Time Complexity

The State of Knowledge Before NP-Completeness

Up to and through the 1960s, algorithm designs such as greedy algorithms, local optimization, divide-and-conquer, and dynamic programming were applied successfully to many problems. But there were also plenty of problems which defied all attempts at solving them with reasonably fast algorithms[1]. Here are four examples of such "intractable" problems.

SATISFIABILITY

SATISFIABILITY, introduced in lecture 1, can be phrased in terms of Boolean expressions or in terms of circuits. In terms of Boolean expressions, the problem is to find out, given an expression E, whether or not there is an assignment of Boolean values to the variables of E that makes E true. In terms of circuits, the problem is to find out, given a circuit C, whether or not there is an assignment of Boolean values to the wires in the circuit that "satisfies" all the gates. "Satisfying a gate" means assigning consistent values to its input and output wires. For example, assigning *true* to both inputs of an AND-gate and *false* to its output would not be consistent, nor would be assigning *true* to one input wire, *false* to the other, and *true* to the output wire.

Boolean expressions or circuits — one can easily translate circuits into expressions and vice versa — turn out to be a rather flexible "language" for expressing constraints on solutions in many situations.

GRAPH 3-COLORABILITY (3-COLOR)

GRAPH 3-COLORABILITY (3-COLOR) problem also was introduced in Chapter 1. It is really a scheduling problem, trying to fit a set of activities, some of which cannot take place at the same time, into three time slots. This situation is captured by a graph with the activities being the nodes and with conflicting activities being connected by an edge. Then the problem becomes, given a

[1]On the question of what is "reasonably fast," there is general agreement that the exponential running of exhaustive search is not "reasonably fast." Below that, let's say that any algorithm whose running time is bounded by a polynomial in the input length is "reasonably fast."

graph, to find out if it is possible[2] to color each node with one of the three colors without ever giving the same color to the two endpoints of an edge.

VERTEX COVER

VERTEX COVER is another problem defined on graphs. The challenge is to "cover" the graph in the following sense. You can choose a node and thereby "cover" all the edges connected to that node. The problem is to cover all the edges of the graph choosing the smallest number of nodes. For example, the graph in Figure 4.1 can be covered with 4 nodes, but not with 3. In the decision version of this problem you are given a graph G and a number n, and the Yes/No question to decide is whether or not the graph G can be covered with n nodes.

HAMILTONIAN PATH

Think of a graph as a map of cities that are connected by roads. In the HAMILTONIAN PATH problem the challenge is to find a tour from a given start node to a given end node that visits each city exactly once. This is a special case of the so-called TRAVELING SALESMAN PROBLEM (TSP) where each edge has a length and the challenge is to find a tour of minimal length that visits each city.

How Algorithm Designs Can Fail: One Example

It is instructive to try to apply some of our algorithm design ideas to some of these intractable problems. As one example, let us try to solve SAT with a divide-and-conquer approach.

Dividing a Boolean expression is easy. Consider the parse tree of the expression and divide it into the left and right subtree. Figure 4.3 provides an illustration.

A simple-minded notion of "conquering" a subtree would be to find out whether or not it is satisfiable. But this information about a subtrees is not enough to find out whether or not the whole expression is satisfiable. For example, if both subtrees are satisfiable and are connected with an AND operation, the whole expression may still not be satisfiable. It could be that in order to satisfy the left subtree it is necessary to make a variable, say x, *true*, but in order to satisfy the right subtree it is necessary to make the same variable x *false*.

This suggests a more sophisticated divide-and-conquer. Instead of just finding out whether or not a subtree is satisfiable, let's find the set of assignments of *true* and *false* to its variables that satisfy it. In fact, we only need to keep track of the assignments to those variables that also occur in the other "half" of the expression. Then to compute the set of assignments that satisfy the whole expression, we can intersect (in the case of an AND) the sets of assignments that satisfy the two parts.

Have we succeeded in devising a divide-and-conquer solution to Boolean satisfiability? We need to fill in some details, such as how to handle operations other than AND, but, yes, it would work. It would not work very fast, though, because the sets of assignments can get large, in fact exponentially large in terms of the number of variables. So we have a exponential-time divide-and-conquer

[2]With all these problems, there is the "decision version" — *"Is it possible ...?"* — and the full version — *"Is it possible and if so, how ...?"*.

