Math/CCS 103 Professor: Padraic Bartlett

Minilecture 6: NP-Completeness

Week 2 *UCSB 2014*

In Wednesday's class, we studied the two classes of problems P and NP, which we defined as follows:

Definition. A problem is said to be of class P if there is an integer k and algorithm A that solves instances of length n of this problem with runtime $O(n^k)$.

Problems in P, in a sense, are the ones we have reasonable hopes of computing for most relevant values of n.

NP is the following second class of problems:

Definition. A true-false problem R is said to be of class NP if it has "efficiently verifiable proofs." Specifically, we ask for the following two things:

- Given any instance I for which our problem responds "true," there is a proof of this claim.
- There is an algorithm A that, given any proof that claims an instance I of our problem is true, can verify whether this proof actually corresponds to I being true in polynomial time.

Roughly speaking, NP is the set of problems that we can "quickly check solutions for." In a sense, pretty much any reasonable task that you'd ever ask anyone to solve falls in this category. This, heuristically speaking, is because of the following idea: if a task is remotely reasonable or useful, then solutions of it should be relatively easy to tell apart from nonsolutions (or you could just not bother finding a solution to your problem in the first place!)

Finally, we proved[1] the following theorem:

Theorem. Any problem in P is in NP.

From there, we were in position to finally understand the following conjecture, which is one of the Millenium Prize Problems:

Conjecture. The two classes P and NP are different. In other words, P $\neq$ NP.

It is perhaps easy to understand why we'd believe that P and NP are different classes — after all, it certainly makes sense that some problems that we'd care about would have "hard" solutions, but be "easy" in some fashion to check. What may not be quite as obvious is why we'd make the claim about all of NP at once. After all, lots of problems are in NP! We list a few famous ones here:

[1] Or rather, I had hoped that we would prove this theorem. Instead, we started Friday's lecture by proving this theorem.

- **SAT**: A **boolean variable** is simply some variable that can be assigned to either true or false. Given two boolean variables x, y, we can create various boolean expressions by combining them in various ways:
  - $x \wedge y$, pronounced "x and y," is the boolean expression that evaluates to **true** whenever both x **and** y are true, and **false** otherwise.
  - $x \vee y$, pronounced "x or y," is the boolean expression that evaluates to **true** whenever one of either x **or** y are true, and **false** otherwise. It bears noting that this is an "inclusive" or: i.e. if both x and y are true, then certainly *one* of x and y are true, and thus we say that $x \vee y$ is true.
  - $\neg x$, pronounced "not x", is true if x is false, and false if x is true.

  A **boolean formula** is just some long chain of boolean variables linked together by and's, or's and not's. For example, the formula below is a Boolean formula:

  $$(x \wedge y) \vee (\neg(z \wedge (x \vee y))) \vee (a \wedge a).$$

  A formula is called **satisfiable** if there is some way of assigning true or false to each of its variables, in such a way that the formula evaluates to true. For example, the formula above is satisfiable: set $x, y, z, a =$ true, which yields

  $$\begin{aligned}(T \wedge T) \vee (\neg(T \wedge (T \vee T))) \vee (T \wedge T) &= T \vee (\neg(T \wedge T)) \vee T \\ &= T \vee (\neg T) \vee T \\ &= T \vee F \vee T \\ &= T.\end{aligned}$$

  SAT is the following problem: given some boolean formula, is it satisfiable?

- **3SAT**: A boolean formula is said to be in **conjunctive normal form** if we can write it as the conjunction (and) of several clauses, where a clause is some disjunction (or) of several literals, and a literal is either a variable or its negation. For example, the following formula is in conjunctive normal form:

  $$(x \vee y \vee z) \wedge (\neg x \vee x) \wedge (z) \wedge (y \vee y \vee x).$$

  A boolean formula is said to be in **3-conjunctive normal form** if we can write it in conjunctive normal form, where each disjunction (or) contains precisely three literals. For example, the following formula is in 3-conjunctive normal form:

  $$(x \vee y \vee z) \wedge (\neg x \vee x \vee x) \wedge (a \vee a \vee a).$$

  3SAT is the following problem: given any boolean formula written in 3-conjunctive normal form, is it satisfiable?

- k-**Independent Set**: A graph $G = (V, E)$ has an **independent set of size** k if there is some subset of k vertices in G, no two of which are connected by an edge.

  Our problem, then, is the following question: given a graph $G = (V, E)$, does it have an independent set of size k?

- k-**Clique**: A graph $G = (V, E)$ has a **clique of size** k if there is some subset of k vertices in G, every two of which are connected by an edge.

  Our problem, then, is the following question: given a graph $G = (V, E)$, does it have a clique of size k?

This is a large collection of problems, many of which look quite different! In this sense, it may not seem obvious that these are all problems we should be grouping together — maybe some of them are easier to solve than others?

1 NP-Hard: Definitions

Hahahaha, . . . no. Consider the following definition:

Definition. A problem P is called **NP-hard** if "solving it allows you to solve every other problem in NP," in the following sense:

- Take any problem R in NP.
- Any algorithm A_P that solves P can be turned into an algorithm that solves R, with an increase in runtime that is at most polynomial.

