CS 3510: Design & Analysis of Algorithms (Sections A & B)

Homework 7: NP-Completeness

Professors Dana Randall and Gerandy Brito **Due: 11/22/2022 11:59pm**

Name: Anthony Wong

<u>**The steps to write a reduction.**</u>

In an **NP** Reduction, we have a Problem B, and we want to show that it is **NP**-complete. These are the steps:

- Demonstrate that problem B is in the class of **NP** problems. This is typically a description of a procedure that verifies a candidate solution. It needs to address the runtime.
- Demonstrate that problem B is at least as hard as a problem believed to be **NP**-complete or **NP**-hard. This is done via reduction from a known problem A to the unknown problem B (A$\rightarrow$ B) as follows:
    1. Show how an instance of A is converted to an instance of B in polynomial time.
    2. Show how a solution to the instance of B can be converted to a solution for the initial instance of A, again in polynomial time.
    3. Furthermore, you must show that a solution for B exists if-and-only-if (IFF) a solution to A exists. This means that, after proving the previous step, you are left with checking that if there is no solution for B, then no solution exists for A. This may be done via the contra-positive approach, showing that if a solution exists for A then a solution for B must exist.

The second bullet point above is the proof that problem B is **NP**-hard which combined with the first bullet point yields the **NP**-complete proof.

You are allow to declare the following problems in the class **NP**-hard without further justification.

- SAT, 3-SAT, IS, Vertex Cover, Rudrata (or Hamiltonian) Cycle, Rudrata (or Hamiltonian) Path.

1.) (10 points each) Show that each of the following problems is in **NP**. This corresponds to completing the first bullet point of the outline on the first page. For each one, assume that S is a potential solution to the problem; you must show how you can verify that S is indeed a solution in polynomial time.

(a.) Given an array of n distinct integers, return those integers sorted in ascending order.

Solution:

If we first assume S is a potential solution to the given problem, then we know S is some array consisting of n distinct integers. If this array is sorted in ascending order, then if we iterate through the array, no next number should be less than a previous value. If we ever encounter a case where a number with a index is lower than than the previous index, then we know the potential solution S is not valid. If we go through the whole array and we never run into such a case, the solutions S is valid. Since we iterate through the whole array S of length n once, and all our comparisons are of constant time we can say this algorithm is $O(n)$ which is of polynomial time.

(b.) Given a directed acyclic graph, $G = (V, E)$ of n vertices, return a valid topological sorting of that graph.

Solution:

The main case we need to account for in this problem is that graph G could have many valid topological sortings. Do combat this, let us first assume the graph G comes with an adjacency list (or any other way you can find vertices that are neighbors of any vertex in the graph). Then, to verify sorting S is a valid solution, we start iterating from the beginning of the ordering to the end. Along the way, every vertex we pass through that is valid is popped into a visited set. We determine the validity of a vertex by checking that all adjacent vertices have not yet been visited (checking that none are in the visited set). If we ever encounter an adjacent vertex that has already been visited, sorting S is not a valid topological sorting. If we iterate through the whole sorting S and never encounter an invalid adjacent vertex, sorting S is a valid topological sorting. Because in the worst case we need to iterate through every vertex and use every edge to check validity of each vertex, our runtime is then $O(|V| + |E|)$ which is of polynomial time.

(c.) Given a family of sets $\{S_1, S_2, ..., S_n\}$ and a budget b, return a set H of size $\leq b$ which intersects every S_i.

Solution:

We first assume the potential solution set H (changing the variable name for clarity) is a set of size less than or equal to b. To verify that the set H intersects every S_i or every set in the family of sets, we will iterate through the family of sets and check if there is a value in set H that exists in set S_i. If there exists a value in H that also exists in S_i, we continue iterating onto the next set in the family. If we ever encounter the case where there is no value in H that exists in any S_i, we can conclusively say that H does not intersect every S_i, which means the potential solution set H is invalid. If we iterate through the family of sets and never encounter an invalid set, the set H is a valid solution. Because we are checking if each set in the family of sets has any of the values in H, we are checking if a set contains a value (constant time) a maximum of $|H|$ (size of set H) times per set in the family of sets. We can then define the runtime of this algorithm as $O(mn)$ where m is the size of solution set H and n is the size of the family of sets, which is polynomial time.

2.) (7.5 points each) We will prove that the following problem is **NP**-complete. Define the LENGTH-K RUDRATA PATH (we will call this LKP for short) problem as: given an undirected graph $G = (V, E)$ of n vertices and integer an k, is there a path in G that passes through exactly k vertices? Note that RUDRATA and HAMILTONIAN mean the same thing.

(a.) We first show that a solution to this problem can be verified in polynomial time. Let S be a sequence of vertices. Provide a polynomial time algorithm to check that this is a valid solution to the LKP problem.

Solution:

Let us first assume the graph G comes with an adjacency list (or any other way you can find vertices that are neighbors of any vertex in the graph). We can also instantiate a visited set for vertices we've seen. To verify some solution sequence S, we should iterate through the sequence, continually adding to the visited set as we go. At each vertex we iterate through, we first should verify that the previous and current vertex are in fact neighbors via something like an adjacency list. If they are not then the sequence S cannot be valid. Then, when we go to add our vertex to the visited set, we need to check if that vertex is already in the set. If it is then the sequence S cannot be valid as it violates the rule for a HAMILTONIAN PATH. If we iterate through the whole sequence and we never encounter the cases where a path between two vertices does not exist nor do we repeat a vertex, we can say the sequence S is valid. Because in the worst case we need to iterate through every vertex and use every edge in G to check validity of each vertex, our runtime is then $O(|V| + |E|)$ which is of polynomial time.

