Analysis of Algorithms Homework 6

USC ID: 3725520208
Name: Anne Sai Venkata Naga Saketh
Email: annes@usc.edu

1. **In Linear Programming, variables are allowed to be real numbers. Consider that you are restricting variables to be only integers, keeping everything else the same. This is called Integer Programming. Integer Programming is nothing but a Linear Programming with the added constraint that variables be integers. Prove that integer programming is NP-Hard by reduction from SAT.**

Solution:

Here, we need to prove that the Integer Linear programming is NP-Hard, by using a reduction from SAT problem.

To prove that Integer Linear Programming is NP-Hard, by using a reduction from SAT, we shall be constructing an ILP from any given instance of the SAT problem. i.e., any instance of the SAT problem can be reduced to an Integer Linear Programming.

Construction of ILP from SAT Problem:

Let's proceed with an SAT instance, that has an 'm' number of clauses. The following describes how we can create an instance of integer programming from the SAT problem:

1. Create a corresponding integer variable in the Integer Programming instance for each SAT variable. In this case, we can assume the variables to be $\mathbf{X_i}$.
2. To indicate true and false values for the SAT instance, accordingly, we will use the values '0' and '1', respectively, in the integer Linear programming instance.
3. Keep in mind that the values of the variable $\mathbf{X_i}$ can be either 0 or 1 can only. (as we are solving an Integer Linear Programming).
4. For each clause in the SAT instance, create a constraint in the instance of integer linear programming. This constraint will be composed of a linear combination of integer variables $\mathbf{X_i}$ that we have already defined in Steps 1 and 2, which correspond to the literals in each clause.
5. The Objective function of the above ILP shall be as follows:

$$Max(\sum_{i=1}^{n} Xi)$$

6. Now we must define the constraints corresponding to each of the 'm' clauses in the SAT, as follows:

   The general form of an SAT is as follows:

$(X_1 \vee X_2 \vee X_3) \wedge (X_1 \vee X_4 \vee X_5 \vee X_7 \vee X_8 \vee X_{11}) \wedge \ldots\ldots (X_3 \vee X_{n-1} \vee X_n)$

For the SAT to be true, every clause must result in a Boolean value of **'True',** which indeed indicates that at least one variable in each clause must have a Boolean value of **'True'.**

We have assigned a value of '1', if the Boolean value of the variable is 'True' and '0' if the Boolean value of the variable is 'False.'

From the above general form, the constraints in the ILP shall be as follows:

$X_1 + X_2 + X_3 >= 1,$

$X_1 + X_4 + X_5 + X_7 + X_8 + X_{11} >= 1$

……

$X_3 + X_{n-1} + X_n >= 1$

To convert the above constraints into the standard form of Linear programming, we shall need to multiply the inequalities with '-1'.

Claim:

Any instance of SAT is satisfiable if and only if the corresponding instance of the Integer Programming is solvable. i.e., the ILP is feasible.

Proving the Claim:

1. **Proof in the Forward Direction (=>):**

   Assume that the initial SAT instance is satisfiable. In that case, the assignment of truth values to the variables exists and satisfies each clause.

   By setting the respective integer variables in the satisfactory assignment to 1 for the variables with a Boolean value of 'true' and 0 for the variables with a Boolean value of 'false', we can provide a feasible solution to the Integer Programming instance.

   Since a satisfying assignment in the SAT instance satisfies all the clauses, the corresponding assignment in the Integer Programming instance also satisfies all the ILP constraints.

2. **Proof in the Forward Direction (<=):**

   Here, consider that an instance of integer programming is feasible.

Then, an assignment of integer values to the variables exists that satisfies every constraint in the ILP. By setting the appropriate variables to true for the integer variables with a value of 1, and false for the integer variables with a value of 0, we can create a satisfying assignment to the SAT instance.

The satisfied assignment fulfills all clauses since the matching solution to the integer programming instance satisfies all the required constraints of the ILP.

Conclusion:

We have therefore demonstrated the correctness of the reduction and the polynomial-time feasibility of reducing every SAT instance to an instance of integer programming.

Since we already know that SAT is an NP-Hard problem, the Integer Linear Programming problem, which has been reduced from the SAT, is at least NP-Hard.

2. **There are N cities, and there are some undirected roads connecting them, so they form an undirected graph G = (V, E). You want to know, given K and M, if there exists a subset of cities of size K, and the total number of roads between these cities is larger or equal to M. Prove that the problem is NP-Complete.**

   (a) Show this problem is in NP.
   (b) Show this problem is in NP-hard.

