EE360C: Algorithms

NP-Completeness

Summer 2019

Department of Electrical and Computer Engineering
University of Texas at Austin

Introduction

Recap

Up until now, we've talked mainly about polynomial time algorithms. But not all problems can be solved in polynomial time.

Coming Up

But there's also a class of problems *for which we don't know*. We know we can verify that a solution is correct in polynomial time, but we don't know if we can solve them in polynomial time.

These are the "NP-complete" problems. This has led to the $P \neq NP$ question of computer science lore

- But the kicker is... if you can solve one of the NP-Complete problems in polynomial time, you can solve them all!

NP Complete Problems

OK. The NP-Complete problems are clearly hard, so they should *look* hard, right?

Shortest vs. Longest Simple Paths

We know how to solve the shortest path problem in polynomial time. Finding the *longest* simple path is much more difficult. In fact, just writing an algorithm to determine if a graph has a path of k edges is an NP-Complete problem.

Euler tour vs. Hamiltonian tour

A Euler tour contains every edge in a graph; a Hamiltonian tour contains every vertex in the graph. We can determine whether a graph has a Euler tour in $O(E)$ time; determining whether a graph has a Hamiltonian tour is NP-Complete.

The Classes P, NP, and NPC

The Class P

The class P contains problems that are solvable in polynomial time (specifically, in $O(n^k)$, for instance size n and some constant k).

The Class *NP*

The class NP contains problems whose solutions can be verified in polynomial time (e.g., given a purported 3-coloring of a graph, we can determine that it is or is not correct using DFS)

A problem is in NPC if it is in NP and is at least as hard as any other problem in NPC (it's unlikely that the problem could be solved in polynomial time   If, as an algorithm designer, you encounter a problem that is NP-complete, it's likely better to focus your attention on an approximation algorithm...

A Summary

- **P**: problems that can be solved in polynomial time.
- **NP**: problems that can be verified in polynomial time (*nondeterministic polynomial*). Clearly $P \subseteq$ NP.
- **NP-Complete**: decision problems (those whose answers are "yes" or "no") that can be verified in polynomial time, but for which no known polynomial time solution is known. NP-Complete $\subseteq$ NP.
- **NP-Hard**: problems that are at least as hard as the hardest problem in NP
    - A problem H is in NP-Hard iff there exists a problem $L \in$ NP-complete that is polynomial time reducible to H.
    - That is, L can be solved in polynomial time given an oracle that can solve H.

Showing a Problem to be NP-Complete

We're turning the tables here a little bit... In showing a problem is NP-complete, we're not trying to design an efficient implementation but to show that one is unlikely to exist

Decision Problems vs. Optimization Problems

We're going to focus on decision problems (with a yes/no answer) instead of optimization problems

- decision problems are closely related to optimization problems (e.g., given a graph G, vertices u, v, and integer k, does there exist a path of length $\leq k$ from u to v?)
- the optimization version is at least as hard as the decision version; if the decision problem is hard, it implies that the optimization problem is hard

Showing a Problem to be NP-Complete (cont.)

To demonstrate a problem is NP-Complete, the key ingredient is a *reduction* that shows that the new problem is at least as hard as a problem known to be NP-complete.

- consider an instance of a decision problem *A* and another decision problem *B* for which we have a polynomial time algorithm
- suppose we have a procedure to transform any instance α of *A* into an instance β of *B* such that
  - the transformation takes polynomial time
  - the answer to β is yes if and only if the answer to α is yes
- this gives us a polynomial time algorithm for solving *A*

Suppose there ends up not to be a polynomial time algorithm for *A*, and *A* polynomial-time reduces to *B*, then there is no polynomial time procedure for *B*.

Polynomial-time reducibility

Polynomial-time reducibility

Intuitively, a problem Y can be reduced to another problem X if any instance of Y can be easily rephrased as an instance of X, and the solution to X provides a solution to the instance of Y.

- consider the problem Y of solving linear equations (e.g., $ax + b = 0$). We can easily rephrase this as the problem X of solving quadratic equations (e.g., $0x^2 + ax + b = 0$), where if we solve the latter, we have a solution to the former.

