CS 461, Lecture 22

Jared Saia
University of New Mexico

Today's Outline

- Intro to P,NP, and NP-Hardness

Efficient Algorithms

- Q: What is a minimum requirement for an algorithm to be efficient?
- A: A long time ago, theoretical computer scientists decided that a minimum requirement of any efficient algorithm is that it runs in polynomial time: $O(n^c)$ for some constant c
- People soon recognized that not all problems can be solved in polynomial time but they had a hard time figuring out exactly which ones could and which ones couldn't

NP-Hard Problems

- Q: How to determine those problems which can be solved in polynomial time and those which can not
- Again a long time ago, Steve Cook and Dick Karp and others defined the class of *NP-hard* problems
- Most people believe that NP-Hard problems *cannot* be solved in polynomial time, even though so far nobody has *proven* a super-polynomial lower bound.
- What we do know is that if *any* NP-Hard problem can be solved in polynomial time, they *all* can be solved in polynomial time.

Circuit Satisfiability

- **Circuit satisfiability** is a good example of a problem that we don't know how to solve in polynomial time
- In this problem, the input is a *boolean circuit*: a collection of and, or, and not gates connected by wires
- We'll assume there are no loops in the circuit (so no delay lines or flip-flops)

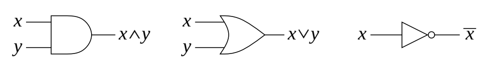
Circuit Satisfiability

- The input to the circuit is a set of m boolean (true/false) values $x_1, \ldots x_m$
- The output of the circuit is a single boolean value
- Given specific input values, we can calculate the output in polynomial time using depth-first search and evaluating the output of each gate in constant time

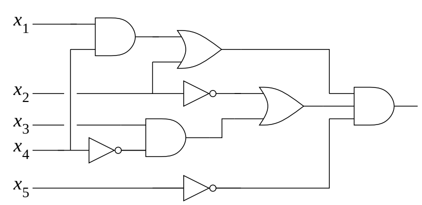
Circuit Satisfiability

- The circuit satisfiability problem asks, given a circuit, whether there is an input that makes the circuit output **True**
- In other words, does the circuit always output false for any collenction of inputs
- Nobody knows how to solve this problem faster than just trying all 2^m possible inputs to the circuit but this requires exponential time
- On the other hand nobody has every proven that this is the best we can do!

Example



An and gate, an or gate, and a not gate.



A boolean circuit. Inputs enter from the left, and the output leaves to the right.

Classes of Problems

We can characterize many problems into three classes:

- **P** is the set of yes/no problems that can be solved in polynomial time. Intuitively P is the set of problems that can be solved "quickly"
- **NP** is the set of yes/no problems with the following property: If the answer is yes, then there is a *proof* of this fact that can be checked in polynomial time
- **co-NP** is the set of yes/no problems with the following property: If the answer is no, then there is a *proof* of this fact that can be checked in polynomial time

NP

- **NP** is the set of yes/no problems with the following property: If the answer is yes, then there is a *proof* of this fact that can be checked in polynomial time
- Intuitively NP is the set of problems where we can verify a **Yes** answer quickly if we have a solution in front of us
- For example, circuit satisfiability is in NP since if the answer is yes, then any set of m input values that produces the **True** output is a proof of this fact (and we can check this proof in polynomial time)

P,NP, and co-NP

- If a problem is in P, then it is also in NP — to verify that the answer is yes in polynomial time, we can just throw away the proof and recompute the answer from scratch
- Similarly, any problem in P is also in co-NP
- In this sense, problems in P can only be easier than problems in NP and co-NP

Examples

- The problem: "For a certain circuit and a set of inputs, is the output **True**?" is in P (and in NP and co-NP)
- The problem: "Does a certain circuit have an input that makes the output **True**?" is in NP
- The problem: "Does a certain circuit have an input that makes the output **False**?" is in co-NP

P Examples

Most problems we've seen in this class so far are in P including:

- "Does there exist a path of distance $\leq d$ from u to v in the graph G?"
- "Does there exist a minimum spanning tree for a graph G that has cost $\leq c$?"
- "Does there exist an alignment of strings s_1 and s_2 which has cost $\leq c$?"

NP Examples

There are also several problems that are in NP (but probably not in P) including:

- **Circuit Satisfiability**
- **Coloring**: "Can we color the vertices of a graph G with c colors such that every edge has two different colors at its endpoints (G and c are inputs to the problem)
- **Clique**: "Is there a clique of size k in a graph G?" (G and k are inputs to the problem)
- **Hamiltonian Path**: "Does there exist a path for a graph G that visits every vertex exactly once?"