   Solution:

   To prove that the given problem is NP-Complete, we need to prove that it is NP as well as an NP-Hard problem.

   (a). Showing the given problem is NP.

   In NP, we need to show that for the given problem, if we are given a solution, then we can verify that the solution which is given is satisfying all the required constraints and is correct in Polynomial-time.

   For example, for the above problem, if we are given a solution with 'K' vertices (cities) and 'M' edges connecting the cities. Then we can easily verify the solution in polynomial time.

   i. We can verify that the solution contains all the 'K' cities by performing a graph traversal on the given solution graph, which can be completed in polynomial time.
   ii. The next thing we need to verify is if there are 'M' different edges. This can be verified by constructing an adjacency matrix of size 'K' and marking all the edges between the different cities. Traversing to find the number of roads(edges) between the 'K' cities can be done in $\boldsymbol{O(K^2)}$, Which is polynomial time.

So, by the above steps, we can say the given problem is in NP.

(b). Showing that the problem is NP-Hard.

To prove that the given problem is NP-hard, we shall need to reduce a known NP-Hard problem to our current problem in polynomial time. Here we shall be reducing the Vertex Cover problem to the given problem. We already know that the vertex Cover problem is an NP-Hard Problem.

Constructing the Vertex Cover problem:

Consider an undirected graph, G = (V, E), and a number K where |V|>=K (the number of vertices in the graph G we are constructing should be greater than or equal to K, as we need to find a vertex cover of size 'K' in the graph G), also where the number of edges E is greater than or equal to 'M', i.e., |E| >= M.

In Graph 'G' that we have constructed, find a vertex cover of size 'K'.

We have proved this for the number of vertices |V|>=K, but now we have to find a subset of vertices(cities) 'K' such that the number of edges connecting them shall be at least 'M'.

Claim:

The instance of the vertex cover problem has a vertex cover of size 'K' if and only if the instance of the given problem contains a subset of cities of size 'K' such that the number of roads connecting the 'K' cities them is at least 'M'.

Proving the Claim:

1. **Proof in the forward direction (=>):**

   Here, we shall consider that the initial vertex cover problem has a vertex cover of size 'K'. Then, to build a subset of cities in the new network, we can choose the K vertices in the vertex cover problem.

   Each road connecting the matching cities in the new graph is covered by at least one city in the chosen subset because every edge in the original graph is covered by at least one vertex in the vertex cover.

   Therefore, there are at least M total roadways connecting the chosen cities.

2. **Proof in the backward direction (<=):**

   Here we need to assume, that the given problem contains a subset of 'K' cities with a total number of roads between them that is at least 'M'. The original graph's 'K' corresponding vertices can then be chosen in order to create a vertex cover.

Each edge in the original graph is covered by at least one vertex in the chosen vertex cover since the total number of roads connecting the specified cities is at least 'M'.

Conclusion:

We now have successfully reduced the Vertex Cover problem into an instance of the given problem in polynomial time. We already know that the Vertex Cover problem is NP-Hard, so the given problem, which has been reduced from the vertex cover problem, is also as hard as an NP-Hard.

Therefore, we have proved that the given problem is both NP and NP-Hard, so we can say that the given problem is NP-Complete.

3. Consider a modified SAT' problem, SAT in which a CNF formula with m clauses and n variables x_1, x_2, . . . , x_n outputs YES if exactly m–2 clauses are satisfied, and NO otherwise. Prove that SAT' is NP-Complete.

(a) Show SAT' is in NP.
(b) Show SAT' is in NP-hard.

Solution:

To prove that the SAT problem is NP-Complete, we need to prove that the given problem is NP and also NP-Hard.

(a). Showing the given problem is NP:

In NP, we need to show that for the given problem, if we are given a solution, then we can verify that the solution which is given is satisfying all the required constraints and is correct in Polynomial-time.

Given that the SAT' is derived from a regular SAT Problem, to prove that SAT' is verifiable in polynomial time, assuming that we are given the truth assignment to the variables in the SAT'. We can solve the Boolean expressions in the clauses to find the Boolean value of each clause and then find the value of the total SAT' such that only '**m-2**' clauses are resulting in a '**True**' value and the other 2 clauses are resulting in a '**False**' value.

Since there are '**m**' clauses in the given SAT' problem, we can find if only '**m-2**' clauses are '**True**' in polynomial time. So, this problem comes under a subset of NP problems.

(b). Showing the given problem is NP-Hard:

To show that the SAT' is an NP-Hard problem, we need to reduce an already known NP-Hard problem to the SAT' in polynomial time. Here we consider the standard SAT problem and reduce it to the SAT'.

