Lecture 16: NP-Hard and NP-Complete Problems

Ryan Bernstein

1 Introductory Remarks

- HW 4 is assigned, and is due next Thursday
- As discussed during the last lecture, I will drop your lowest homework score
- It's likely that we'll finish the remainder of the course material today, which would mean that we have one lecture unaccounted for (next Thursday I'll be bringing in a practice final). Think about what you want to do with that time, assuming this happens

1.1 Assignment 3 Solutions

1.2 Recapitulation

1.2.1 Complexity Classes

On Tuesday, we introduced the idea of time complexity classes. A time complexity class is defined as follows:

> $TIME(f(n)) = \{L \mid L$ is a language decidable by a $O(f(n))$-time single-tape deterministic Turing machine$\}$

We also have time complexity classes for nondeterministic machines, which are defined very similarly:

> $NTIME(f(n)) = \{L \mid L$ is a language decidable by a $O(f(n))$-time single-tape nondeterministic Turing machine$\}$

There are an infinite number of such complexity classes, since there are an infinite number of possible time functions $f(n)$. Therefore, when talking about computability theory, we often restrict ourselves to a couple of larger, (arguably) more important classes. Three such classes are P, NP, and EXP.

$$P = \bigcup_{k \geq 0} (TIME(n^k))$$

This means that P is the set of all languages that are decidable in polynomial time. To show that some problem is in P, we can simply create a single-tape deterministic Turing machine that solves the problem in some polynomial number of steps.

NP stands for "nondeterministic polynomial", and has a similar definition:

$$NP = \bigcup_{k \geq 0} (NTIME(n^k))$$

This means that NP is the set of all languages that are decidable in polynomial time by a nondeterministic machine. Since nondeterminism is not a realistic model of computation, though, we have an alternate model that is defined in terms of standard deterministic Turing machines. To place a problem in NP, we can either:

1. Create a nondeterministic single-tape Turing machine that solves the problem in polynomial time, or
2. Create a deterministic single-tape Turing machine that can verify some candidate solution c in polynomial time.

What does a certificate look like? Many of these decision problems that we deal with are things like $\{\langle G \rangle G$ is a graph that contains a Hamiltonian path$\}$. In this case, we could provide a certificate that was a path in G. If we can verify that the given path is Hamiltonian in a polynomial number of steps, we can consider this verification as proof that such a path exists.

The third complexity class EXP is defined in much the same way as P:

$$EXP = \bigcup_{k \geq 0} (TIME(2^{n^k}))$$

Problems in EXP can be decided by a single-tape deterministic Turing machine in exponential time.

Clearly, P is not larger than NP. If we can solve a problem in polynomial time with a deterministic machine, we should also be able to solve that same problem in polynomial time with a *non*deterministic machine.

Probably less obvious is the fact that NP is not larger than EXP. We can think of nondeterministic computations as a giant tree, with a branch in each place where we have to make a nondeterministic choice. The runtime of a nondeterministic machine is the length of the longest branch.

If a nondeterministic machine can traverse a branch of this tree in $O(n^k)$ steps, a deterministic machine could simply explore all of these branches. Since there are an exponential number of branches, this would take exponential time, making the runtime of such a deterministic machine $O(2^{n^k})$.

We know two things about these classes:

1. $P \subseteq NP \subseteq EXP$
2. $P \subset EXP$, for reasons that are proven in Chapter 9, but that we won't cover here.

Beyond that, the precise relationship between these three classes is still an open problem in computer science.

1.2.2 Poly-Time Reductions

At the end of the last lecture, we also introduced the poly-time reduction, which we can use to examine the relative difficulty of two problems. We say that a problem A is poly-time reducible to a problem B — written $A \leq_p B$ — if there exists a computable function f such that:

1. $\forall s(s \in A \rightarrow f(s) \in B)$
2. $\forall s(s \notin A \rightarrow f(s) \notin B)$
3. f can be computed in polynomial time.

If $A \leq_p B$, we say that A is *not harder than* B. This means that:

1. If $A \leq_p B$ and $B \in P$, then $A \in P$
2. If $A \leq_p B$ and $B \in NP$, then $A \in NP$

Since we don't know whether or not $P = NP$, we can't really go the other direction.

Poly-time reductions are incredibly difficult to create. Some of them have taken years of work by people at the forefront of computer science. We'll examine some of these reductions today, but we won't actually be creating them ourselves.

2 NP-Hard Problems

We can use this idea of poly-time reducibility to introduce a new class of problems, called NP-Hard. We say that a problem A is NP-Hard if every problem in NP is poly-time reducible to A.

If creating a poly-time reduction from some A to some B is already a bit of a tall order, then creating a reduction from *every problem in NP* is taller still. Fortunately, we don't need to do this.