The $1 Million Question

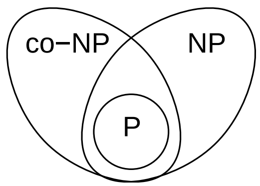
- The most important question in computer science (and one of the most important in mathematics) is: "Does P=NP?"
- Nobody knows.
- Intuitively, it *seems* obvious that P≠NP; in this class you've seen that some problems can be very difficult to solve, even though the solutions are obvious once you see them
- But nobody has proven that P≠NP

NP and co-NP

- Notice that the definition of NP (and co-NP) is not symmetric.
- Just because we can verify every yes answer quickly doesn't mean that we can check no answers quickly
- For example, as far as we know, there is no short proof that a boolean circuit is *not* satisfiable
- In other words, we know that Circuit Satisfiability is in NP but we don't know if its in co-NP

Conjectures

- We conjecture that P≠NP and that NP≠co-NP
- Here's a picture of what we *think* the world looks like:



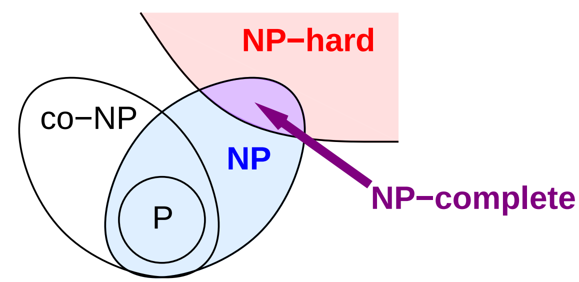
NP-Hard

- A problem Π is **NP-hard** if a polynomial-time algorithm for Π would imply a polynomial-time algorithm for *every problem in NP*
- In other words: **Π is NP-hard iff If Π can be solved in polynomial time, then P=NP**
- In other words: if we can solve one particular NP-hard problem quickly, then we can quickly solve *any* problem whose solution is quick to check (using the solution to that one special problem as a subroutine)
- If you tell your boss that a problem is NP-hard, it's like saying: "Not only can't I find an efficient solution to this problem but neither can all these other very famous people." (you could then seek to find an approximation algorithm for your problem)

NP-Complete

- A problem is *NP-Easy* if it is in NP
- A problem is *NP-Complete* if it is NP-Hard and NP-Easy
- In other words, a problem is NP-Complete if it is in NP but is at least as hard as all other problems in NP.
- If anyone finds a polynomial-time algorithm for even one NP-complete problem, then that would imply a polynomial-time algorithm for *every* NP-Complete problem
- *Thousands* of problems have been shown to be NP-Complete, so a polynomial-time algorithm for one (i.e. all) of them is incredibly unlikely

Example



A more detailed picture of what we *think* the world looks like.

Proving NP-Hardness

- In 1971, Steve Cook proved the following theorem: **Circuit Satisfiability is NP-Hard**
- Thus, one way to show that a problem A is NP-Hard is to show that if you can solve it in polynomial time, then you can solve the Circuit Satisfiability problem in polynomial time.
- This is called a *reduction*. We say that we *reduce* Circuit Satisfiability to problem A
- This implies that problem A is "as difficult as" Circuit Satisfiability.

SAT

- Consider the *formula satisfiability* problem (aka *SAT*)
- The input to SAT is a boolean formula like

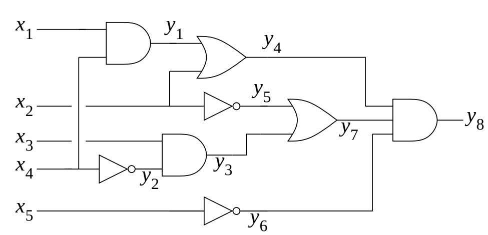
$$(a \vee b \vee c \vee \bar{d}) \Leftrightarrow ((b \wedge \bar{c}) \vee \overline{(\bar{a} \Rightarrow d)} \vee (c \neq a \wedge b)),$$

- The question is whether it is possible to assign boolean values to the variables $a, b, c, \ldots$ so that the formula evaluates to TRUE
- To show that SAT is NP-Hard, we need to show that we can use a solution to SAT to solve Circuit Satisfiability

The Reduction

- *Given a boolean circuit, we can transform it into a boolean formula by creating new output variables for each gate and then just writing down the list of gates separated by AND*
- This simple algorithm is the reduction
- For example, we can transform the example ciruit into a formula as follows:

Example



$$(y_1 = x_1 \wedge x_2) \wedge (y_2 = \overline{x_4}) \wedge (y_3 = x_3 \wedge y_2) \wedge (y_4 = y_1 \vee x_2) \wedge (y_5 = \overline{x_2}) \wedge (y_6 = \overline{x_5}) \wedge (y_7 = y_3 \vee y_5) \wedge (y_8 = y_4 \wedge y_7 \wedge y_6) \wedge y_8$$

A boolean circuit with gate variables added, and an equivalent boolean formula.

