CS 6505, Complexity and Algorithms Week 7: NP Completeness

Reductions

We have seen some problems in P and NP, and we've talked about space complexity. The Space Hierarchy Theorem showed us that there are some problems that can be solved in certain amounts of space but cannot be solved with any less space. In $PSPACE$, we constructed a series of increasingly more difficult problems using Alternating Turing Machines and also introduced the concept of PSPACE-Completeness. Now we want to know about NP – are there some problems that are more difficult than others? Do we have a concept of "completeness" in NP that lets us decide if a class of problems are just as hard to solve as others?

We've seen some NP problems and coNP problems before. Remember, an NP problem is one that has a short proof that a solution is correct, and a coNP problem is one that has a short proof that there is no solution. Here are some examples:

Let $L = \{$ (G,k) : G has an independent set of size $\geq$ k $\}$
Then $\overline{L} = \{$ (G,k) : G has an independent set of size $<$ k $\}$

So these two problems are *complements* of each other. (Note that L is the problem INDEPENDENT-SET.)

We have also briefly mentioned SATISFIABILITY, but let's revisit it. Suppose we have a formula as follows:

$$\phi = (x_1 \vee x_2 \vee \overline{x_3} \vee x_4) \wedge (\overline{x_2} \vee x_3) \wedge (x_3 \vee \overline{x_4} \vee x_5)$$

In the formula ϕ, each variable x_j is a Boolean ***literal*** that may be 0 or 1, and a collection of the literals joined by disjunctions (e.g., $(x_{2,1} \vee x_{2,2})$) is called a ***clause***. All of the clauses are linked together with conjunctions ($\wedge$). We can find an assignment that makes this formula true. For the formula above, we could choose $x_1 = 1,\ x_2 = 0,\ x_3 = 1,\ x_4 = 0,\ x_5 = 1$. There may be many ways to satisfy a formula.

Now let's define a generic problem:

$$SAT = \{\ \phi : \phi\ has\ a\ satisfying\ assignment\ \}$$

$$\overline{SAT} = \{\ \phi : \phi\ does\ not\ have\ a\ satisfying\ assignment\ \}$$

We use SAT in place of "SATISFIABILITY". If each clause has exactly 3 literals within it, then we refer to this problem as ***3-SAT***.

SAT is in NP. Suppose that we are given a certificate with ϕ and a satisfying assignment $x_1 = b_1, x_2 = b_2, \ldots, \ x_n = b_n$. ϕ has at most n clauses, and each clause requires a constant amount of time to evaluate. If we try to solve SAT through brute force, we may end up with an exponential number of possibilities to search through. However, this is not enough to say that a problem is "Complete" for NP: we need something more.

Let's see some more problems that we know are in NP. First, we have the Traveling Salesman Problem (TSP):

$TSP = \{(G, d_{ij}) : \exists$ tour through all vertices of total length $\leq D?\}$

In other words, we are given a graph G with weights d_{ij} for the edge (if any) between vertices v_i and v_j for all possible vertices. Now we want to know if there is a tour, or path through G that visits every vertex and returns to the beginning, where the sum of the edges' weights is less than some value D. Informally, we can see that TSP is also in NP: if we are given an ordering of the vertices to visit, then we can sum up the edges between each of the successive vertices and determine if the total weight is less than the threshold, D.

We have a similar problem in determining the Hamiltonian cycle. A Hamiltonian cycle is a just a tour through a graph's vertices without weights. The question is whether such a cycle exists:

$$HAM - PATH = \{\ G\, there\ exists\ a\ tour\ of\ G's\ vertices\ \}$$

Again, we can see that HAM-PATH is in NP.

Now we turn to Integer Linear Programming, or ILP. Suppose that we are asked to solve

$$\Sigma A_{ij} x_j \leq b_i,\ \forall i = 1, \ldots,\ m,\ and\ x_i\ are\ integers$$

So we are trying to solve m equations where we are given the values of A_{ij} and b_i and want to determine the values of x_j. Another way to think of this is that we are solving the equation $Ax = b$ for matrix A, vector b, and vector x. Further, the x_j values are restricted to integers. If the x_j values are real values and not integers, then we can refer to the simplex algorithm (not covered here) to solve the problem in polynomial time. However, when the x_j are integers, the problem becomes very difficult. So we define the problem as the following:

$$ILP = \{\ A,\ b\ : \Sigma A_{ij} x_j \leq b_i,\ \forall i = 1, \ldots,\ m,\ and\ x_1,\ \ldots,\ x_m \in\ \mathbb{Z}\}$$