Construction of Standard SAT problem from SAT':

In the given SAT' problem, assume that we have a total of 'm' clauses, and to satisfy the SAT' we need to make sure that only '**m-2**' clauses are satisfiable, and the rest 2 clauses are not satisfiable.

Now, we need to construct the Standard SAT problem such that, the Boolean values of the rest 2 clauses from SAT' are negated so that the resulting Standard SAT problem with 'm' clauses will be satisfiable.

For example, consider the SAT' problem to be as follows,

$(X_1 \vee X_2 \vee X_3) \wedge (X_4 \vee X_5 \vee X_6 \vee X_7 \vee X_8 \vee X_9) \wedge \ldots\ldots(X_{n-5} \vee X_{n-4} \vee X_{n-3}) \wedge (X_{n-2} \vee X_{n-1} \vee X_n)$

Assume in the SAT' problem, all the clauses except $(X_{n-5} \vee X_{n-4} \vee X_{n-3})$ and $(X_{n-2} \vee X_{n-1} \vee X_n)$ are satisfied as per the constraint given in the question.

In the construction of the Standard SAT, we need to make sure that the Boolean values of clauses $(X_{n-5} \vee X_{n-4} \vee X_{n-3})$ and $(X_{n-2} \vee X_{n-1} \vee X_n)$ are negated(changed to '**True**' as they were '**False**').

Claim:

The instance of the standard SAT problem is satisfiable if and only if the corresponding instance of SAT' has exactly '**m-2**' clauses satisfied and the rest of the 2 clauses are not satisfied.

Proving the above claim:

1. **Proof in the forward direction (=>):**

   Here, we assume that the SAT' is satisfiable, i.e., exactly the 'm-2' clauses of the SAT' are satisfied, and the rest 2 clauses are NOT satisfied.

   Now, if the above statement is true, then the Standard SAT problem will also be satisfiable based on our construction as we have negated the values of two of the variables.

2. **Proof in the backward direction (<=):**

   Here, we assume that the Standard SAT is satisfiable, i.e., all the '**m**' clauses in the Standard SAT are satisfiable.

   If the above statement is True, then even the SAT' shall be satisfiable. i.e., 'm-2' clauses are satisfiable, and the rest 2 clauses are not satisfiable as we have negated the Boolean values of 2 variables in the construction.

Conclusion:

We now have successfully reduced the Standard SAT problem into an instance of the given SAT' problem in polynomial time.

We already know that the Standard SAT problem is NP-Hard, so the given problem, which has been reduced from the Standard SAT problem, is also as hard as an NP-Hard.

Therefore, we have proved that the given SAT' problem is both NP and NP-Hard, so we can say that the given problem is NP-Complete.

4. **Longest Path is the problem of deciding whether a graph G = (V, E) has a simple path of length greater or equal to a given number k. Prove that the Longest path Problem is NP-complete by reduction from the Hamiltonian Path problem.**

Solution:

To prove that the Longest Path problem is NP-Complete, we need to prove that this problem is both NP and NP-Hard.

(a). Showing the Longest Path problem is NP:

In NP, we need to show that for the given problem, if we are given a solution, then we can verify that the solution which is given is satisfying all the required constraints and is correct in Polynomial-time.

Assuming that we are given the value of 'k' and the longest path in the graph G(V,E). We can apply a graph traversal technique along the given path and find the length of the path such that no vertex is visited more than once, and then compare that length with the value '**k**'. Usually, this graph traversal can be done in polynomial time, i.e., $O(E + V)$.

(b). Showing the Longest Path problem is NP-Hard by reducing it from the Hamiltonian Path Problem:

Here, we are given that we need to reduce the Hamiltonian path problem to the Longest Path problem in polynomial time. We already know that the Hamiltonian Path problem is NP-Hard.

The Hamiltonian path can be defined as the length of the longest path, in graph G, such that no vertex is visited more than once and there exists no cycle.

Construction of Hamiltonian Path problem from the Longest Path Problem:

The length of the longest path in the given graph G=(V, E) is '**k**'.

Now, we need to create a graph G' with '**k+1**' vertices, in the next step, we need to add the edges between the vertices in graph G'.

The above graph shall be enough for the Hamiltonian path problem to be reduced as the Longest Path problem.

Claim:

Graph G contains a Hamiltonian Path, if and only if graph G has a Longest Path of length 'k'.

