Luke Jaffe
Theory of Computation
Due 04/15/16

Problem Set 8

Problem 1.
The social media company you just started working for wants to be able to analyze friend networks to find groups of 7 people who have all friended each other — that is, every pair of people in the group of seven has friended each other. “Great,” your coworker says. “How are we supposed to do that efficiently? Everyone knows CLIQUE is NP-complete!” Your coworker is a little bit right, and a little bit not. Let k-CLIQUE be the problem {<G>: G is an undirected graph that contains k fully connected vertices.} (Notice that k is not an input to the decider; it is “hard-coded” for a particular k.)

a) Show that 7-CLIQUE is in NP by describing a certificate checkable in polynomial time.

A certificate for the problem is a set of 7 vertices in G. Checking whether these 7 vertices are fully connected takes polynomial time. More precisely, this certificate takes 6+5+4+3+2+1 = 21 operations to check, since we must check if each vertex is connected to each other vertex, but we don't need to “re-check”.

b) Show that 7-CLIQUE is also in P, giving a polynomial time algorithm and a big-O bound on the running time. (It does not need to be very fast, and you can be a bit loose with your bound.)

To use the certificate above, we can check all unique sets of 7 vertices in G. If any of these sets is fully-connected, we have found a 7-CLIQUE. If none of them are, then no 7-CLIQUE exists in G. If G contains n vertices, then there are (n choose 7) ways of selecting unique sets of 7 vertices from G. The time complexity of (n choose 7) is $\Theta(n^7)$. Since checking the certificate takes constant time, the time complexity of the full algorithm is still $\Theta(n^7)$.

c) Generalize your algorithm from (b) to show that for any constant k, k-CLIQUE is in P. Be sure to give the formula for the big-O running time.

Generalizing the statement used above, if G contains n vertices, then there are (n choose k) ways of selecting unique sets of k vertices from G. The time complexity of (n choose k) is $\Theta(n^k)$. Checking the certificate takes k-1+k-2+...+2+1 operations, or $\sim(k)(k-1)/2 = \Theta(k^2)$. We use the exact same algorithm as above, checking all sets of k vertices in G to see if any is a k-CLIQUE. The runtime of this algorithm is $\Theta(k^2n^k)$.

d) Recall that CLIQUE= {<G,k>: G is an undirected graph with k fully connected vertices}. Explain why your results from (b) and (c) do not imply CLIQUE is in P (which, since CLIQUE is NP-complete, would prove P=NP).

When k is an input to the problem, it could be up n in size (n being the number of vertices in G), meaning that the problem is to look for n-fully connected vertices. While there is obviously only one possible set of vertices for k=n, the worst case comes up for k = n/2. In this case, there are (n choose n/2) possible different combinations of vertices. Since this value is exponential in n, the problem clearly is not in P.

Problem 2.
Agree or disagree and explain.

(a) The fact that a Diophantine equation solver could be used to solve SAT problems implies that solving Diophantine equations is NP-complete.

Disagree. This only shows that a Diophantine equation solver is NP-Hard. To show that it is NP-complete, there must also be a certificate for the problem checkable in polynomial time.

(b) NP-complete problems can be undecidable.

Disagree. Any NP-complete problem can be used to solve any other NP-complete problem in polynomial time. Therefore, since we know there are NP-complete problems which are decidable, all NP-complete problems must be decidable.

(c) The fact that Sudoku puzzles are a special case of EXACT COVER implies that Sudoku is also NP-complete.

Disagree. Sudoku is NP-complete, but not because Sudoku puzzles are a special case of EXACT COVER. Being a special case of EXACT COVER does not mean you can reduce from EXACT COVER to n^2xn^2 Sudoku. And even if you could, that would only show Sudoku is NP-hard. You would also also need to show give a polynomial checkable certificate for Sudoku to show it is in NP, which the above statement does not.

(d) One proof that PRIMES ({n: n is a prime number written in binary}) is in P is that just checking all divisors up to the square root of n is clearly $O(n^{1/2})$.

Disagree. This is pseudo-polynomial, because it is polynomial in the size of the input, not in the number of bits. So PRIMES is not in P.