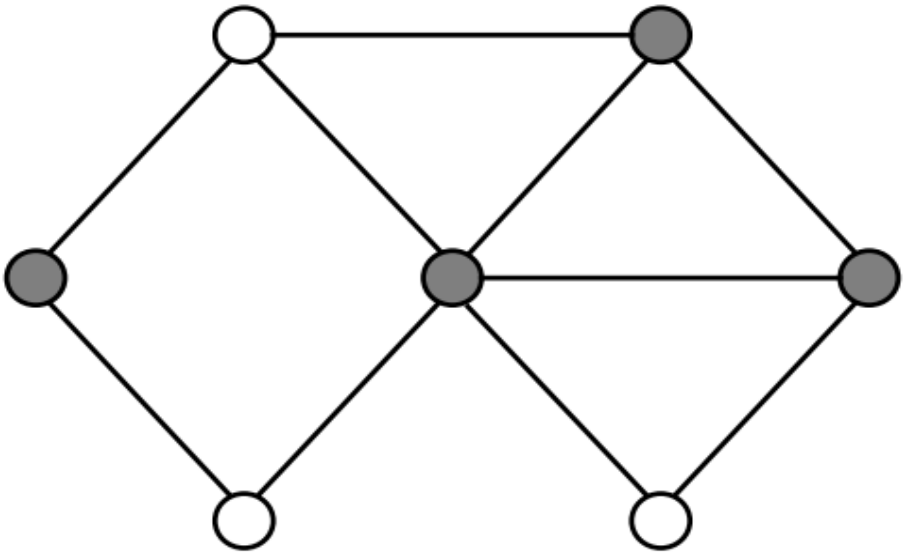

Figure 4.1: Illustrating the VERTEX COVER problem

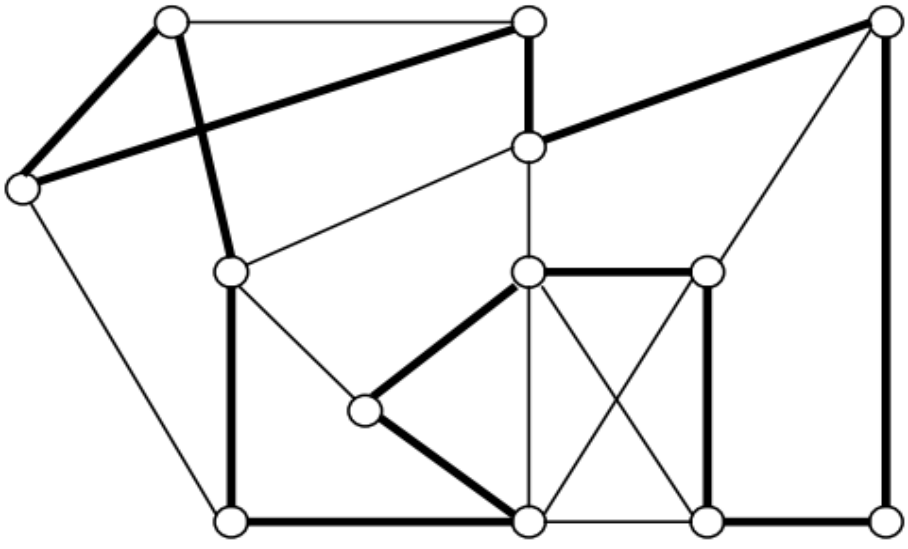

Figure 4.2: Illustrating the HAMILTONIAN CIRCUIT problem

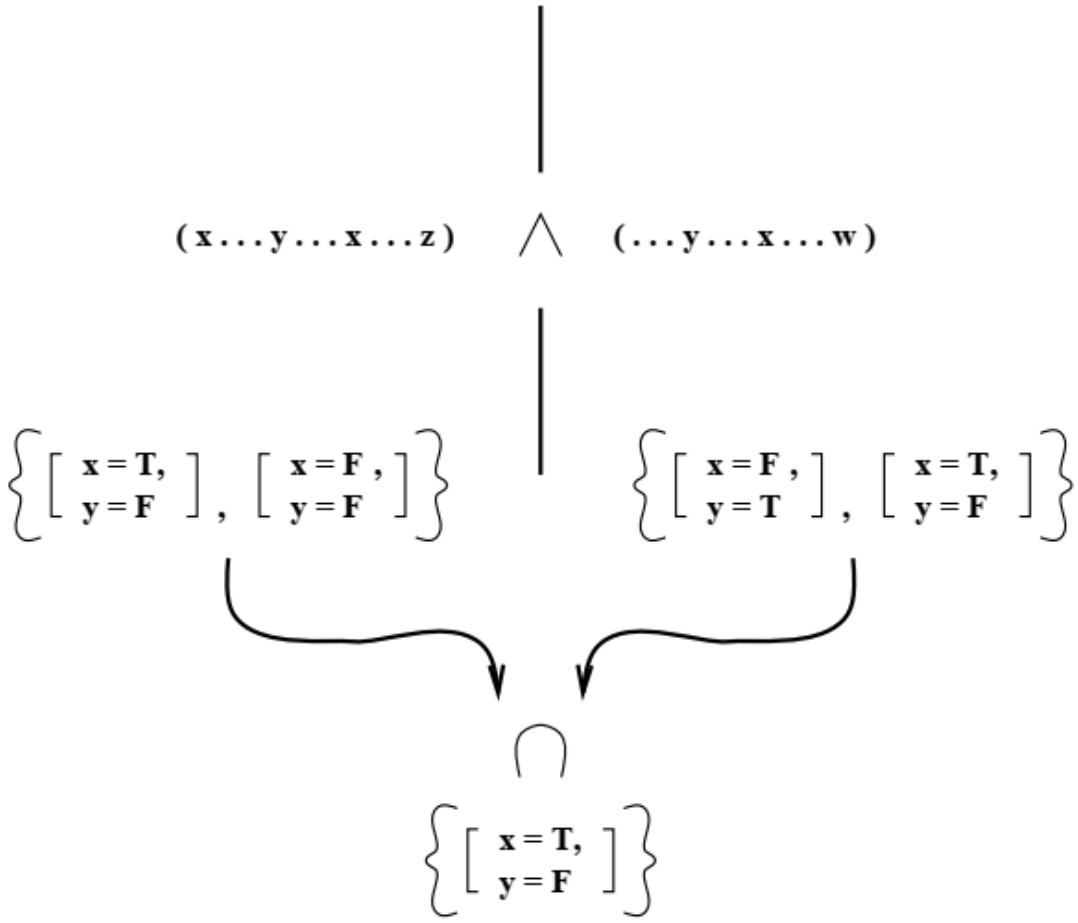

Figure 4.3: Trying to solve SAT with a divide-and-conquer approach

algorithm for SAT. But if we were willing to tolerate exponential running times (which we are not), we could have done a simple exhaustive search in the first place.

What to do? We could try to compute and store those sets more efficiently. To make a long story short, nobody has succeeded in trying to do that. Up to this date, there is no polynomial-time algorithm for SAT, and in fact there is evidence that none might exist. The same is true for the other three problems (3-COLOR, VERTEX COVER, HAMILTONIAN CIRCUIT) and dozens or hundreds[3] of similar "combinatorial optimization" problems. This leads us to developments of the early 1970s.

NP

Nondeterminism

SAT, 3-COLOR and all the other problems are in fact easy to solve, in polynomial time — if we "cheat." The technical term for this form of cheating is "nondeterministic time."

For an example, consider 3-COLOR. A nondeterministic program, as described earlier, may contain a statement like

```
nondet {x[i] = BLUE; x[i] = RED; x[i] = GREEN;}
```

Execution of such a statement causes a three-way branch in the execution sequence, resulting in an execution tree. If this statement is inside a loop that runs `i` from 1 to n, there is successive branching that builds up a tree with 3^n nodes at the end of this loop. Each node at that level in the execution tree corresponds to one assignment of colors to the nodes of the graph. Figure 4.4 shows the shape of such a tree.

