CS 124 Section #11 **NP Completeness** 4/10/17

1 Summary of Material

1.1 P vs. NP Definitions

There are 2 classes of problems that we will examine in this course: **P** and **NP**. In today's section, we will review the difference between the two, as well as get some more practice classifying something as **P**, **NP**, and **NP**-Complete.

- **P** - the set of all yes/no problems that can be deterministically solved in polynomial time, that is runs in $O(n^k)$ steps for some positive integer k.
  - **Examples from Lecture:** Basically everything we have seen so far. Shortest paths, finding MST's, dynamic programming, max flow
- **NP** - the set of all yes/no problems where one could convince you that the answer is "yes" by giving a polynomial length certificate which can be verified in polynomial time as well.
  - **Examples from Lecture:** Compositeness, 3-SAT, Integer Linear Programming, Maximum Independent Set, Vertex Cover, Maximum Clique

Note that P is a subclass of NP because any we can verify that a P algorithm is correct by simply running it, which takes polynomial time.

1.2 Reductions

A reduction is a procedure $R(x)$ which transforms the input for problem A into an input for problem B. This transformation must maintain the integrity of the answer, that is the answer to A is yes from an input x if and only if the answer to B from $R(x)$ is yes. If we can find such a procedure $R(x)$, then we can conclude that A is at least as easy as B. This is true because we know that **if we can solve B, we can solve A** by making this reduction.

To Show that A is Easy: Reduce A to something easy.

To Show that A is Hard: Reduce something else that is hard to it.

1.3 NP-Complete

These are the hardest problems in NP, which have the property that all other problems in NP reduce to them. Our time-line of proving various problems to be NP-Complete is summarized below:

(a) **Circuit-SAT**: Cook's Theorem (informal explanation offered in lecture)

(b) **3-SAT**: Circuit-SAT reduced to 3-SAT by introducing various combinations of 3 clauses to represent x being an AND, OR, or NOT gate of y and z.

(c) **Integer Linear Programming**: 3-SAT reduced to this by replacing each literal with x or $1 - x$ and then setting the sum of each clauses to be greater than 1.

(d) **Independent Set**: 3-SAT reduced to this by constructing a strange graph where nodes represented assignments of the literals, and edges connected assignments that conflicted with each other.

(e) **Vertex Cover:** : Independent Set and Vertex Cover reduce to each other by observing that C is an independent set if and only if $V - C$ is a vertex cover.

2 Practice Problems

Exercise 1. Show that the following optimization problem is in P:

Call two paths *edge-disjoint* if they have no edges in common. Given a directed graph $G = (V, E)$ and two nodes $s, t \in V$, find the maximum number of edge-disjoint paths from s to t.

Solution
In order to show that a problem is in P, it is sufficient to reduce it to another problem in P. We will reduce the edge disjoint paths problem to maximum flow by assigning every edge in the graph a capacity of 1. Then, the maximum flow of the graph is precisely the number of edge disjoint paths.

to see this, note that no two augmenting paths can share an edge, so a max flow of f^* corresponds to exactly f^* edge-disjoint $s - t$ paths. If there were more than f^* edge-disjoint $s - t$ paths, then putting one unit of flow across each path would lead to a valid flow of the graph, contradicting the fact that f^* was a *maximum* flow.

Exercise 2. Recall the *Set Cover* problem: Given a set U of elements and a collection $S_1, ..., S_m$ of subsets of U, is there a collection of at most k of these sets whose union equals U? You may remember that there is a greedy algorithm which is off by a factor of $O(\log n)$. Show that Set Cover is actually NP-Complete.

Solution
First, it is clear that set cover is in NP because a short certificate would be the sets themselves. We can check in polynomial time whether those $\leq k$ sets cover all the elements in the universe.

We will show that Vertex Cover reduces to Set Cover by (1) Explaining how to convert an instance of VC into SC in polynomial time, and (2) showing that the two problems become equivalent (i.e. "yes" in VC if and only if "yes" in SC).

Given an instance of vertex cover with (V, E, k), we construct our universe U to be the set of all edges in E and our sets S_u to be the edges adjacent to the vertex $u \in V$.

Proof. If there exists a valid vertex cover, then the corresponding set cover by choosing the sets constructed from the edges of the corresponding vertices in the vertex cover must form a set cover. This is true by construction because if all the edges have been covered in E, then all the items have been covered in U.