(b.) Show how to take an input I to the RUDRATA PATH problem and convert it to an input I' of LKP. Recall that LKP takes as input a graph G and an integer k, whereas RUDRATA PATH only takes as input a graph G.

Solution:

We can change the input I to the RUDRATA PATH problem and convert it to an input I' of LKP by taking the length of the vertex set V from the graph G and passing part of the input into LKP. More clearly, the input I' would be the graph G from RUDRATA PATH problem and the length of the vertex set V from the graph G as the integer k. Because we are only passing in a length and not doing any other computation, this conversion in input can be done in $O(1)$ time which is polynomial.

(c.) Show how S', a solution of LKP on input I', can be turned into S, a solution of RUDRATA PATH on input I.

Solution:

The solution S' of LKP with the input of graph G and the length of the vertex set V as integer k tells us whether there is a path in G such that we pass through $|V|$ (all) vertices exactly once. LKP with input I' is an equivalent problem to the RUDRATA PATH problem which solves whether there exists a path that visits every vertex exactly once. This means that a solution S', a solution of LKP on input I', is equivalent and can be turned into S, a solution of Rudrata Path on input I. Because our solution S' is the same as our solution S, this conversion is $O(1)$ which is polynomial.

(d.) Show that there exists a solution S' to LKP on input I' if-and-only-if there exists a solution S to Rudrata Path on input I.

Solution:

Above, we have shown that if there exists a solution S' to LKP on input I', there exists a solution S to Rudrata Path on input I. If there does not exist a solution S' to LKP on input I', then there does not exists a path in G that passes through $|V|$ (all) vertices exactly once. If this is the case, we can also say that there does not exist a solution S to Rudrata Path on input I, given that a solution requires there to exist a path that visits all vertices exactly once, which is equivalent to a path that passes through $|V|$ vertices. This means there exists a solution S to Rudrata Path if-and-only-if there exists a solution S' to LKP on input I' on input I.

3.) (35 points) The ALMOST-SAT problem takes as input a boolean formula on n literals, in conjunctive normal form with m clauses. The output is an assignment of the literals such that **exactly** $m-1$ clauses evaluate to `TRUE`, if such assignment exists, and the output is NO, otherwise. Show that ALMOST-SAT is **NP**-complete.

Solution:

First let us demonstrate that ALMOST-SAT is in the class of **NP** problems. To do this we need to be able to verify a candidate solution S in polynomial time. We can do this by evaluating each literal within each clause. For every clause, after iterating through the literals, we can determine if the clause is true or false. We should create a counter to keep track of the number of false clauses, such that we increment it every time we evaluate a false clause. Once we finish iterating through all the clauses, if the value of our false counter is greater than 1, the candidate solution S is not valid. If our the value of our false counter is equal to 1, the candidate solution S is valid. This can be done by evaluating n literals per m clauses, so the runtime is $O(mn)$ which is polynomial.

To prove that ALMOST-SAT is **NP**-complete, we will use SAT which is **NP**-hard. Next, let us convert the input I for SAT to I' for ALMOST-SAT in polynomial time. Lets say the input I for SAT is a boolean formula f that has n' literals in CNF with m' clauses. We can convert this into an input I' for ALMOST-SAT that is a boolean formula f' by introducing two literals y and $\bar{y}$, such that we add the boolean formula f from input I onto the new literals in the following fashion: $f \wedge y \wedge \bar{y}$. This results in a boolean formula f' that has n literals and m clauses where $m = m' + 2$ (m' from the number of literals in f from input I). This is a constant addition to f so the runtime of this conversion can be done in $O(1)$ time which is polynomial.

To show that a solution S' of ALMOST-SAT on input I' can be tuned into a solution S of SAT on input I, we can say that the solutions are equivalent. This means that the solution S' is equals to the solution S. We know this because the input I' of ALMOST-SAT is the boolean formula f from I with the addition of two literals y and $\bar{y}$ such that $f' = f \wedge y \wedge \bar{y}$. Because y can take on either true or false, one of the new literals is true and the other must be false. If the solution S' is true, it means that $m-1$ clauses in f' are true or 1 clause in f' is false. Given the nature of the two additional literals y and $\bar{y}$, the 1 false clause must necessarily come from one of them. This means that in this case, the boolean formula f from I must evaluate to true. Because the solution S for SAT evaluates to true if the boolean formula f from the input I is also true, we know that S and S' are equivalent when S' is true. This conversion is $O(1)$ which is polynomial.

Above, we have shown that if there exists a solution S' to ALMOST-SAT on input I', there exists a solution S of SAT on input I. If there does not exists a solution S' to Almost-SAT on input I', there are more than 1 false clauses in the boolean formula f'. We have stated above that only one false clause can ever come from y and $\bar{y}$ in the formula for f'. This means that the other false clause must come from within f in the formula for f'. This means that f from I evaluates to false. Because the boolean formula f from the input I evaluates to false, we know that the solution S must also not exist. This means there exists a solution S of SAT on input I if-and-only-if there exists a solution S' to Almost-SAT on input I'. This conversion is $O(1)$ which is polynomial.

Given our polynomial time verification algorithm, we can say Almost-SAT is in the class **NP**. Given the reduction we can say Almost-SAT is in the class **NP**-hard as well. Because Almost-SAT is in both these classes, we can say it is **NP**-complete.

4.) (5 points) Hooray! You made it to your last homework problem for CS 3510. Name your favorite algorithm you learned this semester. ☺

Solution:

Kruskal's algorithm