601.433/633 Introduction to Algorithms **Fall 2024**
Homework #9 **Due:** December 5, 2024, 9am

Bella Lu
Partners: Jecia Mao, Luna Liu

1 Graduation Requirements (33 points)

John Hopskins University[1] has n courses. In order to graduate, a student must satisfy several requirements of the form "you must take at least 1 course from subset S". Moreover, courses *are allowed* be used towards multiple requirement. For example, if there was a requirement that a student must take at least one course from $\{A, B, C\}$, and another required at least one course from $\{C, D, E\}$, and a third required at least one course from $\{A, F, G\}$, then a student would only have to take A and C to graduate.

Now consider an incoming freshman interested in finding the *minimum* number of courses required to graduate. Your job is to prove that the problem faced by this freshman is NP-complete. More formally, consider the following decision problem: given n items (say $a_1, \ldots a_n$), given m subsets of these items $S_1, S_2, \ldots, S_m$, and given an integer k, does there exist a set S of at most k items such that $|S \cap S_i| \geq 1$ for all $i \in \{1, \ldots, m\}$.

(a) (11 points) Prove that this problem is in NP.

In this problem, I is the collection of m subsets of the given n items, $S_1, S_2, \ldots, S_m$, and an integer k. The witness, which we can call X, is some subset of the given n items with size at most k.

Algorithm:

Go through each of the m subsets in I, and check if there is at least one element in x is present in each S.

i.e. Go through each element in S_1, and compare each element with each element in x. If an element match then the requirement S_1 is satisfied.

If every requirement has been satisfied, the algorithm will return true. If there is at least one that fails, the algorithm will return false.

Runtime:

The total input size, or in other words the total number of elements in I, is at most mn because there are m subsets, each having at most n terms. The runtime is $O(k(mn))$ because we have to compare each of the elements in I with every element in X, which has at most k elements, to verify if each requirement is satisfied. Thus the runtime is polynomial in the

[1] `https://www.youtube.com/watch?v=JEH2ha1p0WA`

size of the input, $O(mn)$.

Correctness:

If I is a yes-instance, there will be some solution to the problem, and we can let X be one of the solutions.

If I is a no-instance, there will be no solutions feasible. Because if there was, then it would be a solution and I would be a yes-instance instead. So the algorithm will return false for every witness.

Thus X is a valid witness.

(b) (22 points) Prove that this problem is NP-hard.

We can reduce Vertex Cover to the graduation requirement problem.

Given a generic vertex cover graph, we can let each vertex represent a class, and each edge to represent a requirement with two elements. This is a valid instance for the graduation requirement problem.

If I is a yes-instance in the vertex cover problem, by definition, there exists some subset of vertices where each edge has at least a vertex that is in the subset. If an edge is connected to a vertex, it is equivalent to that requirement being satisfied. Thus, for $k =$ size of vertex cover solution, there will exist some witness for which the algorithm for the graduation requirement problem will return true.

We can prove by contrapositive the other condition. If I, with given k and where each S_i has exactly two course requirements, is a yes-instance in the graduation requirement problem, then that means there exist at least one witness where each S_i has at least one class that is in the witness. When we translate this into a vertex cover problem, each edge (S_i) always connect two vertices, which represent two classes. If some witness from the graduation problem satisfies the requirements, that means that this witness will be a subset of vertices where every edge is covered. This is a valid vertex cover problem solution. Thus, there exists some witness which the graduation problem will return true that is a vertex cover solution.

This reduction is valid because there will exist a solution in this problem if and only if there is a solution in the Vertex Cover problem.

Converting each vertex to a class takes $O(n)$ and each edge into a requirement takes $O(m)$. So, this reduction takes polynomial time.

Thus, the fact that Vertex Cover is NP-hard means that this problem is also NP-hard.

2 Magic Subroutines (34 points)

(a) (17 points) Suppose you are given a magic black box that, given an arbitrary graph G, can determine (in polynomial time) the number of vertices in the largest clique in G. Describe a polynomial-time algorithm which uses this magic black box as a subroutine and that, given an arbitrary graph G, returns a clique of G of maximum size. Prove polynomial running time and correctness.