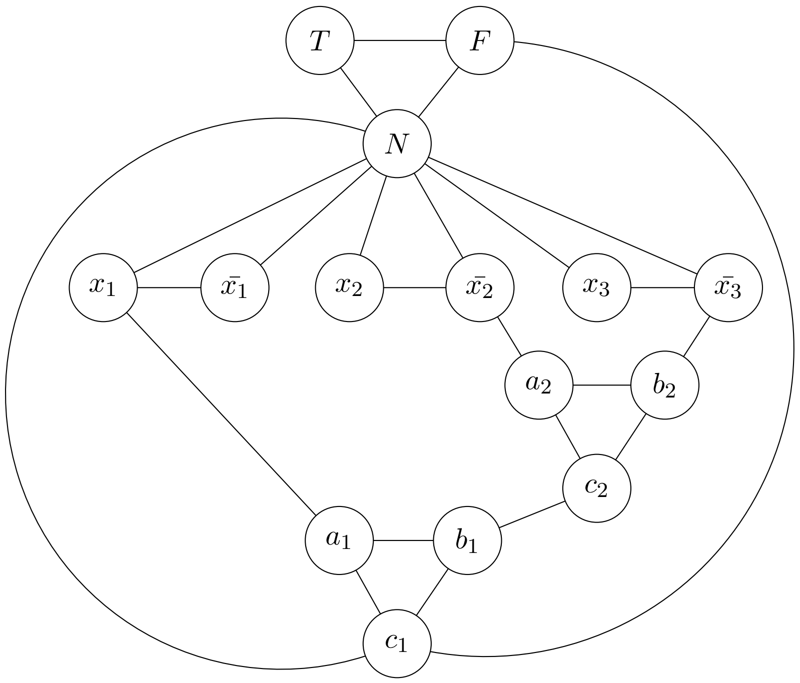
On the other hand, if there exists a valid set cover, then we can choose the vertices that are represented by each set. Since each edge corresponded to at least 1 set, each edge in the graph will have at least one neighbor that was chosen.

Exercise 3. A *3-coloring* of a graph $G = (V, E)$ is a assignment of colors to the vertices

$$f : V \to \{\text{red, green, blue}\}$$

such that for every edge $(u, v) \in E$, $f(u) \neq f(v)$. Show that $3-$coloring is NP-complete.

Hint: Consider the following graph with the clause $(x_1 \vee \bar{x_2} \vee \bar{x_3})$:



Solution
First, create three nodes that we will label T, F, and N. Connect these to make a triangle, so any 3-coloring would assign them distinct colors. Let those three distinct colors be called t, f and n which stand for true, false and none. For each variable x_i, add a node corresponding to x_i and a node corresponding to $\bar{x_i}$. Connect x_i to $\bar{x_i}$, and connect all of these nodes with N. Any 3-coloring must therefore color one of x_i and $\bar{x_i}$ with the color of T, and the other with the color of F. This will correspond to the truth assignment.

For every clause, we will create 6 more nodes, labeled $a_1, b_1, c_1, a_2, b_2, c_2$ in the hint for the clause $(x_1 \vee \bar{x_2} \vee \bar{x_3})$. Let's examine why these 6 nodes imply that at least one of $x_1, \bar{x_2}$, or $\bar{x_3}$ must be colored t. Intuitively, the nodes a_2, b_2, c_2 represent the clause $(\bar{x_2} \vee \bar{x_3})$, while the nodes a_1, b_1, c_1 represent the clause $(x_1 \vee (\bar{x_2} \vee \bar{x_3})$.

It is immediate that node c_1 must be colored t because e is connected to both F and N. This implies that among a_1 and b_1, one of them must be colored f and the other colored n. If x_1 is colored t, then we can color a_1 with f and b_1 with n. It is not hard to verify that for whatever coloring of $\bar{x_2}, \bar{x_3}$, it is possible to color a_2, b_2, c_2 accordingly. If x_1 is colored f, then it must be that a_1 is colored n and b_1 is colored f. This implies that c_2 cannot be colored f. If $\bar{x_2}$ and $\bar{x_3}$ are both colored f, then that would imply c_2 is colored f, which is not allowed. If one of $\bar{x_2}, \bar{x_3}$ is colored t, then we can color c_2 as t as then color one ofa_2 and b_2 f and the other n. Thus, either we have x_1 colored t, or at least one of $\bar{x_2}$ and $\bar{x_3}$ need to be colored t. This simulates the clause $(x_1 \vee \bar{x_2} \vee \bar{x_3})$.