Claim

Problem X is at least as hard as problem Y and write $Y \leq_P X$ (formally, Y is polynomial-time reducible to X) if arbitrary instances of problem Y can be solved using a polynomial number of standard computational steps, plus a polynomial number of calls to a black-box that solves problem X.

Corollary

If X, Y are problems such that $Y \leq_p X$, then $X \in P$ implies $Y \in P$.

Example of Polynomial-time Reducibility

Independent Set

An **independent set** in an undirected graph $G = (V, E)$ is a subset $V' \subseteq V$ of vertices with no edges between them. The **size** of an independent set is the number of vertices it includes.

The **independent set problem** is the optimization problem of finding an independent set of maximum size in a graph. As a decision problem, we ask if an independent set of a given size at least k exists in the graph.

Example of Polynomial-time Reducibility

Vertex Cover

A **vertex-cover** of an undirected graph $G = (V, E)$ is a subset $V' \subseteq V$ such that if $(u, v) \in E$, then $u \in V'$ or $v \in V'$ (or both). The **size** of a vertex cover is the number of vertices in it.

The vertex-cover problem is to find a vertex cover of minimum size in a given graph. The decision version of this problem is to determine whether a graph has a vertex cover of a given size at most k.

Example of Polynomial-time Reducibility

Let $G = (V, E)$ be a graph. Then S is an independent set if and only if it's complement $V - S$ is a vertex cover

Proof

Suppose S is an independent set, consider any edge $e = (u, v)$. Since, S is independent both u and v cannot be in S; one of them must be in $V - S$. It follows that every edge has at least one end in $V - S$, and so $V - S$ is a vertex cover.

Conversely, suppose $V - S$ is a vertex cover. Consider any two nodes u and v in S. If there were joined by an edge e, then neither end of e would lie in $V - S$, contradicting our assumption that $V - S$ is a vertex cover. It follows that no two nodes in S are joined by an edge, so S is an independent set.

Example of Polynomial-time Reducibility

Independent Set $\leq_P$ Vertex Set

Proof: If we have a black box to solve Vertex Set, then we can decide whether G has an independent set of size at least k by asking the black box whether G has a vertex cover of size at most $n - k$.

Vertex Set $\leq_P$ Independent Set

Proof: If we have a black box to solve Independent Set, then we can decide whether G has an vertex cover of size at least k by asking the black box whether G has a independent set of size at most $n - k$.

Showing a Problem to be NP-Complete (cont.)

To show a problem is NP-Complete, we'll reduce a known NP-Complete problem to it.

- We also have to show that it's in NP, but that's usually pretty easy.

We're also going to need a "first" NP-complete problem to start the reductions. We'll see that in a little bit...

Polynomial-time verification (NP)

Showing Membership in NP

What did it mean to be a member of NP? That a certificate to a *true instance* (i.e., a "yes" instance) of a problem can be verified in polynomial time.

- For example, consider the problem of determining whether a graph has a path of length $\leq k$.
- Given a purported solution path p, we can easily "walk" p and count its edges in polynomial time.

Hamiltonian Cycle

A **Hamiltonian cycle** of an undirected graph is a simple cycle that contains every vertex in V.

- What's the certificate for the Hamiltonian cycle?
- just the cycle itself!
- it's easy to efficiently verify that it touches all vertices...

NP or not NP?

Problems in NP

- anything in P (e.g., does there exist a subsequence of size k for sequences X and Y?)
- does there exist a 3-coloring of graph G?
- is there a partition of a set of integers S into two subsets S_0 and S_1 such that the sum of elements in S_0 is equal to the sum of elements in S_1?

Problems Unlikely to be in NP

- counting problems
- problems that involve $\forall x . \exists y$
- provably intractable/undecidable problems

Nothing is known about the relationships between P, NP, co-NP, NP-complete, etc.

NP-Completeness

NP-Completeness

Definition

A problem X is **NP-complete** if

1. $X \in NP$, and
2. $Y \leq_P X$ for every problem $Y \in$ NP.

