CME 305: Discrete Mathematics and Algorithms

Instructor: Professor Aaron Sidford (sidford@stanford.edu)

February 29, 2018

Lecture 15 - NP Completeness[1]

In the last lecture we discussed how to provide evidence that problems are computationally difficult. We introduced the notions of polynomial time reductions and NP-completeness and sketched the proof that a particular problem, Circuit-SAT, is NP-complete. We mentioned that one of the virtues of finding a NP-complete problem is that given one, the recipe to prove that other problems are NP-complete is quite simple, reduce an NP-complete problem to it and show it is in NP, and that we can show that many natural problems are incomplete.

In this lecture we demonstrate this formally in several cases by showing that multiple problems are NP-complete. There has been extensive work on showing different problems are NP-complete and for more complete lists of NP-complete problems the curious reader should look at the suggested reading. The point of this lecture is to give a sense of how these reductions to prove problems are NP-complete work. The main theme in these reductions is that of coming up with clever gadgets to encode one NP-complete problem in another.

1 Recap

We begin with a quick recap of the fundamental principles we will use throughout this lecture. Recall that our main tool to show that problems were equivalent was through the notion of polynomial time reducibility defined below

Definition 1 (Polynomial Time Reducibility)**.** We say problem X is polynomial time reducible to problem Y, denoted $X \leq_p Y$ if and only if there exists an algorithm for solving X in in polynomial time plus the time needed to solve Y a polynomial number of times on polynomially sized input. We define $X =_p Y$ to denote the condition that $X \leq_p Y$ and $Y \leq_p X$.

We said we would focus on decision problems, i.e. problems where the answer is either "yes" or "no," and noted that such problems can be encoded simply as the X the subset of input binary strings which encode "yes" instances. For decision problems we formally defined the classes of P, NP, and NP-complete as follows

Definition 2 (P)**.** A decision problem X is polynomial time solvable, denoted $X \in P$, if and only if there is an algorithm that given any binary string s can compute or decide if $s \in X$ in time polynomial in $|s|$.

Definition 3 (NP)**.** A decision problem X is solvable in nondeterministic polynomial time, denoted $X \in NP$, if and only if there is a polynomial time verifier for X, that is there is an algorithm B that given binary strings s and t output $B(s,t) \in \{"yes","no"\}$ in time polynomial in $|s|+|t|$ with the property that if binary string $s \notin X$ then $B(s,t) = "no"$ for all t and if $s \in X$ then there exists t with $|t| = O(\text{poly}(|s|))$ such that $B(s,t) = "yes"$.

Definition 4 (NP-Hardness and Completeness)**.** A decision problem X is said to be *NP-hard* if for all $Y \in NP$ it is the case that $Y \leq_p X$. If additional $X \in NP$ then X is said to be NP-complete.

One of the key motivations of NP-completeness is the following lemma which provides an easy recipe for proving a problem is NP-complete.

[1] These lecture notes are a work in progress and there may be typos, awkward language, omitted proof details, etc. Moreover, content from lectures might be missing. These notes are intended to converge to a polished superset of the material covered in class so if you would like anything clarified, please do not hesitate to post to Piazza and ask.

Lemma 5. *If X is NP-complete and $X \leq_p Y$ for $Y \in NP$ then Y is NP-complete as well.*

Finally, we showed that the Circuit-SAT problem defined as follows is NP-complete.

Definition 6 (Boolean Circuit)**.** A *Boolean circuit* is defined by Boolean variables $x_1, ..., x_k$ and a connected tree with edges oriented from the leaves towards a root with the following properties:

- Each leaf node is labeled with one of x_i, T for "true," or F for "false"
- Each internal node is labeled with one of $\vee$ for Boolean "or", $\wedge$ for Boolean "and," or $\neg$ for Boolean "not"
- Each $\vee$ and $\wedge$ node has in-degree 2 and each $\neg$ node has in-degree 1.