Proving the above claim:

1. **Proof in the forward direction (=>):**

   Here, we assume that the graph G=(V, E), where |V|=k vertices, contains the Longest path of length 'k'.

   Then graph G also contains a Hamiltonian Path, which shall be the same as the longest path covering all the vertices in graph G such that no vertex is visited more than once and there does not exist any cycle.

2. **Proof in the backward direction (<=):**

   Here, in this case, we assume that there is a Hamiltonian path in graph G, which is of the length 'k' such that no vertex is visited more than once and there does not exist any cycle.

   Then, in this case, the longest path in the graph G=(V, E) is of length '**k**', which is what is required as per the required question.

Conclusion:

We now have successfully reduced the Hamiltonian Path problem to the Longest Path problem in polynomial time.

We already know that the Hamiltonian Path problem is NP-Hard, so the given problem, which has been reduced from the Longest Path problem, is also as hard as an NP-Hard.

Therefore, we have proved that the Longest Path problem is both NP and NP-Hard, so we can say that the Longest Path problem is NP-Complete.

5. **Assume that you are given a polynomial time algorithm that decides if a directed graph contains a Hamiltonian cycle. Describe a polynomial time algorithm that given a directed graph that contains a Hamiltonian cycle, lists a sequence of vertices (in order) that form a Hamiltonian cycle.**

Solution:

Here, it is given that we are given a polynomial time algorithm that decides if the given graph, G, contains a Hamiltonian cycle or not.

A Hamiltonian cycle can be defined as the Hamiltonian path in which the starting and ending vertex in the Hamiltonian path are the same, thereby making it a cycle.

Algorithm to list the sequence of vertices of the Hamiltonian Cycle in-order for a Directed Graph:

Step 1: We need to run the given polynomial time algorithm to find if the given directed graph contains a Hamiltonian cycle. If the algorithm returns '**False**', then we can stop the execution and return saying that the given directed graph does not contain a Hamiltonian Cycle.

Step 2: If the above polynomial time algorithm returns '**True**', then we need to proceed to the next steps.

Step 3: Consider any arbitrary edge, say **E1** from the given directed graph, remove it from the given graph G to get a new graph G'. Now, run the given polynomial time algorithm to check if G' also contains a Hamiltonian cycle.

Step 4: If there does not exist a Hamiltonian cycle in G', then every subgraph of G contains the edge **E1,** so add it back to the graph G.

Step 5: Repeat the above steps until we are exactly left with V edges, because in each step we are left with a subgraph that contains a Hamiltonian cycle. At termination, we are left with edges that form the Hamiltonian cycle in the given graph G.

Step 6: Use a graph traversal algorithm, may be BFS to traverse the entire graph to find all the Vertices V in order that form a part of the Hamiltonian cycle.

6. **Consider the maximum cut problem in an undirected network G = (V,E). In this problem, we are trying to find a set of vertices C ⊂ V that maximizes the number of edges with one endpoint in C and the other endpoint in V \C (note that there is no constraint on any special nodes s and t unlike in the minimum cut problem). A simple greedy algorithm would be to start with an arbitrary set C and then ask if there is a vertex v ∈ V such that we will increase the number of edges in the cut by moving it from one side of the cut to the other. We then iterate this process until there is no such vertex.**

   a) Show that the algorithm must terminate, and

   b) show that when the algorithm stops, the size of the cut is at least half of the optimum.

   Solution:

   (a). Showing the algorithm will terminate:

   Yes, the above-mentioned greedy algorithm shall terminate, because we move at least one vertex from one side of the cut to the other side of the cut in each iteration, as mentioned, until there is no vertex left.

   As there are only a finite set of vertices in the given set C, the method will ultimately run out of vertices to move from one side of the cut to the other side of the cut.

(b). Showing that when the algorithm stops the size of the cut is at least half of the optimal solution:

Now, we must define the size of the cut for a given graph G = (V, E).

But prior to that, we need to define the following variables,

- Let C be the number of vertices selected by our algorithm as the maximum cut.
- Let OPT be the number of vertices selected by the algorithm as the maximum cut.
- Let E be the total number of edges in the given graph G.
- Let Xin be the number of edges for each vertex v, between itself and the vertices that are belonging to the same cut.
- Let Xout be the number of edges for each vertex v, between itself and the vertices that are belonging to the other cut.

Our goal is to demonstrate that E(C, V\C) >= OPT/2, where E(C, V/C) is the set of edges that cross the cut.

Step 1: Getting the inequality from the algorithm we have defined:

When our algorithm stops, then it means that no vertex needs to be moved from one cut to another so that the number of edges will increase, which means that Xout > Xin.