Problem 3.
Some slight variations on other NP-complete problems, which are also NP-complete.

a.) Suppose we associate a single “item” (not necessarily unique) with each vertex in a graph; let M be a set of tuples (v,i) that describe this mapping. Prove SHOPPING = {<G,M,S>: G contains a simple cycle that hits all items in set S at least once} is NP-complete.

A certificate for the SHOPPING problem is a set vertices in G. We can check if the set of vertices contains a simple cycle in G, and hits all items in S at least once, in polynomial time.

To show that SHOPPING is NP-hard, we can use it to solve a known NP-complete problem, Hamiltonian Cycle. To do this, we configure the set S to map one-to-one the the vertices of G. This way, in order to find a simple cycle hitting all items in S, we must hit all vertices in the graph, which describes a Hamiltonian Cycle.

Since SHOPPING has a certificate checkable in polynomial time, and is also NP-hard, it is NP-complete.

b.) Let QUARTER = {<S>: S is a set of nonnegative integers that can be partitioned into four disjoint sets with the same sum}. Prove that QUARTER is NP-complete.

A certificate for the QUARTER problem is a partition of S into 4 disjoint sets. We can check if each of these sets has the same sum in polynomial time.

To show QUARTER is NP-hard, we can use it to solve a known NP-complete problem, Partition.

Using PARTITION is a logical choice, since it does the same thing, except with 2-disjoint sets instead of 4. To solve PARTITION, take the input set S to PARTITION and duplicate each element. Then, run QUARTER on this set with duplicated elements. If QUARTER accepts PARTITION accepts, and if QUARTER rejects, PARTITION rejects. We know this will work correctly if PARTITION would accept on the set, because we could just take the correct 2-partitions and duplicate them to get a valid solution. But we also need to show that QUARTER will reject if PARTITION would reject. We will observe two cases, one in which the sum of the elements in S is even, and one in which it is odd.

If the sum is odd, then PARTITION must reject anyway, because each partition would have sum odd/2, which could not be composed of integers. QUARTER will also reject in this case, because the partitions in both problems must have the same sum.

If the sum is even and PARTITION rejects, then QUARTER would also reject because if there is no subset which sums to n/2, then duplicating the elements of S will not magically produce 4 subsets which sum to n/2.

Since QUARTER has a certificate checkable in polynomial time, and is also NP-hard, it is NP-complete.

Problem 4.

You are interning at a company that makes computer role-playing games, and they want to solve the following problem. A quest is a non-repeatable adventure in the game with at least two paths the player can follow. Each path has a different reward of some number of different items, some of which may be unique to the quest path, and some not. Some sets of items give the player special benefits if the player has all n of them — for example, quest 1 path 1 might give a Helmet of Awesome, and collecting all 7 Armor Pieces of Awesome may give the player some additional benefit — and the company asked you to write a script that checks whether it is actually possible to get a particular collection of n different items in a single playthrough of the game, given the lists of rewards along each quest path.

You wrote that script, and someone is complaining it is too slow for large n. So defend yourself by proving that this problem is NP-complete.

A certificate for this problem is a collection of n different items. Checking this set of items against the desired set can be done in polynomial time. To show that the problem is NP-hard, we can use it to solve a known NP-complete problem, SAT.

The construction is as follows:

Assign a unique item to each clause i.e. clause C_1 is assigned to item I_1. If there are n different clauses $C_1,C_2,...,C_n$, then the set of items our game tests for is $I_1,I_2,...,I_n$. We will construct our quests such that if all items in the set are obtained, then all clauses in the SAT problem are satisfied.

For each variable in the SAT problem, we will construct a new quest. Each quest will have two paths, one for if the variable is true, and one for it is false. The reward for the “true” path will be all items which are assigned to clauses satisfied by that variable being true. The reward for the “false” path will be all items which are assigned satisfied by that variable being true. If the variable satisfies no clauses in one of those states (true or false), then no items will be awarded for following that path.

Now we will show that you can collect the n items in the quest game constructed from the SAT problem iff the SAT problem is satisfiable.