If a problem satisfies the second property but not necessarily the first, then we say it is **NP-hard**.

The following theorem is a direct result:

Theorem

If any NP-complete problem is polynomial time solvable, then $P = NP$. Equivalently, if any problem in NP-complete is not polynomial time solvable, then no NP-complete problem is polynomial time solvable.

NP-Completeness

Most people believe that $P \neq NP$ simply because we've known about the NP-complete problems for a long time and no one has come up with an efficient solution for them.

- again, if we can solve any one of the NP-complete problems in polynomial time, we can solve them all in polynomial time
- on the other hand, if we can *prove* that no polynomial time algorithm exists for an NP-complete problem, then no algorithm exists for any of them

So How Do I Do It?

The following is the general recipe for an NP-complete proof for showing a new problem X is NP-complete:

1. Prove that $X \in \mathrm{NP}$.
2. Select a known NP-complete problem Y.
3. Prove that $Y \leq_P X$, namely: consider an arbitrary instance s_Y of problem Y, and show how to construct, in polynomial time, an instance s_X of problem X that satisfies the following properties:
    - If s_Y is a "yes" instance of Y, then s_X is a "yes" instance of X;
    - If s_X is a "yes" instance of X, then s_Y is a "yes" instance of Y.

   (In other words, the instances s_X and s_Y have the same answer.)

Our First NP-Complete problem: Circuit-SAT

That First Problem

Once we have one NP-complete problem, we can reduce others to it. Our first problem is Circuit-Satisfiability, or Circuit-SAT. The proof that Circuit SAT is NP-complete is quite complex; we'll look at the intuition.

Circuit Satisfiability Problem

A boolean combinational circuit is a collection of boolean combinational elements connected by wires.

- we assume three types of gates: NOT, AND, OR
- no cycles are allowed
- there are some *circuit inputs* and *circuit outputs*

The truth assignment for a circuit is a set of boolean input values.

A one-output boolean combinational circuit is **satisfiable** if it has a **satisfying assignment**: a truth assignment that causes the circuit's output to be 1.

Circuit Satisfiability Problem

The circuit satisfiability problem is:

- Given a boolean combinational circuit composed of AND, OR, and NOT gates, is it satisfiable?

This would be a nice problem to be able to solve efficiently; if there is a sub-circuit that is not satisfiable (i.e., always generates a 0 output), it can be optimized out and replaced with a “constant” 0

The Obvious Solution

The obvious way to solve this is to try all of the possible inputs.
But this is $O(2^k)$ for k inputs.

Can we do better?

Circuit SAT is in NP

Lemma

The circuit satisfiability problem belongs to the class NP.

Proof

Given a certificate (i.e., a purported correct input), we can easily verify its correctness in polynomial time just by running the circuit on the input.

Circuit SAT is NP-Complete

Lemma

The circuit satisfiability problem is NP-Complete.

Proof (Intuition)

To perform this proof, we show that *every* language L in NP-complete can be polynomial time reduced to Circuit-SAT (using the definition of NP-complete). If L is in NP-complete, then there is a verification algorithm A that runs in time $O(n^k)$, assuming strings in L are of length n. We construct a single boolean circuit M that maps one "configuration" of a machine that computes A (recording, e.g., memory state, program counter, etc.) to the next "configuration." We hook together $O(n^k)$ of these circuits (because after all, that's the maximum possible number of steps required to compute A). This big, composed circuit ($C(x)$) is satisfiable by a certificate y (associated with input x) if and only if x was accepted in L. By lots more hand-waving, the size of this circuit is polynomial in n, and the transformation can be done in polynomial time.

Examples

Formula Satisfiability (SAT)

This is actually the first problem ever shown to be NP-complete.

The Problem

An instance of SAT is a boolean formula ϕ composed of:

- n boolean variables: $x_1, x_2, \ldots, x_n$
- m boolean connectives: any boolean function with one or two inputs and one output, such as AND ($\wedge$), OR ($\vee$), NOT ($\neg$), implication ($\rightarrow$), if and only if ($\leftrightarrow$)
- parentheses (we assume there are no redundant parentheses)

A **truth assignment** for ϕ is a set of values for the variables; a **satisfying assignment** is one that returns true. A formula with a satisfying assignment is **satisfiable**.