Definition 7 (Circuit-SAT)**.** The *circuit satisfiability problem*, or *circuit-SAT*, is the problem of given a Boolean circuit determining if their is an assignment of the variables that causes the circuit to evaluate to true.

2 3-SAT is NP-Complete

As we have discussed, the primary goal of this lecture is to show how to use the fact that Circuit-SAT is NP-complete to prove that other problems are NP-complete as well. The class of NP-complete problems is impressively broad and their more reductions and useful gadgets for these reductions to prove that problems are NP-complete then we possibly have time to cover in this course. Instead, in the remainder of this lecture we provide a few canonical examples of NP-complete problems and prove that they are NP-complete.

Now, while the proof that Circuit-SAT is NP-complete is impressive, the definition of Circuit-SAT as a problem is somewhat clunky and therefore if we wish to have a simple problem to reduce from in showing that new problems are NP-complete, then Circuit-SAT is perhaps not the nicest choice. However, Circuit-SAT is simply the problem of checking if an arbitrary Boolean formula is satisfiable. There are nicer canonical forms for Boolean formulas that are in some sense equivalent and therefore we can use these to obtain slightly simpler NP-hard problems.

We begin by defining the conjunctive normal form (CNF) of a Boolean formula.

Definition 8 (Conjunctive Normal Form (CNF))**.** A Boolean formula F of the variables $x_1, ..., x_n$ is said to be in *conjunctive normal form (CNF)*, if there are clauses $C_1, ..., C_k$ where each clause C_i is a set of literals $l_1^{(i)}, ..., l_{j_k}^{(i)}$ where each $l_j^{(i)} \in \{x_1, x_2, ..., x_n, \neg x_1, \neg x_2, ..., \neg x_n\}$ and

$$F(x_1, ..., x_n) = \wedge_{i \in [k]} \left(\vee_{j \in [j_k]} l_j^{(i)} \right) .$$

In other words a Boolean formula is in conjunctive normal form if it is the and of a variety of clauses where a clause is the or of a number of literals, which are simply Boolean variables or there negation. For example,

$$(x_1 \vee x_2 \vee \neg x_3) \wedge (x_1 \vee \neg x_4)$$

is a Boolean formula in conjunctive normal form. Note that a CNF formula is true if and only if for every clause at least one of the literals is true and a CNF formula is satisfiable if and only if there is an assignment of variables to make this happen.

With this definition in hand we define the SAT problem as follows.

Definition 9 (SAT)**.** The SAT problem is the problem of given a CNF formula deciding if there exists a assignment of variables to true of false such that the formula evaluates to true, i.e. whether the CNF formula is satisfiable.

Now it is well known that every Boolean formula can be expressed equivalently in CNF and we essentially reprove this fact to prove that SAT is also NP-complete.

Lemma 10. *SAT is NP-complete.*

Proof. As usual with our proofs of NP-completeness we first need to argue that $SAT \in NP$. However, also as usual this proof is easy. Given an assignment of variables, we can easily evaluate a CNF formula in time linear in the size of the SAT formula and therefore this problem as a polynomial time verifier.

Now, also as usual the more difficult part of the proof is proving that SAT is NP-hard. However, since we now have a NP-complete problem, i.e. Circuit-SAT. To prove this we simply need to show that Circuit-SAT $\leq_p$ SAT. Consequently, suppose we have an instance of circuit SAT. Let $x_1, ..., x_n$ be the variables and let $v_1, ..., v_n$ be the nodes of the tree and let r be the root of the tree. We will overload notation slightly and create a boolean formula on the variables $x_1, ..., x_n, v_1, ..., v_n$ plus to additional variables t and f such that this formula is satisfiable if and only if the circuit is satisfiable.

Now to create this formula we will exploit the nice property of CNF formulas that every clause needs to evaluate true for the formula to evaluate to true. Thus we can think of adding clauses as adding constraints on variables that need to be met. Thus we simply need to add constraints so that the v_i correspond to the output of that node in the tree and such that r outputs true. Additionally we add constraints so that $t = T$ and $f = F$ as we may have these as input to certain v_i. We show how to add clauses corresponding to each of these as follows.