Following this loop, we can have a deterministic function that tests if the assignment of colors is a correct 3-coloring. If so, it prints "Yes," if not, it prints "No."

Now is the time to "cheat," in two ways. First, we define the output of the whole tree to be "Yes" if there exists a path at the end of which the program prints "Yes" (and "No" if the program prints "No" at the end of *all* paths). Second, we define the time spent on this computation to be the *height* of the tree. This is the notion of *nondeterministic time*.

With this notion of programs and time, not only 3-COLOR becomes easy to solve in polynomial time, but so do SAT, VERTEX COVER, HAMILTONIAN CIRCUIT and in fact just about any combinatorial optimization problem that you can think of. After all, this nondeterministic time allows us to conduct what amounts to an exhaustive search without paying for it.

Definition *The class of problems that can be solved in nondeterministic polynomial time is called NP.*

[3] depending on whether one counts related problems as separate ones

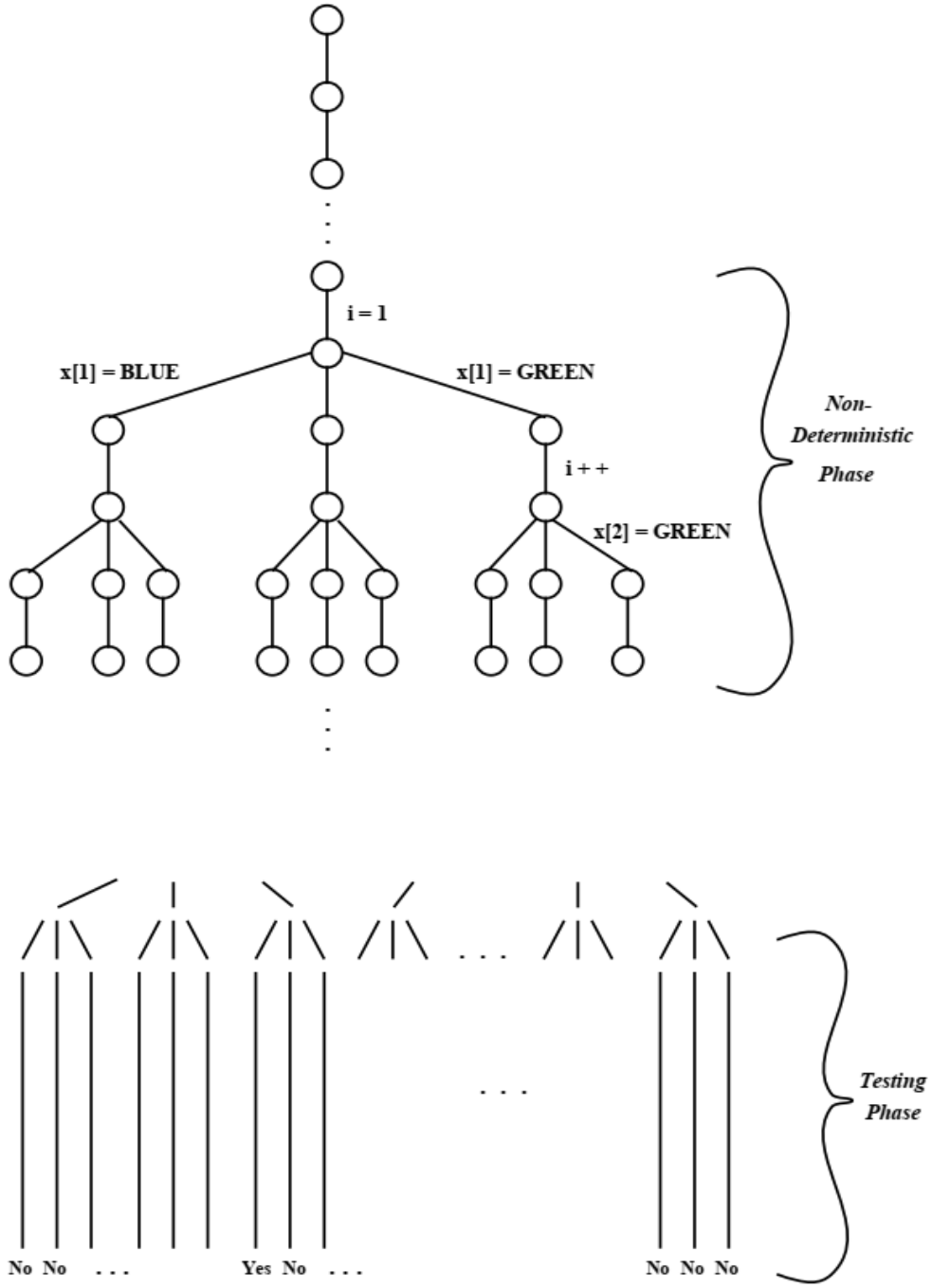

Figure 4.4: The shape of an execution tree for 3-Color

Observation 6 SAT, 3-COLOR, VERTEX COVER, HAMILTONIAN CIRCUIT $\in$ *NP*.[4]

4.3 The NP-completeness of SATISFIABILITY

The big step forward in the early 1970s was to realize that many of these intractable problems were related to each other. For example, as we have seen in Chapter 1, 3-COLOR can be transformed into SAT. Since we are now concerned about running time, it matters how fast that transformation can be carried out, but it is not difficult to verify that

Observation 7 *There is a polynomial-time*[5] *transformation of* 3-COLOR *into* SAT.

As a consequence,