Algorithm:

Let S be an empty list. It will be our final output

Let G' be an empty graph. Set $G' = G$. It will be the graph we are manipulating.

G is the original graph in the question.

Run the given graph G through the magic black box to get the number of vertices in the largest clique.

Let n be the number of vertices G has. Let $v_1, v_2, \ldots, v_n$ be the vertices in G.

We can remove v_1 from G', then pass this new graph into the black box. If the number returned changes, then we know that this vertex, v_1, is in the largest clique in G. We can add v_1 to S, and put v_1 back into G'. If the number returned doesn't change, then we know that this vertex, v_1, is not in the largest clique in G. So we don't add v_1 to S, and we can remove v_1 from G'. We can do the same thing with v_2, and all other vertices.

S will be a list of all the vertices that are in the clique, then that list would be our final output.

Runtime:

We have to check with the magic black box at most number of vertices, n, times. Each checking is in polynomial time, so the overall runtime is still in polynomial time.

Proof of Correctness by Induction:

Let x be the number of vertices in the largest clique in G. We obtained this number by first running the original graph G through the black box.

Base Case: S has x elements, and is the vertex set for the largest clique in the original graph G. This fulfills the condition in the question.

For the sake of induction, let $k + 1$ to be the number of vertices currently in G', and the largest clique of G is still present in G'.

Now we prove that the largest clique is still present in G' when the number of vertices in G' has decreased to k. At this point, let v_i be the vertex we are removing. If we decided to get rid of this vertex, it means that, without the vertex, the number of vertices of the maximum clique in G' didn't change, so the maximum clique in G is still present in G'. This means that removing v_i doesn't affect the validity of G', which means that the clique is still in G'.

Since we were able to remove vertices that aren't in the clique from G' until only vertices that are in the clique remains, (because we know that the clique has x vertices, and if every vertex we remove, as proven, isn't in the clique) then the x vertices that remains must be the correct clique of maximum size. As per our algorithm, the way we added vertices into S ensures that it will contain exactly the vertices that remain in G'. Thus, our final output S will contain the vertex set of the clique of G of maximum size.

(b) (17 points) Suppose you are given a magic black box that, given an arbitrary boolean circuit Φ (with one output and no loops, like in Circuit-SAT), can determine in polynomial time whether Φ is satisfiable. Describe a polynomial-time algorithm that either computes a satisfying input for a given boolean circuit or correctly reports that no such input exists, using the magic black box as a subroutine. Prove polynomial running time and correctness.

Algorithm:

Let S be an empty list that can contain boolean values.

We can first run Φ through the black box with no inputs. If it returns no, that Φ is not satisfiable, then no inputs exist. This means that we can stop here and report that no such input exists.

If the black box returns yes, that Φ is satisfiable, and we can find a satisfying input. Let n be the number of slots in the input. We can set the first slot of the input of Φ as false, then run Φ through the black box again. If the black box still returns yes, then we keep the first slot of the input as false and add false to our solution S. If now the black box returns no, we flip the first slot to true and add true to our solution S. After this, the first slot will be set and we don't modify it anymore. Keeping the first slot, we can make the second slot false, then repeat the same process. We do this until all slots for the input as been filled, and we will get a list S feasible input, which will be our final output.

Runtime:

We have to check with the magic box n times, which is the number of slots in the input. Each checking is in polynomial time, so the overall runtime is still in polynomial time.

Proof of Correctness By Induction:

Base Case: If black box responded with no upon checking Φ, we can simply report no feasible solution. If black box responded with yes, S will be a list of boolean values that, as the input, will satisfy Φ.

Let S have boolean values $s_1, s_2, \ldots, s_n$. Let n be the total number of slots in the input for Φ.

For the sake of induction, let $k-1$ be the number of slots in the input we have verified so far. This means that $s_1, \ldots, s_{k-1}$ has been determined. We assume that with the first $k-1$

slots of the input filled, Φ is still satisfiable.

Now we prove that adding the next boolean input, s_k, so that the total number of slots in the input is k, keeps the satisfiability of Φ. Since we tested at the start and Φ, without any input, is satisfiable, there must exist some solution input. This means that s_k must be either true or false. As per our algorithm, we will test false first. If, with $s_k =$ false makes Φ still satisfiable by the black box, then it is correct and our condition that Φ is satisfiable holds. Otherwise, we know that s_k must be true, and again our condition holds.