- **True and False**: we add the clauses t and the clause $\neg t$. Direct inspection reveals these clauses are true if and only if $t = T$ and $f = F$.
- **The Root Outputs true**: we add the clause r. Direct inspection reveals this clauses is true if and only if the root node outputs true.
- **Negation**: Suppose we want $v_i = \neg u$ for some variable u. In this case we add $(v_i \vee u) \wedge (\neg v_i \vee \neg u)$. Note that if $u = F$ then for $v_i \vee u$ we must have $v_i = T$ and we see that these clauses are true if and only if $v_i = T$. Similarly if $u = T$ then the clauses are true if and only if $v_i = F$. Combining we see that these two clauses are true if and only if $v_i = \neg u$.
- **And**: Suppose we want $v_i = u_1 \wedge u_2$ for some variables u_1 and u_2. In this case we add $(\neg v_i \vee u_1) \wedge (\neg v_i \vee u_2) \wedge (v_i \vee \neg u_1 \vee \neg u_2)$. Suppose u_1 is false, in this case for the first clause to be true we need v_i to be true and we see that when this happens the formula is true. Similarly, we see that if u_2 is false then the clauses evaluate to true if and only if v_i is false. Finally, if both u_1 and u_2 are true the last clause forces v_i to be true for the formula to be true and the whole formula is true when this happens. Consequently, the formula evaluates to true if and only if $v_i = u_1 \wedge u_2$.
- **Or**: Suppose we want $v_i = u_1 \vee u_2$ for some variables u_1 and u_2. In this case we add $(v_i \vee \neg u_1) \wedge (v_i \vee \neg u_2) \wedge (\neg v_i \vee u_1 \vee u_2)$. Suppose u_1 is true, in this case for the first clause to be true we need v_i to be true and we see that when this happens the formula is true. Similarly, we see that if u_2 is false then the clauses evaluate to true if and only if v_i is true. Finally, if both u_1 and u_2 are false the last clause forces v_i to be false for the formula to be true and the whole formula is true when this happens. Consequently, the formula evaluates to true if and only if $v_i = u_1 \wedge u_2$.

Putting this together, we see that given a circuit we can create a CNF formula in linear time whose size is linear in the size of the circuit such that the CNF formula is satisfiable if and only if the circuit is satisfiable. Consequently, to check if the circuit is satisfiable we can simply solve the SAT instance we have created and return this as the answer to circuit SAT. □

Now to show that a problem is NP-complete we simply need to show that it is in NP and reduce SAT to it. Next we show how that we actually only need to consider a restricted class of SAT instances known as 3-SAT.

Definition 11 (3-SAT)**.** The 3-SAT problem is the problem of checking if a CNF formula is satisfiable under the restriction that every clause in the CNF formula has exactly 3 literals.

Below we show that even though 3-SAT is a restriction on the broader class of SAT problems, nevertheless it is still NP-complete.

Lemma 12. *3-SAT is NP-complete.*

Proof. Clearly 3-SAT$\in NP$ as SAT$\in NP$ and SAT is a simple restriction of 3-SAT. Consequently to prove that 3-SAT is NP-complete it suffices to show SAT$\leq_p$3-SAT.

Now suppose we are given an instance of SAT and suppose some clause C_i has more than 3-literals, i.e. $C_i = l_1 \vee l_2 \vee ... \vee l_k$ for some $k > 3$ and literals $l_1, ..., l_k$. Suppose we introduce a new variable u and consider the formula $(l_1 \vee l_2 \vee u) \wedge (\neg u \vee l_3 \vee ... \vee l_k)$. Suppose that $l_1 \vee l_2$ is true, then setting $u = F$ makes this new formula true. Similarly if $l_3 \vee ... \vee l_k$ is true then setting $u = T$ makes the formula true. However, if both $l_1 \vee l_2$ and $l_3 \vee ... \vee l_k$ are false then no matter what u is set to the formula is false. Consequently, we see that replacing the clause C_i with this new formula doesn't change whether or not the formula is satisfiable, but it does replace a clause of length k with one of length 3 and one of length $k-1$. Repeating this, we see that given a CNF formula we can repeat this procedure to create a CNF formula that is satisfiable, where no clause has no more than 3 literals, such that the total size of the formula has only increased by at most a multiplicative constant.