Observation 8 *If we could find a polynomial-time algorithm for* 3-SAT, *we would also have a polynomial-time algorithm for* 3-COLOR.

This last Observation makes 3-SAT the more ambitious of the two problems to solve (or, more precisely, no less ambitious than 3-COLOR). Moreover, what the Observation says about 3-COLOR is also true for VERTEX COVER and HAMILTONIAN CIRCUIT and in fact for all problems in NP:

Theorem 21 (Cook, [Coo71]) *For every decision problem $L \in NP$, there is a polynomial-time transformation of L into* SAT.

The proof of Cook's Theorem needs to be an argument that for any problem in NP there is a polynomial-time transformation into SAT.

In Chapter 1, we did transformations of two different problems into SAT. One was GRAPH 3-COLORING, the other a problem of addition of binary integers. Transforming specific problems into SAT is usually not that hard. The challenge with Cook's Theorem is that we need to argue that there exists a transformation of a problem into SAT without even knowing exactly what that problem is. All we know is that the problem is in NP. As it turns out, there aren't a lot of messy technical details to this proof, just the right use of familiar concepts like compilation and circuits.

CIRCUIT SATISFIABILITY and SATISFIABILITY of Boolean Expressions

What happens when a Boolean circuit executes can be captured by a boolean expression. The value of a wire w of a circuit at time t becomes the value of a variable w_t in the expression, and the operation of a gate in the circuit at time t becomes a boolean operation that defines the correct operation of the gate.

[4] Here we are talking about the "decision versions" of the problems, the versions that ask for a "Yes/No" answer.