Cook & Karp Reductions

Now that we've seen some problems in NP, let's return to SAT. Again, the worst-case time that solves SAT is exponential – $k^n,\ k > 1$, and the input size is n. For example, we may try both 0 and 1 for m literals, which gives us a time of at least $O(2^m)$. However, we've seen problems where we *could* have had exponential running time but we found a way to reduce it – in TQBF, we got rid of exponentially-long equations by using quantifiers, and in dynamic programming we got rid of too many possibilities by solving subproblems and using memorization.

So what makes a problem hard? What will convince us that a problem is hard? We could see an explicit lower bound, but is there some other technique? We want to try to connect together these problems.

Cook first defined NP-completeness, and Karp later defined 21 problems from many fields that were also NP-complete. Suppose that we have a function g that can transform a problem x in polynomial time. A ***Karp reduction*** shows that $x \in A \iff g(x) \in B$. Let's give an example of a reduction. Suppose that we have $x = (G, k)$, where G is a graph and k is an integer, and $A = INDEPENDENT - SET$. We want to know if $x \in A$. Now we can reduce the problem. Let $g(x) = x$, so the transformation is trivial. Now we ask if $g(x) \in VERTEX - COVER$, which we saw on the midterm. We know that the maximal independent set is the complement of the minimal vertex cover, so a graph G has a maximal independent set of size at least k iff G also has a minimal vertex cover of size at most k.

A Karp reduction asks for the computation once. We can think of the determination if $g(x) \in B$ as an oracle or a black box that is run a single time. If the oracle returns "yes", then $x \in A$. Otherwise, $x \notin A$. However, in a ***Cook reduction***, we are allowed to query the oracle multiple times. The total number of times that we can query the oracle is polynomial in the size of the input.

As another example of a reduction, we can reduce from $INDEPENDENT - SET$ to $CLIQUE$. A clique on k vertices means that all k vertices have an edge between them. So, given a graph G and an integer k, how do we reduce the problem of finding an independent set of size at least k to that of finding a clique of size at least k? As Tejas mentioned in class, we can take the complement of the graph G and test if the complement of G has a clique of size at least k. If there is no edge between vertices u and v in G, then there must be an edge between u and v in the complement of G.

SATISFIABILITY: First NP-Complete Problem

Now we want to know if there are any problems that are NP-Complete. If a

problem is NP-Complete, then it must satisfy two conditions:

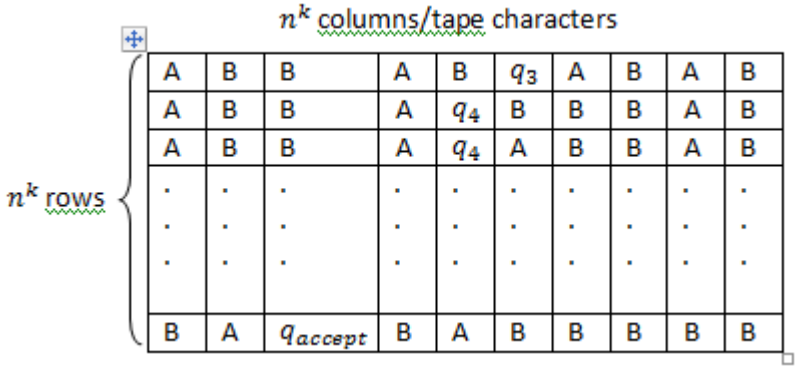
1. the problem is in NP;
2. every problem in NP can be reduced to any other NP-Complete problem in polynomial time.

We have not defined any NP-Complete problems yet. A problem that is in NP must be decided by a nondeterministic Turing Machine in polynomial time. So what are we going to reduce an arbitrary NP language to? Let's start with SATISFIABILITY. That means that we are going to transform *any* NP language to a Boolean formula. If we can do that, then we can determine if an NTM for a language $L \in NP$ will accept its input by transforming the NTM's computation into a Boolean formula. If the Boolean formula is satisfiable, then the NTM will accept. (That's cool.)

So how do we make this transformation? Well, we have to transform each piece of the NTM to a formula (of course). Suppose that an NTM can decide a language L in time n^k for some $k > 0$. Then we could consider a configuration table that shows the computation of the NTM over time. Each row of the table is equivalent to the NTM's configuration at some timestep in the computation. If the NTM accepts its input, then we will end up with a configuration row that corresponds to an accepting state.

