem of size $n = 27$.

But what if we increase the problem size to $n = 54$? This would take our computer about 3 million years to solve. (In fact, it is better to just wait for faster computers to show up, and then try to solve the problem. Although there are good reasons to believe that the exponential growth in computer performance we saw in the last 40 years is about to end. Thus, unless a substantial breakthrough in computing happens, it might be that solving problems of size, say, $n = 100$ for this problem would forever be outside our reach.)

The situation dramatically change if we consider an algorithm with running time $10n^2$. Then, in one second our computer can handle input of size $n = 10^4$. Problem of size $n = 10^8$ can be solved in $10n^2/10^9 = 10^{17-9} = 10^8$ which is about 3 years of computing (but blue-gene might be able to solve it in less than 20 minutes!).

Thus, algorithms that have asymptotically a polynomial running time (i.e., the algorithms running time is bounded by $O(n^c)$ where c is a constant) are able to solve large instances of the input and can solve the problem even if the problem size increases dramatically.

Can we solve all problems in polynomial time? The answer to this question is unfortunately no. There are several synthetic examples of this, but in fact it is believed that a large class of important problems can not be solved in polynomial time.

Problem: Circuit Satisfiability

Instance: A circuit C with m inputs

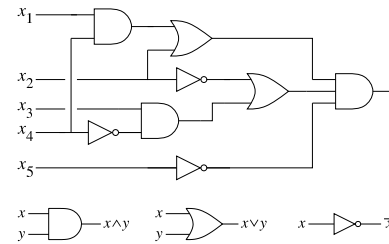
Question: Is there an input for C such that C returns true for it.

¹But the recently announced Super Computer that would be completed in 2011 in Urbana, is naturally way faster. It supposedly would do 10^{15} operations a second (i.e., petaflop). Blue-gene probably can not sustain its theoretical speed stated above, which is only slightly slower.

As a concrete example, consider the circuit depicted on the right.

Currently, all solutions known to **Circuit Satisfiability** require checking all possibilities, requiring (roughly) 2^m time. Which is exponential time and too slow to be useful in solving large instances of the problem.

This leads us to the most important open question in theoretical computer science:



Question 1.2 Can one solve **Circuit Satisfiability** in polynomial time?

The common belief is that **Circuit Satisfiability** can **NOT** be solved in polynomial time.

Circuit Satisfiability has the property that given a supposed positive solution, with a detailed assignment (i.e., proof): $x_1 \leftarrow 0, x_2 \leftarrow 1, \dots, x_m \leftarrow 1$ one can verify in polynomial time if this assignment really satisfies C . This is done by computing what every gate in the circuit what its output is for this input. Thus, computing the output of C for its input. This requires evaluating the gates of C in the right order, and there are some technicalities involved, which we are ignoring. (But you should verify that you know how to write a program that does that efficiently.)

Intuitively, this is the difference in hardness between coming up with a proof (hard), and checking that a proof is correct (easy).

1.1 Complexity classes

Definition 1.3 (P: Polynomial time) Let P denote is the class of all decision problems that can be solved in polynomial time in the size of the input.

Definition 1.4 (NP: Nondeterministic Polynomial time) Let NP be the class of all decision problems that can be verified in polynomial time. Namely, for an input of size n , if the solution to the given instance is true, one (i.e., an oracle) can provide you with a proof (of polynomial length!) that the answer is indeed **TRUE** for this instance. Furthermore, you can verify this proof in polynomial time in the length of the proof.

The question of whether **Circuit Satisfiability** can be solved efficiently, is essentially thus the following question.

Question 1.5 $NP = P$?

Several observations:

- Clearly, if a decision problem can be solved in polynomial time, then it can be verified in polynomial time. Thus, $P \subseteq NP$.
- If a problem can be solved in polynomial time on a “real” computer it can be solved in polynomial time on a TM.

- In particular, NP are just problems that can be solved in polynomial time on a non-deterministic TM.

As such, it is quite possible that $P = NP$, although this would be extremely surprising.

Definition 1.6 A problem Π is NP-HARD, if being able to solve Π in polynomial time implies that $P = NP$.

Question 1.7 *Are there any problems which are NP-HARD?*

Intuitively, being NP-HARD implies that a problem is ridiculously hard. Conceptually, it would imply that proving and verifying are equally hard - which nobody that did 473g believes is true.

In particular, a problem which is NP-HARD is at least as hard as ALL the problems in NP, as such it is safe to assume, based on overwhelming evidence that it can not be solved in polynomial time.

Theorem 1.8 (Cook's Theorem) *Circuit Satisfiability* is NP-HARD.

We do not have enough time to prove Cook's theorem formally, but in fact, we have all the tools to do it:

1. We need to show that given a decision problem that have a decider that is a non-deterministic TM then we can generate a circuit of polynomial size such that deciding if it has a satisfying assignment is equivalent to finding a sequence of guess of polynomial length such that if we run the NTM with this sequence of guess on the given input then it would accept.
2. So, given an NTM M , we first convert it so that it uses a tape with 0, 1 as an alphabet, and at every stage, its non-deterministic guess is either 0 or 1.
3. Let assume we know that M makes at most n^{10} steps when running, for an input of size n . As such, it can use at most n^{10} cells of its tape. We introduce a variable for each one of these cells and its value at any point in time. We explicitly write formulas (i.e., circuits) that verify that the values of this variable changes only according to the TM logic. Similarly, we verify that the transitions are legal (again, but using formals, and encoding the state of the TM at each one of these steps by a set of variables, and making sure again, that there is a satisfying assignment only if they change legally.
4. Now, very careful inspection of the generated circuit/formula (that has a polynomial size), reveals that M accepts its input, if and only if, the generated circuit is satisfiable.

