Chapter 16

NP

Some tasks definitely require exponential time. That is, we can not only display an exponential-time algorithm, but we can also prove that the problem cannot be solved in anything less than exponential time. For example, we've seen how to solve the Towers of Hanoi problem using $O(2^n)$ steps, and it is known that no better algorithm is possible.

However, there is another large class of tasks where the best known algorithm is exponential, but no one has proved that it is impossible to construct a polynomial-time algorithm. This group of problems is known as **NP**.[1] It is an important unsolved question whether the problems in NP really require exponential time.

16.1 Finding parse trees

Suppose we are given an input sentence, such as

> Scores of famished zombies are roaming the north end of campus.

If the sentence contains n words and we have a context-free grammar G, it takes $O(n^3)$ time for a properly-designed parsing algorithm to find a parse

[1] NP is short for "non-deterministic polynomial," but that full name makes sense only when you have additional theory background.

tree for the sentence or determine that none exists. However, generating **all** parse trees for such a sentence takes exponential time.

To see this, consider sentences that end in a large number of prepositional phrases, e.g.

> Prof. Rutenbar killed a zombie with a red hat near the Academic Office on the ground floor by the statue with a large sword.

When building the parse tree for such a sentence, each prepositional phrase needs to be associated with either the main verb or with some preceding noun phrase. For example, "with a large sword" might be the instrument used to kill the zombie, in which case it should be linked with "killed" in the parse tree. But it's also possible that the sword is part of the description of the statue, in which case "with a large sword" would be attached to "statue" in the tree. No matter what you do with earlier bits of the sentence, there are at least two possible ways to add each new phrase to the parse tree. So if there are n prepositional phrases, there are at least 2^n possible parse trees.

This example may seem silly. However, similar ambiguities occur with other sorts of modifiers. E.g. in "wild garlic cheese," which is wild, the garlic or the cheese? In normal conversation, sentences tend to be short and we normally have enough context to infer the correct interpretation, and thus build the correct parse tree. However, explosion in the number of possible parse trees causes real issues when processing news stories, which tend to have long sentences with many modifiers.

16.2 What is NP?

Now, let's look at a problem that is in NP: **graph colorability.** Specifically, given a graph G and an integer k, determine whether G can be colored with k colors. For small graphs, it seems like this problem is quite simple. Moreover, heuristics such as the greedy algorithm from Chapter 11 work well for many larger graphs found in practical applications. However, if $k \geq 3$, for unlucky choices of input graph, all known algorithms take time that is exponential in the size of the graph. However, we don't know whether exponential time is

really required or whether there is some clever trick that would let us build a faster algorithm.

To make the theoretical definitions easier to understand, let's upgrade our colorability algorithm slightly. As stated above, our colorability algorithm needs to deliver a yes/no answer. Let's ask our algorithm to, in addition, deliver a justification that its answer is correct.[2] For example, for graph colorability, a "yes" answer would also come with an assignment of the k colors to the nodes of G. It's easy to verify that such an assignment is a legal coloring using the specified number of colors and, thus, that the "yes" answer is correct.

The big difference between graph colorability and parse tree generation is this ability to provide succinct, easily checked solutions with justifications. Specifically, a computational problem is in the set **NP** when an algorithm can provide justifications of "yes" answers, where each justification can be checked in polynomial time. That is, the justification checking time is polynomial in the size of the input to the algorithm (e.g. the graph G in our colorability problem). This implies that the justification itself must be succinct, because the checker must read through it. Graph colorability is in NP. Parse tree generation is not in NP, because even the raw answer without justification has exponential length.

For graph colorability, it only seems to be possible to provide succinct justifications for "yes" answers. For "no" answers, we can sometimes give a short and convincing explanation. But, for difficult graphs, we might have to walk through all exponentially-many possible assignments of colors to nodes, showing why none of them is a legal coloring. So there doesn't seem to be a general-purpose way to provide succinct justifications for negative answers.

Obviously, the choice of which answer is "yes" vs. "no" depends on how we phrase the problem. Consider the **non-colorability problem**: given a graph G and an integer k, is it impossible to color G with k colors? For this problem, we can give succinct justifications for "no" answers but apparently not for "yes" answers. Problems for which we can give polynomial-time checkable justifications for negative answers are in the set **co-NP**.

There are some difficult problems where we can provide justifications for both negative and positive answers. For example, there are efficient ways

[2]Like we force you folks to do, when answering questions on theory exams!

to verify whether an integer is prime or not, but factoring integers is much harder. For example, we do not know of any polynomial-time way to determine whether an integer m has a factor in the range $[2, n]$.[3] However, having gone to the considerable pain of factoring m, we can provide succinct, easily-verified justifications for both "yes" and "no" answers using the prime factorization of m. So this problem is in both NP and co-NP.

