451/651 Lecture 16 – **NP**-completeness

Outline

- Reductions and expressiveness
- Formal definitions: decision problems, **P** and **NP**.
- Circuit-SAT and 3-SAT
- Examples of showing **NP**-completeness.

Reductions and Expressiveness

In the last few lectures we have seen:

- Bipartite matching can be solved with a max flow algorithm.
- The max flow problem can be solved by a linear programming algorithm.

In this lecture we expand the idea of a reduction from one problem to another. And we expand the application of reductions to prove lower bounds on problem difficulty.

Polynomial Time

Definition: We say that an algorithm runs in **Polynomial Time** if, for some constant c, its running time is $O(n^c)$, where n is the size of the input.

Input size: size of the problem description in bits.

> *Think about why the basic Ford-Fulkerson algorithm is* not *a polynomial-time algorithm for network flow when edge capacities are written in binary, but both of the Edmonds-Karp algorithms* are *polynomial-time.*

Reducibility

Definition: A problem A is **poly-time reducible** to problem B (written as $A \leq_p B$) if we can solve problem A in polynomial time given a polynomial time black-box algorithm for problem B.[1]
Problem A is **poly-time equivalent** to problem B ($A =_p B$) if $A \leq_p B$ and $B \leq_p A$.

Think about the examples mentioned above – bipartite matching, max flow, linear programming.

[1] You can loosely think of $A \leq_p B$ as saying "A is no harder than B, up to polynomial factors."

Decision Problems

We consider *decision problems*, whose answer is YES or NO.

E.g., "Does the given network have a flow of value at least k?"

E.g., "Does the given graph have a 3-coloring?"

For such problems, we split all instances into two categories: YES-instances (whose correct answer is YES) and NO-instances (whose correct answer is NO). We put any ill-formed instances into the NO category.

Karp Reductions

Definition: Karp reduction (aka Many-one reduction) from problem A to problem B: To reduce problem A to problem B we want a function f that maps arbitrary instances of A to instances of B such that:

1. if x is a YES-instance of A then $f(x)$ is a YES-instance of B.
2. if x is a NO-instance of A then $f(x)$ is a NO-instance of B.
3. f can be computed in polynomial time.

Superficially this seems more limited than the $B \leq_p A$ reductions we defined earlier. But it's cleaner and simpler and is not known to be different.

Defintion of **P**

We can now define the complexity classes **P** and **NP**. These are both classes of decision problems. (Sets of sets of strings.)

Definition: P is the set of decision problems solvable in polynomial time.

E.g., the decision version of the network flow problem: "Given a network G and a flow value k, does there exist a flow $\geq k$?" belongs to **P**.

The Concept of **NP**

Informally **NP** is the class of problems for which any YES-instance can be efficiently checked if given a proper hint.

The Traveling Salesperson Problem asks: "Given a weighted graph G and an integer k, does G have a tour that visits all the vertices and has total length at most k?"

If someone gave us such a tour we could easily check if it satisfied the desired conditions. Therefore it's in **NP**.

The 3-Coloring problem asks: "Given a graph G, can vertices be assigned colors red, blue, and green so that no two neighbors have the same color?"

Again, to check a proposed solution is easy. Therefore it's in **NP**.

Formal Definition of **NP**

Definition: **NP** is the set of decision problems that have polynomial-time *verifiers*. Specifically, problem Q is in **NP** if there is a polynomial-time algorithm $V(I, X)$ such that:

- If I is a YES-instance, then there exists X such that $V(I, X) = \text{YES}$.
- If I is a NO-instance, then for all X, $V(I, X) = \text{NO}$.

Furthermore, X should have length polynomial in size of I (since we are really only giving V time polynomial in the size of the instance, not the combined size of the instance and solution).

The question: Does **NP** $=$**P** is THE major unsolved problem in theoretical computer science. We won't talk about it here.

Definition of **NP**-Complete

Loosely speaking, **NP**-complete problems are the "hardest" problems in **NP**, if you can solve them in polynomial time then you can solve any other problem in **NP** in polynomial time. Formally,

Definition: Problem Q is **NP**-complete if:

1. Q is in **NP**, and
2. For any other problem Q' in **NP**, $Q' \leq_p Q$.

So if Q is **NP**-complete and you could solve Q in polynomial time, you could solve *any* problem in **NP** in polynomial time. If Q just satisfies part (2) of this definition, then it's called **NP**-hard.

CIRCUIT-SAT – the First **NP**-complete problem

CIRCUIT-SAT: Input: an acyclic circuit C of NAND gates with a single output. Answer: YES if there is a setting of the inputs that causes C to output 1? NO otherwise.

Theorem: CIRCUIT-SAT is **NP**-complete.

Proof: Wave hands vigorously. (For more on how to wave your hands, read the lecture notes.)

3-SAT – the Second **NP**-complete problem

3-SAT: Given: a CNF formula (AND of ORs) over n variables $x_1, \ldots, x_n$, where each clause has at most 3 variables in it. E.g., $(x_1 \vee x_2 \vee \bar{x}_3) \wedge (\bar{x}_2 \vee x_3) \wedge (x_1 \vee x_3) \wedge \ldots$. Answer YES if there exists an assignment to the variables that satisfies the formula, output NO otherwise.

The *literals* of each conjunction are distinct. (Delete dupliates.)