Now, suppose some clause has less than 3 literals. In this case we can always introduce a new variable t and add the clause t to ensure that $t = T$ in any satisfying assignment. Then for any clause with less than 3 literals we can just add $\vee \neg t$ terms until it has length 3. Since $\neg t = F$ in any satisfying assignment this whole procedure does not change whether or not the CNF formula is satisfiable.

Combining these two reduction tricks we see that given any SAT instance in linear time we can create a 3-SAT instance of linear size where the answer to each problem is the same. □

With the fact that 3-SAT in NP-complete in hand we are now ready to prove the NP-complete of more combinatorial optimization problems.

3 Independent Set is NP-Complete

As our first example of showing that a more natural combinatorial optimization problem is NP complete, here we show that the independent set problem considered in the previous lecture is NP-complete. Recall the following definition from the previous lecture.

Definition 13 (Independent Set)**.** For undirected $G = (V, E)$ a set $S \subseteq V$ is an *independent set* if and only if every edge $e = \{i, j\} \in E$ it is not the case that both its endpoints are in S, i.e. at least one of its endpoints

is not in S or equivalently, $|e \cap S| \leq 1$. The *independent set (decision) problem* asks given a number k as part of the input is there an independent set of size $\geq k$.

We now prove that the independent set problem is NP-complete. As with many proofs of NP-completeness the key is provide a gadget that can be used to encode one NP-complete problem as another.

Lemma 14. *The independent set problem is NP-complete.*

Proof. We have already argued that indpendent set is in NP (and the proof is trivial) so we just need to show that 3-SAT$\leq_p$independent set. Now suppose we have an instance of 3-SAT, i.e. a CNF formula $F = \wedge_{i\in[k]} C_i$ where each $C_i = l_1^{(i)} \vee l_2^{(i)} \vee l_3^{(i)}$ for literals $l_j^{(i)} \in \{x_1, ..., x_n, \neg x_1, ..., \neg x_n\}$. We need to encode this problem as an instance of independent set.

Recall that F is satisfiable if and only if there is a way to assign values to the x_i such that at least one literal in each C_i is true. Consequently, we will try to encode this choice of true literal in a independent set instance. To do this we create a graph G with a vertex $v_j^{(i)}$ for each $i \in [k]$ and $j \in [3]$ corresponding to literal $l_j^{(i)}$ and add an edge $\{v_{j_1}^{(i)}, v_{j_2}^{(i)}\}$ for all $j_1, j_2 \in [3]$ with $j_1 \neq j_2$. Now note in any independent set for each i at most one of the $v_j^{(i)}$ can be included. Consequently, the maximum indpendent set in this graph is exactly k. Now what we will do is add edges so that the choice of true literals are consistent with some assignment of the variables. To do this for all $l_j^{(i)} = \neg l_b^{(a)}$ we also add an edge.

Now we claim that the size of the maximum independent set in this graph is at least k if and only if F is satisfiable. First, suppose F is satisfiable. Then there is an assignment of x_i so that at least one literal in each clause is true. If we take these clauses as the elements of a set, then clear the set is independent as we did not include both endpoints of the edges within a clause and we clearly did not include both endpoints of an edge induced by a $l_j^{(i)} = \neg l_b^{(a)}$ as at most one of these two literals is true.

