Illinois Institute of Technology
Department of Computer Science

Lectures 22–24: April 11–18, 2016

CS 430 Introduction to Algorithms
Spring Semester, 2016

Clarke's Second Law: The only way to find the limits of the possible is by going beyond them to the impossible.

—Arthur C. Clarke

When you come to a fork in the road, take it.

—Yogi Berra

1 NP-completeness

Over the past few months we have developed and examined algorithms in many problem domains. Although these algorithms have been of varying efficiencies, they have all run in polynomial time—that is, in time $O(n^k)$ for some constant k. Do any problems require more than polynomial time?

Yes. However, more interesting than problems that are known to require superpolynomial time are the so-called NP-complete problems,[1] whose status is unknown—it is not known whether NP-complete problems can be solved in polynomial time or not. Although the complexity of NP-complete problems is not known, if it could be shown that a single NP-complete problem could be solved in polynomial time, then all NP-complete problems could be solved in polynomial time. Conversely, if it were shown that a single NP-complete problem could not be solved in polynomial time, then no NP-complete problem could be solved in polynomial time. Many problems in diverse areas have been shown to be NP-complete, including graph-theoretic problems, number-theoretic problems, scheduling problems, and even narrow tree drawing. Hence the $\mathrm{P} \stackrel{?}{=} \mathrm{NP}$ question is perhaps the most significant open question in computer science.

1.1 Decision Problems

Before proceeding further it is important to understand how we talk about problems in the context of complexity. For the purposes of the study of NP-completeness we restrict our attention to *decision problems*. A decision problem asks a yes/no question about some *input*, or *instance*. For example, given an integer n, we could ask if n is a composite number and expect an answer of either "yes" or "no."

Many problems are not naturally expressed as decision problems. For example, one problem we recently dealt with was that of finding a shortest path in a graph between two vertices u and v. Although this problem is not a decision problem, it can be easily recast as one: "Given an integer k and a graph G with two distinguished vertices u and v does there exist a path from u to v of length at most k?" This technique can be used to transform many optimization problems to decision problems.

[1] NP-complete problems are studied in depth, in a somewhat different fashion, in chapters 34 and 35 of CLRS.

1.2 Polynomial-time solvability and the class P

We define the complexity class P to be the set of decision problems that can be solved in time $O(n^k)$ for some constant k.[2] This class, then, includes all of the problems we have studied so far in this course (or, actually, the decision problems corresponding to them).

1.3 Polynomial-time verifiability and the class NP

It is useful to think about algorithms for verifying solutions to problems. For instance, consider the problem of determining if a number is composite. Let us not think about algorithms for solving the problem from scratch; instead, say that I claim that some number n is composite but that you don't believe me. I can provide you with a *certificate*, or piece of evidence, namely a set of factors of n. You can then verify, in polynomial time, that the product of those factors is n and, hence, that n is indeed composite.

In general, a problem can be *verified in polynomial time* if there exists some polynomial-length certificate that can be used to verify an answer of "yes" to the problem in polynomial time. We can then define the class NP as the set of problems that can be verified in polynomial time.[3] Note that answers of "no" need not be verifiable in polynomial time, only answers of "yes."

Certain problems, though, have "no" answers that *can* be verified in polynomial time. We call the class containing these problems co-NP. An alternate definition of co-NP is that it consists of the *complements* of problems in NP, where the complement of a problem Q is simply Q with its yes/no outputs reversed. For instance, the problem of determining if a number is prime is in co-NP, since it is the complement of the problem of determining if a number is composite, which we have already shown to be in NP.

1.4 Reductions and Hardness

Crucial to the study of NP-completeness is the notion of reductions. Intuitively, we can *reduce* a problem Q to another problem Q' if we can take an instance of Q and quickly "dress it up" as an instance of Q' that will correctly solve the original problem Q. More formally, we say that Q is *polynomial-time reducible* to Q', or that Q' is *harder* than Q, written $Q \leq_{\mathrm{P}} Q'$, if there is some function f that can be computed in polynomial time such that $Q(x) = Q'(f(x))$ for all instances x.

1.5 NP-hardness and NP-completeness

As we will see, it is useful to deal with problems that are at least as hard as all problems in the class NP. We say that a problem Q is *NP-hard* if it is harder than every problem in NP—formally, Q is NP-hard if and only if $Q' \leq_{\mathrm{P}} Q$ for all $Q' \in \mathrm{NP}$.

Finally, we define NP-completeness. A problem Q is *NP-complete* if $Q \in \mathrm{NP}$ and Q is NP-hard. That is, the NP-complete problems can be thought of as the hardest problems in NP.

It is not difficult to see that if any NP-complete problem is solvable in polynomial time, then P = NP. Say that Q is such a problem, so $Q \in P$ and Q is NP-complete. The definition of NP-completeness requires that $Q \leq_{\mathrm{P}} Q'$ for every $Q' \in \mathrm{NP}$. Clearly, then, every such Q' is also in P.