SAT asks whether a formula is satisfiable.

The Naïve SAT Algorithm

What's the naïve way to solve SAT?

Easy Solution

Enumerate all of the possible assignments for the variables and test to see if any of them return true.

Easy is Bad

What's the running time? $O(2^n)$ for a formula with n variables.

SAT is NP-Complete

Theorem

Satisfiability of boolean formulas is NP-complete.

Proof: Part I

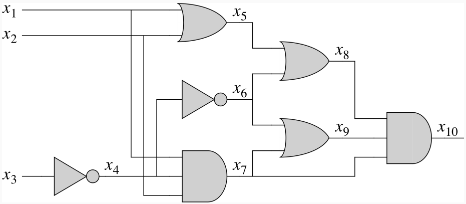
SAT$\in$NP. The certificate consisting of a satisfying assignment for ϕ can be verified in polynomial time by replacing each variable with its value in the input and then evaluating the expression.

SAT is NP-Complete (cont.)

Proof: Part II

We show that CIRCUIT-SAT$\leq_P$SAT, i.e., that any instance of circuit satisfiability can be reduced in polynomial time to an instance of SAT. We might think we can just replace any boolean circuit as a boolean formula. Unfortunately, when gates have large fanouts, the formula can grow exponentially. Instead, in addition to having variables from inputs of each the circuit, we create new variables, one per gate in the circuit. We encode the functioning of each gate by a small formula, e.g., an AND gate with inputs x and y and output variable w is encoded as $(w \leftrightarrow (x \wedge y))$. Then we take the conjunction of all of these formulas and in addition z, where z is the output of the circuit. The resulting formula is satisfiable if and only if the circuit is satisfiable. If the circuit has a satisfying assignment, each wire of the circuit has a well-defined value, and the circuit's output is 1. Therefore the assignment of wire values to variables in ϕ makes each clause of ϕ evaluate to 1, and thus the conjunction evaluates to 1.

SAT Reduction Example



$$
\begin{aligned}
\Phi = x_{10} \quad & \wedge \quad (x_4 \leftrightarrow \neg x_3) \\
& \wedge \quad (x_5 \leftrightarrow (x_1 \vee x_2)) \\
& \wedge \quad (x_6 \leftrightarrow \neg x_4) \\
& \wedge \quad (x_7 \leftrightarrow (x_1 \wedge x_2 \wedge x_4)) \\
& \wedge \quad (x_8 \leftrightarrow (x_5 \vee x_6)) \\
& \wedge \quad (x_9 \leftrightarrow (x_6 \vee x_7)) \\
& \wedge \quad (x_{10} \leftrightarrow (x_7 \wedge x_8 \wedge x_9))
\end{aligned}
$$

3-SAT

Reducing from SAT can be painful because the reduction has to handle any input formula. So it's useful to reduce from a more restricted language. That's what 3-SAT is used for.

A **literal** in a boolean formula is an occurrence of a variable or its negation. A formula is in **conjunctive normal form** if it expressed as an AND of clauses, each of which is an OR of one or more literals.

3-SAT

A boolean formula is an instance of **3-SAT** if each clause has exactly three distinct literals. An example:

$$(x_1 \vee \neg x_1 \vee \neg x_2) \wedge (x_3 \vee x_2 \vee x_4) \wedge (\neg x_1 \vee \neg x_3 \vee \neg x_4)$$

3-SAT is NP-Complete

Theorem

3-SAT is NP-Complete.