2 CYK parsing

In lecture 19, we saw that A_{CFG} is decidable, but the construction made no attempt to be practical. There were two problems with this algorithm. First, we converted the grammar to Chomsky Normal Form (CNF). Most practical applications need the parse to show the

structure using the original input grammar. Second, our method blindly generated all parse trees of the right size, without regard to what was in the string to be parsed. This is very inefficient since there may be a very large number of parse trees of this size.

The Cocke-Younger-Kasami (CYK) algorithm solves the second of these problems, using a clever data-structure called the “chart.” This basic technique can be extended (e.g. Earley’s algorithm) to handle grammars that are not in Chomsky Normal Form and to linear-time parsing for special types of CFGs.

In general, the number of parses for a string w is exponential in the length of w . For example, consider the sentence “Mr Plum kill Ms Marple at midnight in the bedroom with a sword.” There are three prepositional phrases: “at midnight”, “in the bedroom,” and “with a sword.” Each of these can either be describing the main action (e.g. the killing was done with a sword) or the noun right before it (e.g. there are several bedrooms and we mean the one with a sword hanging over the dresser). Since each decision is independent, a sentence with k prepositional phrases like this has 2^k possible parses.²

So it’s really bad to organize parsing by considering all possible parse trees. Instead, consider all substrings in our input. If w has length n , then it has $\sum_{k=1}^n (n - k + 1) = \frac{n(n+1)}{2}$ substrings.³ CYK computes a table summarizing the possible parses for each substring. From the table, we can quickly tell whether an input has a parse and extract one representative parse tree.⁴

Suppose the input sentence w is “Jeff trains geometry students” and the grammar has start symbol S and the following rules:

$S \rightarrow N VP$

$N \rightarrow N N$

$VP \rightarrow V N$

$N \rightarrow \text{students} \mid \text{Jeff} \mid \text{geometry} \mid \text{trains}$

$V \rightarrow \text{trains}$

CYK builds a table containing a cell for each substring. The cell for a substring x contains a list of variables V from which we can derive x (in one or more steps).

length	4				
	3				
	2				
	1	Jeff	trains	geometry	students
		first word in substring			

The bottom row contains the variables that can derive each substring of length 1. This is easy to fill in:

²In real life, long sentences in news reports often exhibit versions of this problem.

³Draw an n by $n + 1$ rectangle and fill in the lower half.

⁴It still takes exponential time to extract all parse trees from the table.

length	4			
	3			
	2			
	1	N	N,V	N
		Jeff	trains	geometry
				students
				first word in substring

Now we fill the table row-by-row, moving upwards. To fill in the cell for a 2-word substring x , we look at the labels in the cells for its two constituent words and see what rules could derive this pair of labels. In this case, we use the rules $N \rightarrow N N$ and $VP \rightarrow V N$ to produce:

length	4			
	3			
	2	N	N,VP	N
	1	N	N,V	N
		Jeff	trains	geometry
				students
				first word in substring

For each longer substring x , we have to consider all the ways to divide x into two shorter substrings. For example, suppose x is the substring of length 3 starting with “trains”. This can be divided into (a) “trains geometry” plus “students” or (b) “trains” plus “geometry students.”

Consider option (a). Looking at the lower rows of the table, “students” has label N . One label for “trains geometry” is VP , but we don’t have any rule whose righthand side contains VP followed by N . The other label for “trains geometry” is N . In this case, we find the rule $N \rightarrow N N$. So one label for x is N . (That is, x is one big long compound noun.)

Now consider option (b). Again, we have the possibility that both parts have label N . But we also find that “trains” could have the label V . We can then apply the rule $VP \rightarrow V N$ to add the label VP to the cell for x .

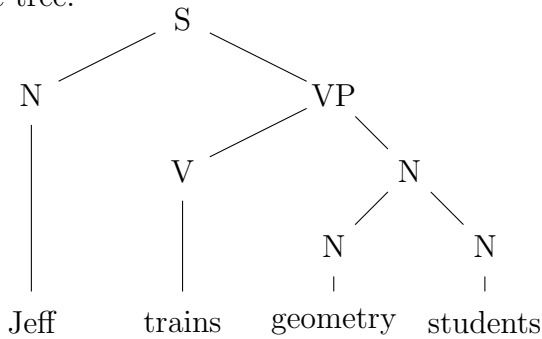
length	4			
	3		N,VP	
	2	N	N,VP	N
	1	N	N,V	N
		Jeff	trains	geometry
				students
				first word in substring

Repeating this procedure for the remaining two cells, we get:

length	4	N,S		
	3	N,S	N,VP	
	2	N	N,VP	N
	1	N	N,V	N
		Jeff	trains	geometry
				students
				first word in substring

Remember that a string is in the language if it can be derived from the start symbol S . The top cell in the table contains the variables from which we can derive the entire input string. Since S is in that top cell, we know that our string is in the language.

By adding some simple annotations to these tables as we fill them in, we can make it easy to read out an entire parse tree by tracing downwards from the top cell. In this case, the tree:



We have $O(n^2)$ cells in the table. For each cell, we have to consider n ways to divide its substring into two smaller substrings. So the table-filling procedure takes only $O(n^3)$ time.