[2] Note that a problem is in P if some polynomial-time algorithm exists to solve the problem, not that you personally know of one. Your inability to find a polynomial-time algorithm is not evidence that a problem is not in P; it may be that you just have not found one yet.

[3] The class NP can be defined in various ways; in particular, the name NP stands for "nondeterministic polynomial-time," and indeed NP is the set of problems that can be solved in polynomial time on a nondeterministic machine. Although these define the same set, the notion of polynomial-time verifications is more useful here.

We can also prove that if any problem in NP is *not* solvable in polynomial time, then *no* NP-complete problem is solvable in polynomial time. Consider the NP-complete problem $Q \in \text{NP} - \text{P}$. Say that some other problem $Q' \in P$ is NP-complete. In that case, since $Q \leq_\text{P} Q'$, clearly $Q \in \text{P}$, contradicting the assumption.

This leads to the following strategy for proving that some problem Q is NP-complete:

- Prove that $Q \in \text{NP}$. This can be done by showing how Q can be verified in polynomial time. This step is generally fairly simple, but it can be easy to forget.
- Prove that Q is NP-hard. This can be done by selecting some problem Q' that is known to be NP-complete and showing that $Q' \leq_\text{P} Q$—that, given an instance of Q', it can be dressed up as an equivalent instance of Q in polynomial time.[4] Notice that showing instead that $Q \leq_\text{P} Q'$ gives no useful information; it is crucial to perform the reduction in the correct direction. It is often not immediately obvious which NP-complete problem to select as Q'. Theoretically, if a reduction exists from one NP-complete problem, it exists from all others as well, but there is usually one "most appropriate" NP-complete problem that makes the reduction relatively simple.

1.6 Bootstrapping: proving the NP-completeness of circuit satisfiability

At this stage, we cannot yet use the above strategy, since we have not yet established the NP-completeness of any problem. Cook (1971) proved that the problem of satisfiability (defined below) is NP-complete without appealing to a reduction from another NP-complete problem. We instead sketch a proof that the similar problem of circuit satisfiability (CIRCUIT-SAT) is NP-complete.

A *satisfying assignment* for a boolean combinational circuit is some set of boolean input values that causes the output of the circuit to be 1. A circuit is *satisfiable* if it has some satisfying assignment. Given a boolean combinational circuit C consisting of AND, OR, and NOT gates, CIRCUIT-SAT returns "yes" if C is satisfiable and "no" otherwise. Our goal is to prove that CIRCUIT-SAT is NP-complete.

Recall that for CIRCUIT-SAT to be NP-complete, it must be included in the class NP and it must be NP-hard. First we prove that CIRCUIT-SAT $\in$ NP. This can be done by showing that there is a polynomial-time algorithm to verify a "yes" solution given a polynomial-length certificate. In this case, the certificate is a satisfying assignment, and the algorithm simply evaluates the circuit using the satisfying assignment as inputs; if the solution is correct, the output must be 1. Clearly this algorithm takes polynomial time.

A formal proof that CIRCUIT-SAT is NP-hard is beyond the scope of this course, but we sketch an argument here. The text (Lemma 36.6) gives a slightly more detailed version of this argument, but the proof itself requires the formal language material covered in CS 375. For CIRCUIT-SAT to be NP-hard, there must be reductions from all problems in NP to CIRCUIT-SAT. It seems plausible that any problem in NP can be solved on a computer. How can an instance of an arbitrary problem Q be "dressed up" as a CIRCUIT-SAT instance? Here we rely on the fact that every computer is physically realized as a large boolean combinational circuit. Thus, we build a circuit that outputs 1 if the input is a correct solution to Q and outputs 0 otherwise, basically encoding the algorithm the computer would have followed in the circuit. If this circuit is satisfiable, then some solution must exist.

[4] It is sufficient to prove $Q' \leq_\text{P} Q$ for any NP-complete problem Q' since we already know that any other problem in NP is reducible to Q' and the $\leq_\text{P}$ relation is transitive.

1.7 Further NP-completeness results

Now that we have found an NP-complete problem, we can build a chain of NP-completeness results from it. As soon as we determine that a problem is NP-complete, we can use it to prove other problems NP-complete, so it is in our interests to generate a reasonably large set.

1.7.1 Satisfiability

The problem of *(formula) satisfiability* (SAT) is similar to the problem of circuit satisfiability. Given some boolean formula ϕ consisting of boolean variables $x_1, x_2, \ldots$, boolean connectives $\wedge$, $\vee$, $\rightarrow$, and $\leftrightarrow$, and parentheses, ϕ is *satisfiable* if there is some assignment of 0 and 1 to each of the variables x_i under which ϕ evaluates to 1. Given some such ϕ, SAT asks if ϕ is satisfiable.