1. If it is possible to collect a set of n items constructed from a SAT problem in this fashion, the SAT problem is satisfied: Each variable in the SAT problem can be either true or false. This is modeled by creating a quest for each variable, and having a true path and a false path, which makes sense, since it models the irreversible decision of selecting a variable's status as either true or false. In SAT, if a variable is true, then all clauses which contain that variable as true (not inverted) are satisfied. Likewise, if a variable is false, then all clauses which contain that variable as false (inverted) are satisfied. This is modeled by assigning an item to each clause. Any time a clause is satisfied by a given variable assignment, we collect that item from the quest. Since there is a one-to-one mapping between the n clauses in the SAT problem and the n items in the quest problem, if we have collected all items, then all clauses must have been satisfied.

2. If a SAT problem is satisfied based on this construction, then all n items have been collected: First, we will show that satisfying a clause correponds to collecting the item assigned to that clause. If a clause is satisfied, then at least one of the its elements (negated or non-negated variable) is true. We have already described the quest path construction, in which each variable allows attainment of all items which correspond to clauses the variable satisfies for the given setting of true or false. Therefore, when any element of a clause is true, we know that the corresponding item for that clause has been obtained in the quest path corresponding to that setting of the variable. Since there is a one-to-one mapping between the n clauses in the SAT problem and the n items in the quest problem, if all n clauses have been satisfied, then all n items have been collected.

Problem 5.
NP problems are all decision problems, but being able to decide a problem in polynomial time often means that the solution to the implicit problem can be found in polynomial time. For each NP-complete problem, show how a TM that simply decides the problem in polynomial time (ACCEPT or REJECT) could be used as a subroutine to solve the associated problem in polynomial time. Be sure to explain both why your approach should work and why it is polynomial time if the problem’s decider is polynomial time.

(a) EXACT COVER: Find the covering set in polynomial time.

To find the covering set in polynomial time, we only need to make one pass through the set of sets, S. First we apply the EXACT COVER decider to S to see if it decides the problem at all. If it rejects, then there is no valid covering set. Otherwise, for each set in S, we drop that set, and then apply the EXACT COVER decider to S. If it still accepts, then we leave that set out of S, and try the next one. If it rejects, then we put that set back into S, and try the next one. When we have iterated through all sets in S, removing ones which still allow acceptance, then we must have arrived at a set of sets in which each one is a member of the exact cover, since removing any one of them would cause the decider to reject. Since this single pass takes only linear time, the solution is polynomial time with the decider.

(b) INDEPENDENT SET: Find the maximum number K of independent vertices and their identity.

First, do a binary search on the graph to get the largest K which causes the INDEPENDENT SET finder to accept. If it doesn't accept for any K up to the number of vertices in G, then there is no independent set. Otherwise, we have found the largest K, and will now determine which vertices compose the independent set. To do so, we will iterate through the vertices of G, and remove them one by one, running the detector after each removal. If removing the vertex causes the INDEPENDENT

SET test to reject, then we add it back in and move on to the next vertex. This way, when we have iterated through all vertices in G (or accepted when there were only K vertices remaining), then we will have removed all non-critical vertices, and therefore all remaining vertices compose the independent set.

(c*) 3-COLORING: Find the color assignment. (Note that 3-COLORING does not accept a partially colored graph as an input; think of how you might impose restrictions on the coloring in a different way.)

First, run 3-COLORING on the graph to see if it accepts. If not, then there is no valid coloring, otherwise, we will find a valid coloring.

The first thing to note is that the 3-colors used are interchangeable. So the correct coloring could swap all vertices of one color for all vertices of another, and still be correct. This means we can randomly pick the first vertex to color, and it will be part of a correct color assignment.

To find the color assignment, we will add three new vertices to our graph, each connected to the others. These vertices will correspond to colors C_1, C_2, and C_3. It does't matter which corresponds to which color, for the reason mentioned above. These vertices are not connected to the other vertices in the graph.

Then, we iterate through each vertex in the graph. First, we try to connect the vertex to vertex C_1. Then we run the 3-COLORING detector, and if it succeeds, we assign the vertex color C_1. Otherwise, we try to connect the vertex to C_2, and do the check again. If it succeeds, then we assign the certex color C_2. If it fails on both of these, then we connect the vertex to C_3, and assign the vertex color C_3, since this is the only remaining possibility, and we know a valid 3-COLORING exists. Then, we repeat this process for each vertex, maintaining all edges to the 3 special vertices. In the end, we can remove the 3 special vertices, and we will have a valid 3-COLORING of the graph. This takes linear time in the number of vertices, since we just check each one once, so the process is polynomial.