On the other hand suppose that the size of the maximum independent set is at least k. Then, as we have argued there exist $j_1, ..., j_k \in [3]$ such that the vertices $(v_{j_1}^{(1)}, v_{j_2}^{(2)}, ..., v_{j_k}^{(k)})$ are independent. Now set the x_i to make each of these literals true letting the x_i be arbitrary when no literal corresponding to these vertices includes x_i. Clearly this can be done since at most one of the literal $l_j^{(i)}$ and its negation can be included by the edges we add. Furthermore, we see that this assignment of x_i is clearly satisfying.

Consequently, given any 3-SAT instance we can construct a graph of size polynomial in the size of the 3-SAT instance such that there is an independent set of size at least k if and only if the 3-SAT instance is satisfiable. □

Recall that in the last lecture we showed that Minimum Vertex Cover $=_p$ Maximum Independent Set and consequentlythe above lemma also shows the decision version of vertex cover, i.e. is there a vertex cover of size $\leq k$, is also NP-complete.

4 Hamiltonian Cycle is NP-Complete

Here we consider another canonical hard combinatorial optimization problem, known as the Hamiltonian cycle problem and prove that it is NP-complete. Formally a Hamiltonian cycle and the decision problem associated with it are defined as follows.

Definition 15 (Hamiltonian Cycle)**.** Given a directed graph $G = (V, E)$ a Hamiltonian cycle is a cycle containing every vertex in the graph. The Hamiltonian cycle problem is that of determining whether or not a graph contains a Hamiltonian cycle.

Note that a Hamiltonian cycle can be thought of as a permutation on the vertices. For example if $V = \{v_1, .., v_n\}$ then $G = (V, E)$ has a hamiltonian cycle if and only there is a permutation $i_1, ..., i_n$ such that $(v_{i_j}, v_{i_j+1}) \in E$ for all $j \in [n-1]$ and $(v_{i_n}, v_1) \in E$. Next we show that the Hamiltonian cycle problem is NP-complete.

Lemma 16. *The Hamiltonian Cycle problem is NP-complete.*

Proof. Clearly the Hamiltonian cycle problem is in NP as we can clearly check if a set of edges forms a cycle and goes through every vertex in linear time. In the remainder of the proof we show that 3-SAT$\leq_p$Hamiltonian Cycle. Consequently, suppose we have an instance of 3-SAT, i.e. a CNF formula $F = \wedge_{i\in[k]} C_i$ where each $C_i = l_1^{(i)} \vee l_2^{(i)} \vee l_3^{(i)}$ for literals $l_j^{(i)} \in \{x_1, ..., x_n, \neg x_1, ..., \neg x_n\}$, we will show how to encode the satisfiability of this formula as an instance of Hamiltonian cycle.

Perhaps the first difficulty in thinking about this problem is how to encode choosing a Boolean assignment for variables as a Hamiltonian cycle. We first show this is possibly by constructing a graph that has 2^n different Hamiltonian cycles, corresponding to every possible truth assignment of the variables and then we show how to modifying this graph to add the constraint that the truth assignment makes at least one literal in each clause true.

First we construct a directed graph as follows. We add a special vertex s and a vertex t and for some parameter $k = \Theta(n)$ that we will choose later and for all $i \in [n]$ we add a vertex $v_k^{(i)}$. Now we add the following edges, $(s, v_1^{(1)})$, $(s, v_k^{(1)})$, $(v_1^{(n)}, t)$, $(v_k^{(n)}, t)$, and (t, s). Furthermore for all $i \in [n]$ and $j \in [n-1]$ we add $(v_j^{(i)}, v_{j+1}^{(i)})$ edge and a $(v_{n+1-j}^{(i)}, v_{n-j}^{(i)})$ edge and for all $i \in [n-1]$ we add the edges $(v_1^{(i)}, v_1^{(i+1)})$,$(v_1^{(i)}, v_n^{(i+1)})$,$(v_n^{(i)}, v_1^{(i+1)})$, and $(v_n^{(i)}, v_n^{(i+1)})$ This graph is essentially the vertices s and t as well as bidirectional paths $P_1, ..., P_n$ each of length k where P_i corresponds to $v_1^{(i)}, ..., v_n^{(i)}$ and the edges between them. Furthermore, we have s, connected to each end of P_1 and we have each end of P_i connected to each end of P_{i+1} with each end of P_{i+1} connected to t and t connected to s.