From the above assumptions that we have made,

C = 1 / 2 * Xout

E = 1 / 2 * (Xin + Xout)

→ E = C + 1 / 2 * Xin

We already know that Xout > Xin,

Therefore, C >= E / 2

→ 2 * C >= E

Step 2: Getting the inequality from the optimal algorithm:

From the above assumptions that we have made, we can derive that OPT <= E

This is because the size of even the maximum possible cut, cannot be more than the total number of edges in the graph.

Now, we need to combine both the in-equalities from Step 1 and Step 2, as shown below,

OPT <= E <= 2 * C

→ OPT <= 2 * C

→ OPT/2 <= C

Therefore, we have now proved that our algorithm terminates, and the size of the cut defined by our algorithm is at least half of what the optimal algorithm shall provide.

7. **It is well-known that planar graphs are 4-colorable. However, finding a vertex cover on planar graphs is NP-hard. Design an approximation algorithm to solve the vertex cover problem on planar graph. Prove your algorithm approximation ratio.**

Solution:

Planar graphs are like normal graphs, but the difference is that in the planar graph is that no two edges cross each other.

The Algorithm to solve the vertex cover problem on the planar graph is:

The algorithm to solve the vertex cover problem on a planar graph is as follows, assuming that "The minimum vertex cover is of size at least" is used:

Step 1: Coloring the Planar Graph G(V,E) vertices in 4 colors using the polynomial run-time approach.

Step 2: The vertices will be divided into the following four-color groups: C1, C2, C3, and C4.

Step 3: Now Let us assume that C1, is the most common color group.

Step 4: The objective is to create the vertex cover with vertices that don't belong to the most popular color family.

Step 5: Therefore, the vertex cover shall be represented as the set of C2, C3, and C4.

This process of classification into different color groups can be done in polynomial time, as we just need to iterate over all the vertices and add them to the different groups.

Now, the largest color group C1, shall have the size of at least V/4. So, the remaining vertices in C2, C3 and C4 combined shall be at most of 3V/4.

Because the lower bound as per the given question is V/2. The approximation ratio shall be 3V/4 / V/2 = **3/2 → 1.5**.

8. **Consider a well-known 0-1 Knapsack problem. Given collection of n items with integral sizes $w_1,...,w_n > 0$ and values $v1,...,v_n > 0$, and an integer knapsack capacity $W > 0$. The problem to find integers $w_1,...,w_n > 0$ such that the total value is maximized is known to be NP-Hard. As a possible heuristic, let's try the following greedy algorithm: sort items in non-increasing order of v_i/w_i and greedily pick items in that order. Show that this approximation algorithm is pretty bad, namely it does not provide a constant approximation.**

Solution:

It is given that we are given a list of items with weights $w_1,w_2,w_3,w_4……w_n$, such that every weight $w_i > 0$. Also, we are given the profits or the values for each of the items like $v_1,v_2,v_3,v_4……v_n$.

It is given that we need to sort the items in the non-increasing order of the ratios of $\mathbf{v_i/w_i}$.

Assuming the ratios of the values to weights are as follows, $r_1,r_2,r_3,r_4……r_n$, where $\mathbf{r_i = v_i/w_i}$

The capacity of the knapsack is given as 'W', and we need to choose the items in the non-increasing order of the r_i value.

For example, consider the following weights, values, and the knapsack capacity,

The knapsack is of capacity 'W' = 4

Item 1: weight $w_1 = 3$, value $v_1 = 10$, $r_1 = v_1/w_1 = 3.33$

Item 2: weight $w_2 = 2$, value $v_2 = 8$, $r_2 = v_2/w_2 = 4$

Then as per the given question, we must sort the items by $\mathbf{r_i = v_i/w_i}$ in non-increasing order, then we shall have the following order:

Item 2: $r_2 = v_2/w_2 = 4$

Item 1: $r_1 = v_1/w_1 = 3.33$

As given, if we proceed with using the greedy algorithm then the algorithm shall **first pick Item 2**, since it has the highest v_i/w_i ratio. However, since its weight is 2, we cannot include any other item in the knapsack, and the total value obtained would be only **8**.

But, in the optimal case, if we pick Item 1 instead, we shall fill the knapsack in a better way, thereby getting a total value/profit of **10**.

Therefore, the above-mentioned greedy approach fails to be a decent approximation to the Knapsack problem, i.e., it doesn't provide a good constant approximation because any slight variation in the given input may cause the output to deviate drastically from the optimal solution.