[5] honestly, i.e., deterministically, polynomial-time

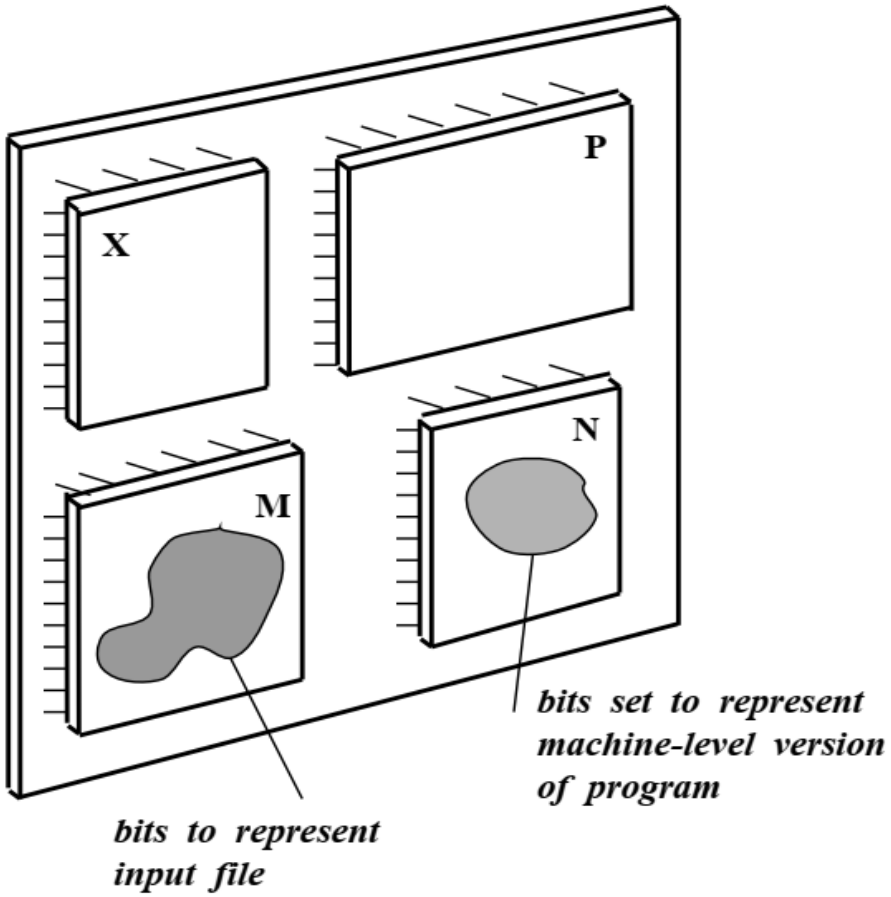

Figure 4.5: A simple computer, just before executing a program with an input file

Thus, at the risk of being repetitive, SATISFIABILITY of Boolean expressions is the problem, given a Boolean expression which may contain constants, of deciding whether or not it is possible to assign values to the variables such that the value of the whole expression becomes *true*.

CIRCUIT SATISFIABILITY is the problem, given a circuit with some constant values given to some wires at some points in time, of deciding whether or not it is possible to assign values to all the wires at all times such that all gates operate correctly at all times.

If you issue a sequence of two commands like

```
cc my.c
a.out < infile
```

then the situation just before the execution of the compiled program is illustrated in Figure 4.5. The program is stored as some pattern of 0s and 1s on

some of the wires, and we might as well assume that the input file has been copied from disk and is also stored on some set of wires.[6]

"Compilation" into a circuit

What happens when you issue the statement

```
cc my.c
a.out < infile
```

to this circuit? The "compiling" process creates the machine-level version of the program `my.c` and stores it by setting some of the wires of the circuit to specific values.

What are we accomplishing with all of this? A transformation. Let's say the program `my.c` is such that it always produces a 0 or a 1 as an answer (or a "Yes" or "No"). Then the question

Does `my.c` on input `infile` produce a "Yes"?

is equivalent to a satisfiability question about our circuit. What question? With certain wires initially set to 0s and 1s to represent the program and the input file, and with the "output wire" set to 1 when the computation terminates, is that whole circuit satisfiable?

This transformation of a question about a program and input file into a satisfiability question — a transformation of software and data into hardware — is most of the proof of Cook's Theorem. In fact, what we have proven so far is that every problem in "P," i.e., every problem that can really (i.e., deterministically) be solved in polynomial time, can be transformed (by a polynomial-time transformation) into CIRCUIT SATISFIABILITY. That result is worthless, but our proof works equally well if we now allow nondeterminism in the program `my.c`.

Enter Nondeterminism

What happens to the preceding "compilation" process if the program `my.c` is nondeterministic? What if `my.c` had a statement

```
nondet {x = TRUE; x = FALSE;}
```

How would we compile this into our circuit? Much like an `if`-statement

```
if ...
    x = TRUE;
else
    x = FALSE;
```

except that we let the "choice" between the two options depend on a wire that is not the output of any gate. This is a wire which in the circuit satisfiability problem that is being constructed is simply not assigned a value. All we need is

[6] Otherwise we would need to draw a disk and turn our neat "circuit" satisfiability problem into a "circuit–and–disk" satisfiability problem.

one such wire. Then compiling a two-way nondeterministic branch is just like an `if`-statement with the choice being determined by the value of that wire at the time of the execution of the statement.