Reduction Picture

$$
\begin{tikzcd}
\boxed{\text{boolean circuit}} \arrow[r, "O(n)"] & \boxed{\text{boolean formula}} \arrow[d, Rightarrow, "\text{SAT}"] \\
\boxed{\text{True or False}} & \boxed{\text{True or False}} \arrow[l, "\text{trivial}"']
\end{tikzcd}
$$

Reduction

- The original circuit is satisifiable iff the resulting formula is satisfiable
- We can transform any boolean circuit into a formula in linear time using DFS and the size of the resulting formula is only a constant factor larger than the size of the circuit
- Thus we've shown that if we had a polynomial-time algorithm for SAT, then we'd have a polynomial-time algorithm for Circuit Satisfiability (and this would imply that P=NP)
- This means that SAT is NP-Hard

Showing NP-Completeness

- We've shown that SAT is NP-Hard, to show that it is NP-Complete, we now must also show that it is in NP
- In other words, we must show that if the given formula is satisfiable, then there is a proof of this fact that can be checked in polynomial time
- To prove that a boolean formula is satisfiable, we only have to specify an assignment to the variables that makes the formula true (this is the "proof" that the formula is true)
- Given this assignment, we can check it in linear time just by reading the formula from left to right, evaluating as we go
- So we've shown that SAT is NP-Hard and that SAT is in NP, thus SAT is NP-Complete

Take Away

- In general to show a problem is NP-Complete, we first show that it is in NP and then show that it is NP-Hard
- To show that a problem is in NP, we just show that when the problem has a "yes" answer, there is a proof of this fact that can be checked in polynomial time (this is usually easy)
- To show that a problem is NP-Hard, we show that if we could solve it in polynomial time, then we could solve some other NP-Hard problem in polynomial time (this is called a reduction)

3-SAT

- A boolean formula is in *conjunctive normal form* (CNF) if it is a conjunction (and) of several *clauses*, each of which is the disjunction (or) or several *literals*, each of which is either a variable or its negation. For example:

$$\overbrace{(a \vee b \vee c \vee d)}^{\text{clause}} \wedge (b \vee \bar{c} \vee \bar{d}) \wedge (\bar{a} \vee c \vee d) \wedge (a \vee \bar{b})$$

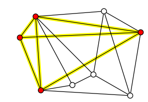
- A *3CNF* formula is a CNF formula with exactly three literals per clause
- The 3-SAT problem is just: "Is there any assignment of variables to a 3CNF formula that makes the formula evaluate to true?"

3-SAT

- 3-SAT is just a restricted version of SAT
- Surprisingly, 3-SAT also turns out to be NP-Complete (proof omitted for now)
- 3-SAT is very useful in proving NP-Hardness results for other problems, we'll see how it can be used to show that CLIQUE is NP-Hard

CLIQUE

- The last problem we'll consider in this lecture is CLIQUE
- The problem CLIQUE asks "Is there a clique of size k in a graph G?"
- Example graph with clique of size 4:



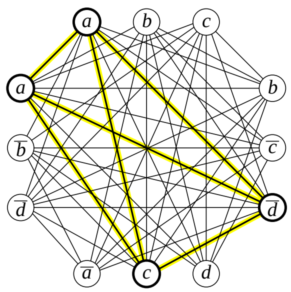
- We'll show that Clique is NP-Hard using a reduction from 3-SAT. (the proof that Clique is in NP is left as an exercise)

The Reduction

- Given a 3-CNF formula F, we construct a graph G as follows.
- The graph has one node for each instance of each literal in the formula
- Two nodes are connected by an edge is: (1) they correspond to literals in different clauses and (2) those literals do not contradict each other

Reduction Example

- Let F be the formula: $(a \vee b \vee c) \wedge (b \vee \bar{c} \vee \bar{d}) \wedge (\bar{a} \vee c \vee d) \wedge (a \vee \bar{b} \vee \bar{d})$
- This formula is transformed into the following graph:



(look for the edges that *aren't* in the graph)

Reduction

- Let F have k clauses. Then G has a clique of size k iff F has a satisfying assignment. The proof:
- **k-clique $\Longrightarrow$ satisfying assignment:** If the graph has a clique of k vertices, then each vertex must come from a different clause. To get the satisfying assignment, we declare that each literal in the clique is true. Since we only connect non-contradictory literals with edges, this declaration assigns a consistent value to several of the variables. There may be variables that have no literal in the clique; we can set these to any value we like.
- **satisfying assignment $\Longrightarrow$ k-clique:** If we have a satisfying assignment, then we can choose one literal in each clause that is true. Those literals form a k-clique in the graph.

Reduction Picture