It might help to review what constitutes a Turing Machine's configuration. A configuration consists of all of the tape to the left of the head, then the state of the Turing Machine, and finally the character under the head and all characters to the right of the head. So suppose that the tape reads "abbababbaaa", that the Turing Machine is in state q_3, and the Turing Machine's tape head is over the sixth character (or the third "a"). Then the configuration would be "abbabq7abbaaa" – the five characters to the left of the head, the state of the Turing Machine, and the characters under the tape head and to the right of the head.

Each entry in the configuration table is written as $x_{i,j,s}$, where i is the row number, j is the column number, and s is a state or tape character. The row number i corresponds to timestep i, the column number j corresponds to the j-th entry of a configuration, and s corresponds to a state or tape character. The table entry $x_{i,j,s}$ is 1 when the table entry on row i, column j contains the symbol s. Otherwise, $x_{i,j,s}$ contains 0. The setup is shown below.



Now we notice that each entry in the table can only contain at most one symbol. That means that our Boolean formula has to restrict itself to make sure that $x_{i,j,s}$ is 1 for only one s. How do we make sure that only one symbol is turned on? If we write this as $x_{i,j,1} \vee x_{i,j,2} \vee x_{i,j,3} \ldots x_{i,j,m}$, then two or more of the m variables could be turned on and the equation is still true. So we could use the following observation: $(x_1 \vee x_2) \wedge (\overline{x_1} \vee \overline{x_2})$ is true only when exactly one of x_1 and x_2 is active. If we need to test multiple values (say, n), then we test every pair of values x_i and x_j and we apply a logical AND to the result. If there are two variables, say x_i and x_j, that are both enabled, then the equation will fail. So we want to add the following constraint to our formula: for every row and every column in the table, add a Boolean formula that tests every possible pair of symbols:

$$\phi = \bigwedge_{1 \leq i,j \leq k} \left(\left(\vee_{\{all\ s\}} x_{i,j,s} \right) \wedge \left(\wedge_{\{all\ s,t\}} x_{i,j,s} \vee \overline{x_{i,j,t}} \right) \right) \qquad (1)$$

This is a large expression. We had to be careful with TQBF, which had an exponential number of entries before the quantifiers were applied. In this case, we have n^k entries, and each entry has at most $m + 2m^2$ symbol literals, where m is the number of possible symbols. In other words, this is large, but it's still polynomial in the size of the input.

Now we need to start our computation somehow. How do we do this? Well, we add a formula that includes the variable $x_{1,j,s}$ when the starting configuration contains symbol s in the jth entry. Then we combine them all together with logical ANDs. So if the starting configuration tape only contained "abc", then our formula would be:

$$x_{1,1,a} \wedge x_{1,2,b} \wedge x_{1,3,c} \qquad (2)$$

The space for the configuration is at most n^k, so we have at most a polynomial number of such literals.

We append this starting-configuration formula to ϕ. When we want to include the accepting configuration, though, we want to make sure that *any* accepting configuration is included. The symbol corresponding to accepting configuration is q_{accept}, but it could appear anywhere in the configuration. So we include the following in ϕ as well:

$$\bigvee_{1 \leq i,j \leq n^k} x_{i,j,q_{accept}} \quad (3)$$

Now we must make sure that all of the transitions between timesteps are legitimate. Legitimate transitions must follow the NTM's transition function. That means that the head can only move at most one tape character to the left or right, the state transition is correct, and a character may be written at the head's current position. Let's see this in the context of some cells of the computation's table. Suppose that we have a tape alphabet that consists of the letters {a,b}. Three cells of the current configuration may look like this:

a	q_2	b

So the head is currently over the b, the Turing Machine is in state q2, and the character to the left of the head is a. This is just part of the complete configuration. Now what are some valid transitions that can take place? Here are some possibilities:

1. The Turing Machine stays in the same state and its head moves to the left. Then the configuration is "$q_2$ab".
2. The Turing Machine stays in the same state and its head moves to the right. Then the new configuration is "ab$q$2" – the head is over a tape character to the right that just happens to not be visible yet.
3. The Turing Machine replaces the character "b" with a "c". Then the configuration is "aq2c".
4. The Turing Machine transitions from state $q2$ to $q4$. Then the new configuration is "aq4b".
5. The Turing Machine executes some combination of moving left or right, writing a character to the character under its current position, and changing state.