Algorithms that can be solved in polynomial time are in the set P. P is a subset of both NP and co-NP. Theoreticians strongly suspect that these three sets—P, NP, and co-NP—are all different. However, no one has found a way to prove this. So it is possible that all three sets are equal, i.e. that we've missed some big trick that would let us build more efficient algorithms for the problems in NP and co-NP.

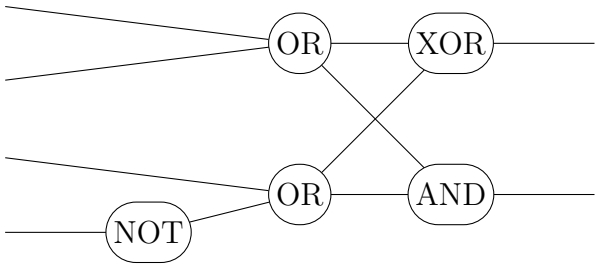
16.3 Circuit SAT

Another interesting problem in NP is satisfiability of boolean circuits. A **boolean circuit** is a type of graph consisting of a set of **gates** connected together by wires. Each wire carries one of the two values 0 (false) and 1 (true). The label on each gate describes how the value on its output wires (all output wires have the same value) is determined by the values on its input wires. For example, the output wires of an AND gate have value 1 if both input wires have value 1, and the output wires have value 0 otherwise.

Like a logical operator, the behavior of a gate can be described by a truth table. For simplicity, we'll restrict ourselves to gates with one or two inputs. A boolean circuit with any possible behavior can be built using the basic three operators (AND, OR, NOT). However, in practical applications, it's common to also use other gates, e.g. XOR (exclusive OR), NAND (negation of AND).

For example, here is a very simple boolean circuit. In this picture, inputs are on the left and outputs on the right. For ease of understanding by software and theory folks, the type of each gate is indicated with the name of the operator. For serious circuit design, you'd need to memorize the special node shapes commonly used for different gate types.

[3] That is, polynomial in the number of digits in the input numbers.



When designing circuits for practical applications, it's important to ensure that the circuit really does what it is supposed to do. Or, at least, that it doesn't make certain critical errors that would lead to tragedy. For example, the circuit controlling a 4-way intersection must ensure that two crossing lanes of traffic cannot both have a green light at the same time. A slow-moving 6-legged robot should never raise more than three of its legs at a time. A heating system should not open the gas jet if the pilot light is off. Significant portions of such controllers are well-modelled as boolean circuits.

To find significant design flaws in a boolean circuit, we need to determine whether there is a set of n input values could create one of the forbidden patterns of output values. This problem is called *circuit satisfiability*. There is an obvious exponential algorithm for circuit satisfiability: if we have n input wires, generate all 2^n possible input patterns and see what happens on the output wires. That's too slow, so circuit testers must be content with partial testing. However, if we determine that the bad output is possible, we can provide a succinct easily-verified justification: show one pattern of input values that gives rise to the bad output. So this problem is in NP.

Sadly, what we'd really like for practical applications would be easily-verified justifications of negative answers. Or, equivalently, positive answers to the the complementary problem of circuit safety: is the circuit safe from bad outputs? This seems to be much harder than justifications for circuit satisfiability, because proving a negative answer seems to require walking

through all exponentially many input patterns and showing why none of them produces the bad output pattern. So circuit safety is in co-NP but apparently not in NP.

16.4 What is NP complete?

An interesting, and very non-obvious, fact about the harder problems in NP is that they share a common fate. It can be proven that finding a polynomial-time algorithm for either circuit satisfiability or graph colorability would imply that *all* problems in NP have polynomial-time algorithms. Such problems are called *NP-complete*.

Other examples of NP complete problems include

- Propositional logic satisfiability: given an expression in propositional logic, is there an assignment of true/false values to the variables which would make the logic expression true?
- Clique: given an integer n and a graph G, does G contain a copy of K_n?
- Vertex cover: given a graph G, find a set S of nodes in G such that each edge of G has at least one of its endpoints in S.

Circuit satisfiability and propositional logic satisfiability remain NP-complete even when these problems are drastically simplified. For example, nothing changes if we restrict our circuit to have only a single output. We can restrict our logic expression to be the AND of some number of 3-variable OR expressions. Or we can use only the primitive operators/gates AND, OR, and NOT. Only the most grossly simplified versions of these problems fail to be NP-complete.

There are two techniques for showing that a problem is NP-complete. First, we can show directly that the problem is so general that it can simulate any other problem in NP. The details are beyond the scope of the book. But notice that circuits are the backbone of computer construction and logic is the backbone for constructing mathematics, so it makes sense that both

mechanisms are general and powerful. Once we have proved that some key problems are NP-complete, we can then show that other problems are NP-complete by showing that the new problem can simulate the basic primitives of a problem already known to be NP-complete. For example, we might simulate logic expressions using colored graph machinery.

16.5 Variation in notation

A justification for a positive answer can also be called a "proof," "certificate," or "witness."