Now it is easy to see that this graph has exactly 2^n different Hamiltonian cycles. Starting at s we can then go to either end of P_1 then following P_1 until we get to one of the ends of P_2, etc. until we go to t and back to s. Essentially for each path we can choose whether to traverse it from left to right or from right to left and each such choice induces a Hamiltonian cycle. Consequently, to map this to an assignment of Boolean variables we can think of traversing path P_i from left to right as setting x_i to true and traversing path P_i from right to left as setting x_i to false.

We simply need to add to this graph to force a traversal of a path to correspond to setting one literal in each clause to true. To do this, suppose we have some clause $C = l_1 \vee l_2 \vee l_3$. Further suppose we then create some new vertex c and for some $m \in [k] \setminus \{k, 1\}$ if l_i is a non-negated variable we add edges $(v_m^{(l_i)}, c)$ and $(c, v_{m+1}^{(l_i)})$ and if l_i is a negated variable we add the edges $(v_{m+1}^{(l_i)}, c)$ and $(c, v_m^{(l_i)})$. Further suppose we do this for all i and consider an arbitrary Hamiltonian cycle in this new graph. The cycle needs to go through vertex c and suppose without loss of generality that the edge we use is $(v_m^{(l_i)}, c)$. Now, the only vertex other than $v_m^{(l_i)}$ and c which has an edge to $v_{m+1}^{(l_i)}$ is $v_{m+2}^{(l_i)}$. However if the cycle cannot have the $(v_{m+2}^{(l_i)}, v_{m+1}^{(l_i)})$ edge as after following this edge the next edge would have to be $(v_{m+1}^{(l_i)}, v_m^{(l_i)})$ which it cannot be as we have already visited $v_{m+1}^{(l_i)}$. Consequently, the cycle must contain both the $(v_m^{(l_i)}, c)$ and the $(c, v_{m+1}^{(l_i)})$ edge for one of the l_i.

Consequently, if we add the gadget described above for C for each of the C_i where for each C_i some m is chosen separated from the m of the other C_i by a constant, then we see that the resulting graph has a Hamiltonian cycle if an only if the 3-SAT formula is satisfiable in which case the cycle simply goes through each P_i either left to right or right to left, occasionally skipping an edge to go through a C_i. Since picking $m = 4n$ is enough to completely separate the m (though a smaller number may suffice) we see that the the

size of the graph is polynomial in the size of the SAT instance and the result follows. □

Using the fact that the Hamiltonian Cycle problem is NP-complete we can show that other related, common combinatorial optimization problems are NP-complete as well. For example consider the following problem known as the traveling salesman problem.

Definition 17 (Traveling Salesman Problem)**.** In the traveling salesman problem (TSP), we are given a graph $G = (V, E)$ a set of positive edge lengths $l \in \mathbb{R}^V_{>0}$ and wish to compute a tour, that is an ordering on the vertices, that is $v_1, ..., v_n$ such that the the total length of the tour, that is sum of the shortest path distances between v_i and $v_{i+1 \mod n}$ is minimized. In the decision version we wish to know if there is a tour of length at most c.

This problem encapsulates the problem of having a set of locations one wish to visit and then return back to start where the total distance traveled is minimized. It is also easily seen as a generalization of Hamiltonian cycle simply by setting the length of every edge in the graph to be 1, in this case there is a tour of length at most n if and only if the graph has a Hamiltonian cycle. However, since the problem is also clearly in NP it is NP-complete.

This should hopefully give you a taste for how to show problems are NP-complete. NP-complete problems are quite pervasive and on the website we'll have a link to a webpage that gives a long list of NP-complete problems.

..