Theorem: Circuit-SAT $\leq_p$ 3-SAT.

Proof: (Coming up in a moment)
Hence 3-SAT is **NP**-complete. Why?

CIRCUIT-SAT $\leq_p$ 3-SAT implies 3-SAT is **NP**-complete

Proof that $\textsc{Circuit-SAT} \leq_p \text{3-SAT}$

Proving **NP**-completeness in Two Easy Steps

If you want to prove that problem Q is NP-complete, you need to do two things:

1. Show that Q is in NP.
2. Choose some NP-hard problem P to reduce from. This problem could be 3-SAT or CLIQUE or $\cdots$ any of the zillions of NP-hard problems known.

   Now you want to reduce **from** P **to** Q. In other words, given any instance I of P, show how to transform it into an instance $f(I)$ of Q, such that

   $$I \text{ is a YES-instance of } P \iff f(I) \text{ is a YES-instance of } Q.$$

   Note the "$\iff$" in the middle—*you need to show both directions*.
   You also need to show that the mapping $f(\cdot)$ can be done in polynomial time (and hence $f(I)$ has size polynomial in the size of the original instance I).

Why is it useful to prove a problem is **NP**-complete?

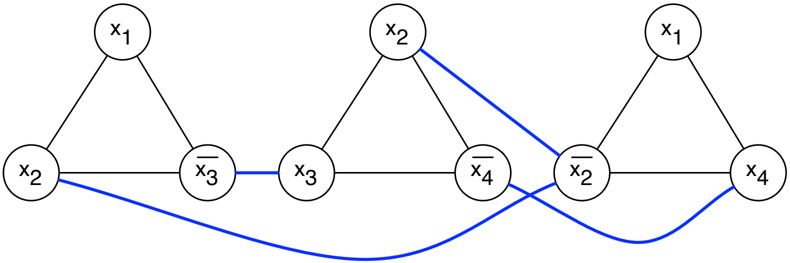
From the point of view of algorithm design, why is this useful?

Independent Set is **NP**-complete

The Independent Set problem is: given a graph G and an integer k, does G have an independent set of size $\geq k$?

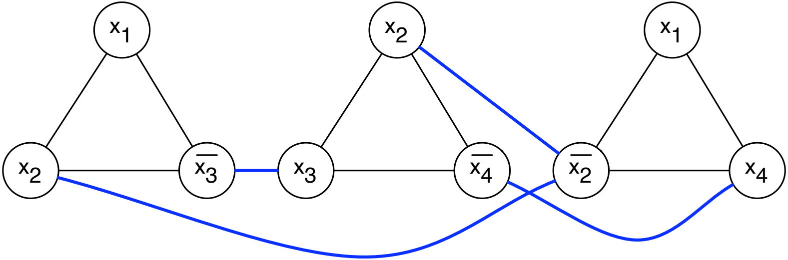
Theorem: Independent Set is **NP**-complete.
Proof: First we observe that Independent Set $\in$ **NP**. (Trivial.)
Then we show that 3-SAT $\leq_p$ Independent Set, as shown in the following example:

$$(x_1 \vee x_2 \vee \overline{x_3}) \wedge (x_2 \vee x_3 \vee \overline{x_4}) \wedge (x_1 \vee \overline{x_2} \vee x_4)$$



Independent Set is **NP**-complete, contd.

$$(x_1 \vee x_2 \vee \overline{x_3}) \wedge (x_2 \vee x_3 \vee \overline{x_4}) \wedge (x_1 \vee \overline{x_2} \vee x_4)$$



We need to show two things:

1. A satisfying assignment gives us an independent set of size k.
2. An independent set of size k gives us a satisfying assignment. (This one is subtle.)

Vertex Cover, Set Cover, Clique

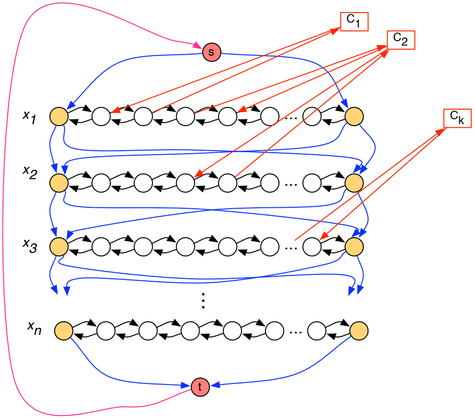
These are easy. See the lecture notes.

Ham Cycle

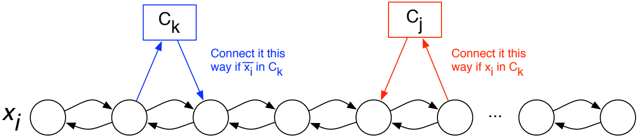
A *Hamiltonian cycle* is a cycle in a graph that visits every vertex exactly once. The Ham Cycle problem asks, given a directed graph G, is there a Hamiltonian cycle.

Theorem: Ham Cycle is **NP**-complete

Proof: Obviously Ham Cycle is in **NP**. We reduce from 3-Sat. Let ϕ be an arbitrary 3SAT instance with clauses $c_1, \ldots, c_m$ and variables $x_1, \ldots, x_n$. Construct the following gadget that represents all possible truth assignments: (See next page)



Add a new node for each clause:



Direction we travel along this chain represents whether to set the variable to **true** or **false**.

← **true**

false →