Thus, we have proved that we will find a list S which is the input of Φ, making Φ satisfiable, should there be a solution present.

3 Integer Linear Programming (33 points)

In class we talked about linear programming, and the fact that it can be solved in polynomial time. Slightly more formally, we defined the feasibility version of linear programming to be the following decision problem

- Input: n variables $x_1, x_2, \ldots, x_n$, and m non-strict linear inequalities over the variables.
- Output: YES if there is a way of assigning each variable a value in $\mathbb{R}$ so that all m linear constraints are satisfied, NO otherwise.

Let Integer Linear Programming be the same problem, but where each variable is only allowed to take values in $\mathbb{Z}$ rather than in $\mathbb{R}$.

(a) (11 points) Prove that Integer Linear Programming is in NP.

In this problem, I is the m non-strict linear inequalities, and the witness, which we can call x, is some input with n variables $x_1, x_2, \ldots, x_n$, all assigned value.

Algorithm:

Plug the values of the input for each variable into each constraint, and check if it is satisfied.

If all of the inequalities are satisfied, then such x is feasible. If at least one inequality isn't satisfied, then such x is not feasible.

Runtime:

Assigning the value of each variable for each inequality takes $O(n)$ because there are at most n variables, and there are m inequalities, so the total runtime would be $O(mn)$. This is polynomial time in the size of the input, which is $O(m + n)$.

Correctness:

If I is a yes-instance, there will be some solution to the problem which will satisfy all the constraints. We can let X be one of the solutions.

If I is a no-instance, there will be no solutions feasible. Because if there was, then it would be a solution and I would be a yes-instance instead. So the algorithm will return false for every witness.

Thus X is a valid witness.

(b) (22 points) Prove that Integer Linear Programming is NP-hard.

We can reduce 3-SAT into an integer linear programming question.

Given a generic 3-Sat problem, an instance is 3CNF formula, where every clause has ≤ 3 literals. We can let each clause to be an inequality, where each inequality has ≤ 3 variables. Each input x_i is a boolean variable, so translated to an integer LP problem, each input variable x_i will only take the values 0 or 1, respectively false or true. Since each clause is connected by ANDs and inside each clause is connected by ORs, each clause can be translated into an inequality of a summation of the variables. For example, $(x_1 \cup \neg x_2 \cup x_3) \cap (\neg x_4 \cup x_5)$ will become constraints $x_1 + (1 - x_2) + x_3 \geq 1$ and $(1 - x_4) + x_5 \geq 1$. In general, each clause turns into an individual constraint. We add the variable x_i to the summation if it doesn't have a NOT gate it front of it, otherwise we add $(1 - x_i)$ to the summation. For every constraint, the final summation should be ≥ 1, to reflect the ORs. In the end, every constraint must be satisfied, reflecting the ANDs. Also, for all $1 \leq i \leq n$, $x_i \geq 0$ and $x_i \leq 1$, which, since it is an integer, that means it takes the value of 0 or 1.

If I is a yes instance in 3-SAT, then there exists a solution in the 3-SAT problem. This means that the Integer LP problem will also exist a solution because we can set each $x_i = 1$ if x_i was true in the 3-SAT problem, and $x_i = 0$ if x_i was false in the 3-SAT problem.

For the other direction, we can prove by contraposition. If I is a yes instance in the Integer LP problem, then there exist a solution in the Integer LP. This means that the 3-SAT problem will also exist a solution because we can set each x_i to be true if $x_i = 1$ in the Integer LP problem, and x_i to be false if $x_i = 0$ in the Integer LP problem.

Thus, this reduction is valid because there will exist a solution in this problem if and only if there is a solution in the 3-SAT problem.

Converting the $x_i \geq 0$ and $x_i \leq 1$ takes $O(n)$ because there are n variables, and the summations take $O(m)$ because there are m constraints. So, this reduction takes polynomial time.

Thus, the fact that 3-SAT is NP-hard means that this problem is also NP-hard.