This reduction takes an instance of 3-SAT and converts it to an instance of 3-coloring. The size of the input grows by a factor of 6 and thus the reduction can be achieved in polynomial time. It is easy to see from the reasoning above that a valid 3-SAT assignment corresponds to a valid 3-coloring.

Exercise 4. The class NP was defined as a class of decision problems. However, typically we have an optimization problem we would like to solve and not just a decision problem. For example, in the $\textsc{VertexCover}_k$ problem we must decide whether a vertex cover exists of size at most k, as opposed to the $\textsc{MinVertexCover}$ problem of finding a vertex cover of minimum size.

(a) Show that a polynomial time algorithms for $\textsc{VertexCover}_k$ for all k imply a polynomial time algorithm for $\textsc{MinVertexCover}$.

(b) In the $\textsc{HamiltonianCycle}$ problem we must decide whether a directed graph G has a cycle of length n that touches every vertex exactly once. This is in contrast with the non-decision $\textsc{FindHamCycle}$ problem of actually *finding* a cycle. Show that a polynomial time algorithm for $\textsc{HamiltonianCycle}$ implies a polynomial time algorithm for $\textsc{FindHamCycle}$

Solution

(a) First, check if G has a vertex cover of size $k = 1$ with the decider D. Increment k until we have found the size of the minimum vertex cover. Since we know that every graph has a vertex cover of size n, this process will terminate. Once we know that G has a minimum vertex cover of size k, we can find the vertex cover using the following algorithm:

1. Choose an unmarked vertex v from the graph G with minimum vertex cover k. Delete v and all of its edges, leaving G'.
2. If G' has a vertex cover of size $k - 1$, then v is in a minimum vertex cover of G. Continue the algorithm with G'.
3. If G doesn't have a vertex cover of size $k - 1$, v isn't in a minimum vertex cover of G. Continue the algorithm with G after marking v.

If we follow this algorithm, we will end up with a graph G^* with no more edges. All deleted nodes make up a vertex cover. If we have a graph G with a minimum vertex cover of size k, then deleting any node in the vertex cover (as well as its edges) must give us a graph with a minimum vertex cover of size $k - 1$. However, deleting any other node and its edges does not change the size of our minimum vertex cover. Therefore, we delete all nodes in the vertex cover and their edges. By the definition of a vertex cover, we have no edges left. This algorithm runs in polynomial time. We know that our decider D runs in time $O(n^c)$ for a constant c. We first check it at most n times, giving us $O(n^{c+1})$ operations. Then we run our main algorithm. Since we delete or mark every vertex, and we run D once for every vertex, we run D at most n times. Therefore, our overall algorithm runs in $O(n^{c+1} + n^{c+1}) = O(n^{c+1})$. Given D, we have a polynomial time algorithm for $\textsc{MinVertexCover}$.

(b) If we have a polynomial time algorithm D to decide if $G \in \textsc{HamiltonianCycle}$, we can use D as a black box to find a Hamiltonian path in G. To prove this, we will describe a polynomial time algorithm to do so. Take a graph $G \in \textsc{HamiltonianCycle}$ with n nodes. Order all edges and initialize them as unmarked, then follow this algorithm until all edges are marked:

1. Delete the first unmarked edge e, giving the graph G'.
2. If $G' \in \textsc{HamiltonianCycle}$, e wasn't necessary for a cycle, so we continue with G'.
3. If $G' \notin \textsc{HamiltonianCycle}$, e is necessarily in G's Hamiltonian cycle, so restore the edge and mark it. We continue with G.

If we follow this algorithm until all edges are marked, we will be left with a graph G^* with the same set of nodes as G and n edges forming a Hamiltonian cycle. How do we know this?

- Each iteration of the algorithm leaves a graph with a Hamiltonian cycle, so $G^* \in$ HamiltonianCycle.
- G^* must have the same set of nodes as G, since our algorithm never delete nodes.
- Since each edge in G^* is marked, removing any of them results in a graph not in HamiltonianCycle. Thus, each is necessary for a Hamiltonian cycle. Since a Hamiltonian cycle has length n, there are exactly n edges in G^*.

This algorithm runs in polynomial time. A graph of size n has $O(n^2)$ edges. Our algorithm thus iterates $O(n^2)$ times. Each iteration requires checking our decider D, which runs in $O(n^c)$ time, once, for some constant c. Thus, the algorithm to find a Hamiltonian cycle runs in $O(n^{c+2})$ time.