For SAT to be NP-complete, we must show that SAT $\in$ NP and that SAT is NP-hard. To verify a "yes" solution, we use a satisfying assignment as a certificate; in polynomial time, we can easily plug the assignment into the formula and verify that it indeed evaluates to 1.

To prove that SAT is NP-hard, we reduce it from CIRCUIT-SAT. The idea behind the reduction is to assign a variable to each wire in the circuit and to express correct operation of each wire in ϕ. The full reduction is given in the text.

1.7.2 3-CNF satisfiability

A formula ϕ is said to be in *3-conjunctive normal form*, or *3-CNF*, if it is of the form $(l_{11} \vee l_{12} \vee l_{13}) \wedge (l_{21} \vee l_{22} \vee l_{23}) \wedge \cdots \wedge (l_{n1} \vee l_{n2} \vee l_{n3})$, where l_{ij} is any literal (variable or negated variable) l_{i1}, l_{i2}, and l_{i3} are distinct for any i—that is, if ϕ is made up of the conjunction of any number of clauses, where each clause is the disjunction of exactly three distinct literals. The 3-CNF-SAT problem is identical to the SAT problem, only we restrict the input to a formula in 3-CNF.

The containment of 3-CNF-SAT in NP follows directly from our proof that SAT $\in$ NP. To establish the NP-completeness of 3-CNF-SAT, then, we need only prove its NP-hardness. That is, for a given formula ϕ, we must show how to construct some formula ϕ' in 3-CNF such that ϕ' is satisfiable if and only if ϕ is satisfiable. This can be done by first forming a binary parse tree for ϕ (see figure 36.9 on page 943 of the text) and then representing each node of the tree with a clause. This produces a formula that is a conjunction of clauses, each of which has at most three literals, but the clauses are not disjunctions of literals. A technique for creating these clauses and converting them to 3-CNF form is given in the text.

1.7.3 Graph 3-coloring

The graph 3-coloring problem is

Input: An undirected graph $G = (V, E)$.

Output: Is there a a coloring

$$c : V \rightarrow \{\text{red}, \text{blue}, \text{green}\}$$

such that for every edge e in E the vertices joined by e are not colored with the same color?

Construct a 2-or-3-SAT Boolean expression from a graph 3-coloring problem G as follows.

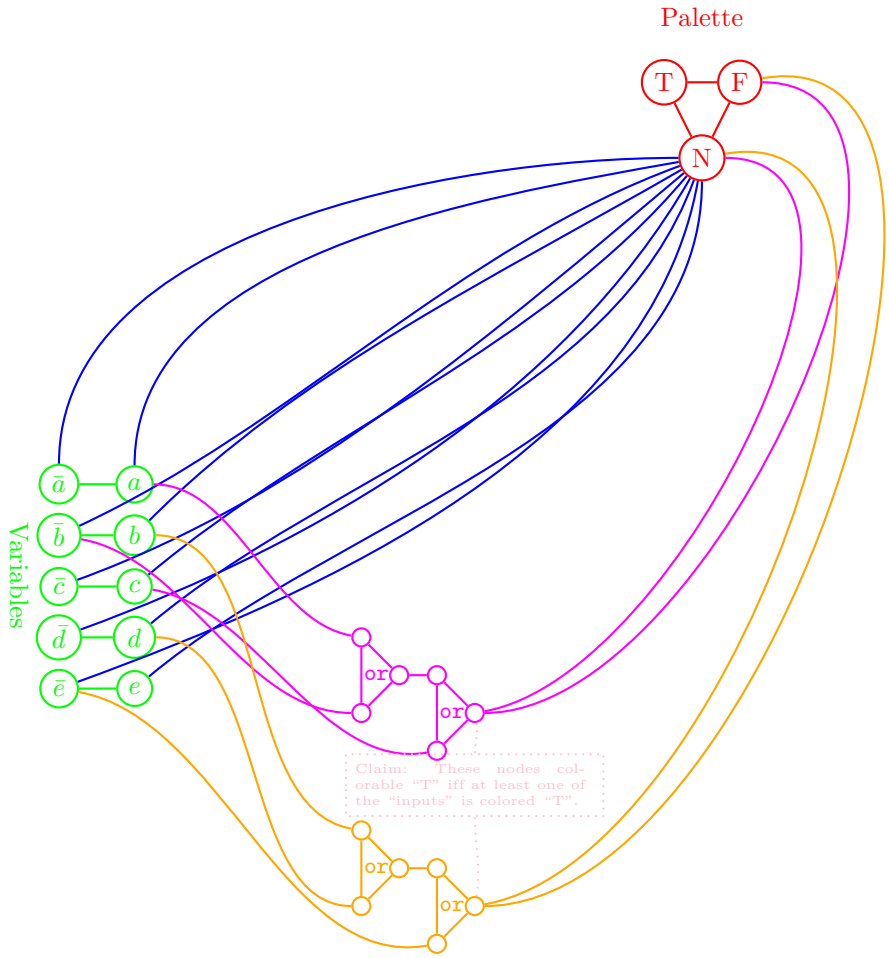
- For each *vertex* v_i include a subexpression