In other words, if you can solve an NP-hard problem in polynomial time, then you can solve **every NP problem** in polynomial time.

Theorem. (Cook-Levin) There is an NP-hard problem. In particular, there is an NP-hard problem contained within NP! Specifically, 3SAT is NP-hard.

(We call any problem that is both NP-hard and in NP a **NP-complete** problem.)

This might be, um, unexpected. After all, all of these problems look remarkably different! How can solving 3SAT allow you to solve **all** of them?

The proof of this statement is somewhat beyond the scope of this class, though the wikipedia article on the Cook-Levin theorem is certainly worth reading if you want an idea of how the argument goes.

Instead, what I want to spend the rest of class talking about is how to discover **more** things like this! Specifically, consider the following blueprint for showing that a problem is NP-hard:

- Take any problem P that we suspect is NP-hard.
- Take as well some other problem R that we know is NP-hard, like 3SAT.
- Suppose that we can "reduce" R to P, in the following sense: given any algorithm that solves P, suppose that we can "transform" this algorithm into a new algorithm that can solve R, with an increase in time that is at most polynomial. (The idea here is that if R can be reduced to P, then knowing how to solve P lets you solve R: i.e. the problem R is relatively easy if we understand P, and has therefore been "reduced" to essentially the same problem as P.)

- Then P is NP-hard! This is because given any algorithm that solves P, we can turn it into an algorithm that solves R, which (because R is NP-hard) we can turn into an algorithm that solves **any** NP problem, with at most polynomial increases in runtime.

2 NP-Hard: Results

We show a few problems are NP-hard here:

Theorem. Given a graph G, determining for which values of k this graph has an **independent set** of size k is NP-hard.

Proof. We reduce 3SAT to the independent set problem. Doing this will give us that the independent set problem is NP-hard, which is our claim.

To do this, we perform the following process:

- Take any formula F in 3SAT. We will describe a way to construct in polynomial time a graph G such that the following holds:

$$F \text{ is satisfiable} \Leftrightarrow G \text{ has an independent set of size } k.$$

- If we can do this, then given any algorithm that solves the k-independent set problem, we can solve any instance F of 3SAT by simply encoding F as a graph in polynomial time and then running our algorithm to decide if F is satisfiable.

- This is a reduction of 3SAT to the independent set problem, which shows that the independent set problem is NP-complete!

So: take any boolean formula F in 3-conjunctive normal form, with k clauses. Create a graph G as follows:

- Vertices: for each literal in a clause of F, create a vertex for that literal.

- Edges: Connect two vertices whenever one of the two following conditions hold:

  1. The two literals that these two vertices correspond to are from the same clause.
  2. The two literals that these two vertices correspond to are a variable and its negation (i.e. one is x and the other is $\neg x$, for some x).

We now observe that our formula with k clauses is satisfiable if and only if our graph has an independent set of size k. This is not too hard to see:

1. First, assume that our graph has an independent set of k variables. Then in our independent set, we must have one vertex from each clause, because all of the vertices corresponding to any given clause are connected with edges. Furthermore, we never have a vertex x and its negation $\neg x$ associated to variables in our independent set, because we connected all such pairs with edges.

   Look at the truth assignment that makes every literal equal to true in the independent set (i.e. picks out truth values for the variables so that each literal $x, \neg y, \neg z \ldots$ in our independent set is true.) This is consistent, in that because we have no edges between

x's and $\neg x$'s, we don't try to assign both true or false to any given variable. This might leave some variables with unassigned truth values; set those to true as well.

This truth assignment then satisfies F! To see why, simply notice that in every clause, we have insured that one literal at least is true. This means that each clause, as it is an "or" of three things, one of which is true, evaluates to true. Therefore, the entire formula is k true's "and-ed" together, which is clearly true.

2. Now, we go the other direction: we will show that if our formula is satisfiable, then our graph has an independent set of size k. This is easy: take any set of variables that satisfies our formula. Because our formula is satisfied, there is one literal from each formula that is true: take one such vertex for each clause. We then know that none of these vertices are linked by edges, as they come from different clauses and they all correspond to true (so we never picked connected pairs of the form $x, \neg x$.) Therefore this is an independent set with one element for each clause: i.e. an independent set of size k!

This completes our reduction proof. □

Theorem. Given a graph G, determining for which values of k this graph has a **clique** of size $k$is NP-hard.

Proof. We reduce the independent set problem to the clique problem. Doing this gives us that the clique problem is NP-hard, as claimed.

This is easier than before! To reduce the independent set problem to the clique problem, simply make the following observation:

A graph has an independent set of size $k \Leftrightarrow$ its complement graph[2] has a clique of size k.

Therefore, if we can quickly decide if a graph has a clique of size k, we can quickly decide if it has an independent set of size k, by simply constructing the complement graph (which we can do in polynomial time.)

Therefore, this process reduces the independent set problem to the clique problem! □

[2]The complement graph $\overline{G}$ to a graph G is the graph with the same set of vertices as G, where we connect two vertices in $\overline{G}$ precisely when those two vertices are not connected in G.