The Cook-Levin theorem shows that all problems in NP are poly-time reducible to a single problem, called SAT (short for "Boolean Satisfiability"). This is defined as follows:

$$\text{SAT} = \{\langle \Phi \rangle \mid \Phi \text{ is a Boolean sentence with a satisfying assignment}\}$$

This means that Φ is a propositional sentence like the ones that we saw in CS 251 (e.g. $(x_1 \wedge x_2) \vee \neg x_3$). We say that Φ has a satisfying assignment if there exists some series of T/F values that, if assigned to the variables in Φ, causes Φ to evaluate to T.

From here, we can harness the fact that reducibility is transitive:

$$(\forall A \in NP(A \leq_p SAT)) \wedge SAT \leq_p B \rightarrow (\forall A \in NP(A \leq_p B))$$

Since all problems in NP are reducible to SAT, showing that SAT is reducible to some new problem B is sufficient to show that all problems in NP are reducible to B.

To simplify our reductions, we often use a related problem called 3SAT. 3SAT is similar to SAT, but places the following restrictions on the form of Φ:

1. Φ is in Conjunctive Normal Form
2. Every disjunctive clause in Φ contains three variables

An example of this would be something like $(x_1 \vee x_2 \vee \overline{x_1}) \wedge (x_2 \vee \overline{x_3} \vee \overline{x_1})$.

Example In graph theory, a *clique* is a subgraph that is fully connected (in other words, a subset of V in which every node is connected to every other node). Let $KClique = \{\langle G, k\rangle \mid G \text{ is an undirected graph with a clique of size } k\}$. Show that $KClique$ is NP-Hard.

To do this, we'll show that $3SAT \leq_p KClique$. This means that we need a poly-time function that transforms a $3SAT$ problem into a graph which contains a clique of size k if and only if that $3SAT$ problem has a satisfying assignment.

As mentioned before, poly-time reductions are incredibly difficult and time consuming. Therefore:

1. This is the only one we're going to see, and
2. We're going to go over it at a fairly high level of abstraction.

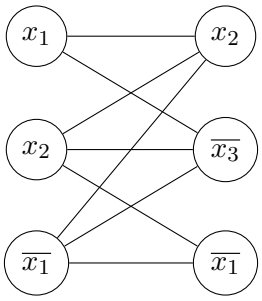
If you're stressing out about this, remember that you're not going to be expected to create one of these on the exam. Anyway, here goes.

$F =$ "On input $\langle Phi\rangle$:

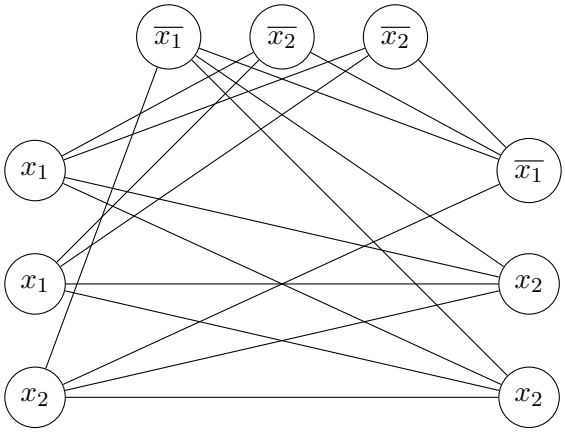
1. Construct a graph G as follows:
   (a) Add a vertex for each variable that appears in Φ
   (b) Add an edge between every v_i and v_j such that 1) v_i and v_j did not originate in the same clause and 2) the variables represented by v_i and v_j are *not* mutually exclusive (i.e. pairs that are not of the form x_k and $\overline{x_k}$)
2. Output $\langle G\rangle$"

If the $3SAT$ problem has a satisfying assignment, then there will be a clique with a size equal to the number of conjunctive clauses.

How does this work? Consider the example formula we saw above, $(x_1 \vee x_2 \vee \overline{x_1}) \wedge (x_2 \vee \overline{x_3} \vee \overline{x_1})$. Let's build this graph, arranging vertices from the same clause close to each other:



This is a pretty simple example, since $k = 2$: every pair of connected vertices is a 2-clique. As a slightly more interesting example, consider the 3-clause sentence $\Phi = (x_1 \vee x_1 \vee x_2) \wedge (\overline{x_1} \vee \overline{x_2} \vee \overline{x_2}) \wedge (\overline{x_1} \vee x_2 \vee x_2)$. Since we have three clauses, we now must search for a 3-clique.



3 NP-Completeness

We've now very painstakingly shown that *KClique* is NP-Hard. We can actually go a bit further, though. Consider the following determnistic verifier for *KClique*. c is a set of k distinct vertices.