Now we want to place constraints on the Turing Machine's next configuration based on its current configuration. For a single transition, we want to just restrict our focus to two rows of the table – the current timestep's configuration and the next timestep's configuration. Then we check all of the possible

2x3 "windows" across the entire configuration table. So we add the following formula:

$$\bigwedge_{1<i\leq n^k,\ 1<j<n^k} (\textit{the window at position } i,j \textit{ is legal})$$

$$= \bigwedge_{all n^2 k cells} (\bigvee_{6 cells c_i are legal} (x_{i,j-1,c_1} \wedge x_{i,j,c_2} \wedge x_{i,j+1,c_3} \wedge x_{i+1,j-1,c_4} \wedge x_{i+1,j,c_5} \wedge x_{i+1,j+1,c_6})) (4)$$

So now we have a large formula that we have to satisfy, but is it of polynomial size? Well, we have n^k rows and we have up to n^k entries per configuration, so we have up to n^{2k} entries. Our first formula, 1, checked that only one transition was performed. That required $O(n^{2k})$ entries since each cell had a fixed-size formula. The formula from 2 only affects the top row, so it has size $O\left(n^k\right)$. The last two formulas also had formulas that were constant in size, so they required size $O\left(n^{2k}\right)$ entries. Then the entire formula is polynomial in the size of the input. Therefore we can construct a formula in polynomial time that determines if a nondeterministic Turing Machine accepts an input. So our reduction is complete, and SATISFIABILITY is NP-complete.

Sample Reduction: CLIQUE to SATISFIABILITY
So we have shown that solving SATISFIABILITY is as difficult as determining if an NTM will accept its input, and we have our first NP-complete problem. What other problems are NP-complete based on reducing from SATISFIABILITY to the target problem? It turns out that CLIQUE is NP-complete, and we will show it as follows.

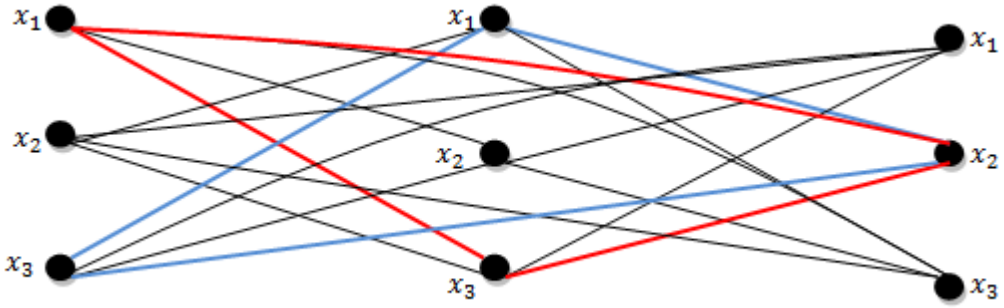
Suppose that we have the following formula:

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

We construct a graph as follows. We create one vertex per literal. We connect an edge between any two vertices subject to the following two constraints:

1. If the literals are in the same clause, then do not draw an edge between them;
2. If the literals have the same "label" (e.g., x_1 has the same label as x_1 and $\overline{x_1}$), then do not draw an edge between them.

So in our example we would have the following graph:



Can we find a clique of size 2? Sure – a clique of size two is just any edge (since we just need to connect two vertices together). What about a clique of size 3? Two such cliques are highlighted in the graph – they're just cycles of length 3. What about a clique of size 4? In this case, we cannot. Why not? Well, a clique of size 4 would require that at least one of the partitions has at least two vertices in the clique. (You can reason this out for yourself or try the pigeonhole priniciple.) We have constructed our graph in a way that two vertices in the same partition have no edge between each other, so no such edge can exist. Then a clique of size four cannot be found here.

When we find a clique among all m partitions, we have found a satisfying assignment for m clauses. Why? When we choose a vertex in the clique, we are stating that it is true. Then we choose one vertex from each partition such that it is connected to the vertices chosen in all other partitions. If such an edge exists, then it cannot refer to the same literal by construction, so we will not choose x_2 in two clauses/partitions and we will not choose both x_2 and $\overline{x_2}$ (since satisfying both of those would be impossible). So we have a satisfying assignment iff we have found a clique of size at least m. Then SATISFIABILITY reduces to CLIQUE.