$$\begin{aligned}&(R_i \vee B_i \vee G_i) \wedge (\overline{R_i \wedge G_i}) \wedge (\overline{R_i \wedge B_i}) \wedge (\overline{B_i \wedge G_i}) \\ =\ &(R_i \vee B_i \vee G_i) \wedge (\bar{R}_i \vee \bar{G}_i) \wedge (\bar{R}_i \vee \bar{B}_i) \wedge (\bar{B}_i \vee \bar{G}_i)\end{aligned}$$

- For an *edge* e connecting v_i and v_j include a subexpression

$$\begin{aligned}&(\overline{R_i \wedge R_j}) \wedge (\overline{G_i \wedge G_j}) \wedge (\overline{B_i \wedge B_j}) \\ &= (\bar{R}_i \vee \bar{R}_j) \wedge (\bar{G}_i \vee \bar{G}_j) \wedge (\bar{B}_i \vee \bar{B}_j)\end{aligned}$$

- Replace each 2-literal term $(a \vee b)$ with $(a \vee b \vee t) \wedge (a \vee b \vee \bar{t})$ for a new variable t.
- Construct a graph G from the 3-SAT expression as shown by the example below for the 3-SAT expression $(a \vee \bar{b} \vee c) \wedge (b \vee d \vee \bar{e})$:



1.7.4 Clique

Given an undirected graph $G = (V, E)$, recall that a *clique* is some subset of vertices $V' \subseteq V$ that is a complete subgraph of G—each pair of vertices in V' is connected by some edge in E. For a graph G and a

positive integer k, the CLIQUE problem asks if G contains a clique of size k.[5]

Clearly CLIQUE $\in$ NP—given a set of edges V' and an integer k, it is easy to verify that V' is a clique of size k. To prove NP-hardness of CLIQUE, we reduce from 3-CNF-SAT. Given a 3-CNF formula $\phi = C_1 \wedge \cdots \wedge C_n$, ϕ is satisfiable if and only if at least one literal from each clause C_i can be set to 1 without creating a contradiction (that is, without setting a literal and its negation both to 1). If we represent each literal as a vertex and we insert edges between vertices representing literals in different clauses which are not negations of each other, then a clique of size equal to the number of clauses in ϕ exists if and only if ϕ is satisfiable.

1.7.5 Vertex cover

Given an undirected graph $G = (V, E)$, a *vertex cover* is some subset of vertices $V' \subseteq V$ such that if $(u, v) \in E$, then $u \in V'$ or $v \in V'$. For a graph G and a positive integer k, the VERTEX-COVER problem asks if G contains a vertex cover of size k.

Proving VERTEX-COVER $\in$ NP is simple, as above. A reduction from CLIQUE to VERTEX-COVER is not difficult to see. Given an undirected graph $G = (V, E)$, its *complement* is defined as $\overline{G} = (V, \overline{E})$, where $\overline{E} = \{(u, v) \mid (u, v) \notin E\}$. It can be easily confirmed that G has a clique of size k if and only if $\overline{G}$ has a vertex cover of size $|V| - k$.

1.7.6 Subset-sum

Given a finite set of natural numbers S and some number t, SUBSET-SUM asks if there is some subset $S' \subseteq S$ such that $t = \sum_{s \in S'} s$, in other words, "is there is some subset of S whose elements sum to t?" The verification algorithm proving that SUBSET-SUM $\in$ NP is simple if we take the certificate to be the subset itself. The reduction from VERTEX-COVER is covered in CLRS, pages 1097–1100.

1.7.7 Narrow tree drawing

Recall the discussion of tree drawing in the first lecture. While we arrived at an algorithm to draw trees, it was under the stipulation that they need not be drawn as narrowly as possible. Now we require this. Given a tree T and a width w, the TREE-DRAWING problem asks if it is possible to draw T in width w.

It is easy to verify a proposed solution, so TREE-DRAWING $\in$ NP. The reduction from 3-CNF-SAT is complex, and relies on the encoding of clauses of a formula in 3-CNF with portions of a tree. The reduction is given in K.J. Supowit and E.M. Reingold, "The Complexity of Drawing Trees Nicely," *Acta Informatica* 18 (1982), pp. 377-392.

[5] The size of a clique is the number of vertices it contains. A more natural form of the problem asks for the largest clique in a graph, but we are considering here only decision problems.