$V =$ "On input $\langle G, k, c \rangle$:

1. For each $v_i \in c$:
   (a) For each $v_j \in c - v_i$:
      i. If E does not contain an edge connecting v_i and v_j, REJECT
2. ACCEPT"

What is the runtime of V? Both loops should execute $O(k)$ times. The innermost statement requires an iteration over E. Therefore, this machine runs in $O(k^2 \cdot |E|)$ time. This is polynomial.

Since we can build a poly-time deterministic verifier for *KClique*, we've shown that $K \in NP$. We actually have a name for this intersection. We say that a problem B is *NP-Complete* if:

1. $\forall A \in NP(A \leq_p B)$
2. $B \in NP$

Since we conceive of poly-time reductions as illustrating a "not harder than" relationship, we can say that NP-Complete problems are the *hardest problems in NP*.

The other examples of problems in NP that we've seen are actually also NP complete. Similar mapping reductions exist that can map SAT or 3SAT to these problems in poly-time, which makes them NP-hard, and we've already shown that they're in NP. As a reminder, the other NP problems we've seen are:

1. HAMPATH
2. KNAPSACK
3. SUBSETSUM

4 Implications of NP-Hardness and NP-Completeness

Why are the classes of NP-Hard and NP-Complete problems important? As it turns out, they contain some important problems with the potential to significantly advance technology. Efficient solutions to these problems would greatly help with things like route planning, network mapping, protein folding, and a whole bunch of other stuff that I'm not smart enough to understand.

The other reasons is their relationship to each other. Because all problems in NP are poly-time reducible to all NP-Hard problems, an efficient poly-time solution to *any* NP-Hard problem will allow us to solve *all* problems in NP in poly-time. This would prove that $P = NP$, and nets you a Turing award and a million dollar prize.

5 Optimization Problems

What problems are *not* in NP? A common example is that of optimization problems. Perhaps unsurprisingly, optimization problems require us to find the absolute best solution to some problem.

On Tuesday, we looked at the Knapsack Problem, which we defined as follows: $\{\langle V, W, m, n\rangle \mid V$ is a sequence of item values, W is a parallel sequence of item weights, and there exists a set of items with a value of at least m with a weight less than $n\}$. More informally, it's the question "If I can only carry a certain amount of weight, is there a collection of items that I can fit in my knapsack that has a value of at least m?"

How do we decide when we're satisfied, though? What if we simply want the maximal value we can achieve? This leads to an an alternative take on the Knapsack problem:

$\textsc{Knapsack}_O = \{\langle V, W, n, S\rangle \mid V$ is a sequence of item values, W is a parallel sequence of item weights, and S is a set of items that has the maximal value that I can create with a weight limit of $n\}$.

Rather than deciding that we're satisfied with some value m, we've decided that we want the best possible series of items that we can fit into our knapsack. Since we're trying to optimize the value of our carrying capacity, this is our first example of an optimization problem.

What's the major difference between this and the Knapsack that we originally introduced on Tuesday? Recall our second definition of NP:

> NP $= \{L \mid L$ is a problem that can be verified in polynomial time using a deterministic Turing machine$\}$

When we're given some target value, it's pretty easy to just examine a given set of items (the certificate c) and ensure that it matches the target value without exceeding the weight limit. Showing that a sequence of items has the *optimal* value is a much trickier issue, though. We don't really have a way to do it besides comparing our set of items to all other sets of items and ensuring that ours has the highest value.

Another famous optimization problem is called the Traveling Salesman Problem, defined as follows:

> Given a weighted, undirected graph, what is the shortest path that will visit all nodes exactly once and return to the origin?

We can see that this is similar to finding a Hamiltonian circuit, but with weighted edges. We see problems like this — where we have some "travel cost" which we wish to minimize — all the time, including in the fields of chip design and DNA sequencing. Like the Knapsack problem, this optimization problem is NP-Hard, but has not been shown to be in NP. However, it too can be turned into an NP-Complete decision problem by assigning some cost limit that we want to stay below.

6 Things to Take Away

This whole time complexity section has been a bit abstract. There have been a lot of procedures that we simply haven't touched, because this is where computability gets complicated in ways that put it beyond the scope of an undergraduate final exam. What, then, should we have taken away from these last two lectures? We can break concepts into two categories: 1) things we should be able to do and 2) things we should be able to discuss. The first category involves things like proofs that you may have to write, whereas the second encompasses concepts that may show up in true/false questions or short, informal arguments.

Things We Should Be Able to Do	Things We Should Be Able to Discuss
• Analyze the running time of a Turing machine • Show that a problem is an element of P by creating a poly-time deterministic solver for it • Show that a problem is an element of NP by either: 1. Creating a poly-time nondeterministic solver for it, or 2. creating a poly-time deterministic verifier for it	• The meanings of the complexity classes P, NP, NP-Hard, and NP-Complete • The relationship between those same complexity classes • The significance of a poly-time reduction between some problems A and B