Proof: Part I

The same argument that SAT$\in$NP can be given as was given for SAT$\in$NP (i.e., 3-SAT formulas are just special cases of formulas)

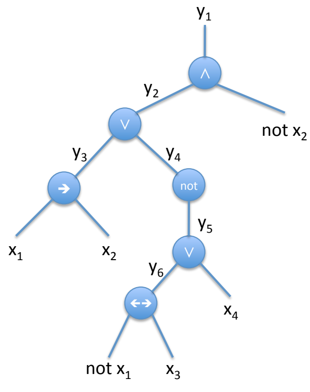
Proof: Part II

First, given a formula ϕ, construct a “parse” tree for ϕ with literals as leaves and connectives as internal nodes. This parse tree is basically a circuit built directly from ϕ. A node in the tree can have at most 2 inputs; we can adjust ϕ slightly to account for this by adding parenthesizations. We introduce a variable y_i for the output of each node.

3-SAT is NP-Complete (cont.)

For example, the tree for the formula:

$$\phi = ((x_1 \rightarrow x_2) \vee \neg((\neg x_1 \leftrightarrow x_3) \vee x_4)) \wedge \neg x_2$$

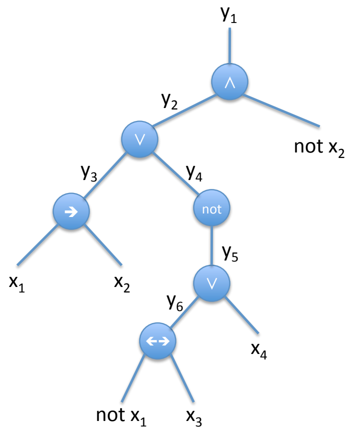


3-SAT is NP-Complete (cont.)

Proof: Part II

We rewrite the original formula ϕ as the AND of the root variable and the conjunction of clauses describing the operation at each node. Call this new formula ϕ'. ϕ' is satisfiable if and only if ϕ is satisfiable. Each clause in ϕ' has no more than 3 literals.

3-SAT is NP-Complete (cont.)



$$
\begin{aligned}
\phi' = y_1 \quad & \wedge \quad (y_1 \leftrightarrow (y_2 \wedge \neg x_2)) \\
& \wedge \quad (y_2 \leftrightarrow (y_3 \vee y_4)) \\
& \wedge \quad (y_3 \leftrightarrow (x_1 \rightarrow x_2)) \\
& \wedge \quad (y_4 \leftrightarrow \neg y_5) \\
& \wedge \quad (y_5 \leftrightarrow (y_6 \vee x_4)) \\
& \wedge \quad (y_6 \leftrightarrow (\neg x_1 \leftrightarrow x_3))
\end{aligned}
$$

3-SAT is NP-Complete (cont.)

- We need to convert our ϕ' to 3-SAT form.
- We first create a truth table for each clause in ϕ'.
- We use the truth table to build a formula in *disjunctive normal form* that is equivalent to $\neg\phi_i'$ (i.e., we just OR all of the 0 values from the truth table).
- Then we can just easily apply DeMorgan's laws to convert $\neg\phi_i'$ to ϕ_i''.
- The resulting clauses each have at most 2 literals. If they only have two, create a new dummy variable, p, and form two clauses, one with p, and one with $\neg p$.
- This final formula is satisfiable if and only if the original ϕ is satisfiable.
- Oh, and we can compute this reduction in polynomial time, which is the last part of the proof.

3-SAT Example

Example

Consider $\phi_1' = (y_1 \leftrightarrow (y_2 \wedge \neg x_2)$. Construct the truth table (the value of the clause for all eight combinations of y_1, y_2, and x_2.

Then the DNF for $\neg\phi_1'$ is

$$(y_1 \wedge y_2 \wedge x_2) \vee (y_1 \wedge \neg y_2 \wedge x_2) \vee (y_1 \wedge \neg y_2 \wedge \neg x_2) \vee (\neg y_1 \wedge y_2 \wedge \neg x_2)$$

We apply DeMorgan's Laws, which complements all of the literals and converts all OR's into AND's and all AND's into OR's. Then ϕ_1'' is:

$$(\neg y_1 \vee \neg y_2 \vee \neg x_2) \wedge (\neg y_1 \vee y_2 \vee \neg x_2) \wedge (\neg y_1 \vee y_2 \vee x_2) \wedge (y_1 \vee \neg y_2 \vee x_2)$$

Which is in 3-SAT form.

Questions