CSCE 629-601 Analysis of Algorithms

Fall 2022

Instructor: Dr. Jianer Chen
Office: PETR 428
Phone: (979) 845-4259
Email: chen@cse.tamu.edu
Office Hours: MWF 3:50 pm - 5:00 pm

Teaching Assistant: Vaibhav Bajaj
Office: EABC 107B
Phone: (979) 739-2707
Email: vaibhavbajaj@tamu.edu
Office Hours: T,TR 2:00 pm - 3:00 pm

Solutions to Assignment # 6

1. A *vertex cover* in an undirected graph G is a set C of vertices in G such that every edge in G has at least one end in C. Consider the following two versions of the Vertex-Cover problem:

> VC-D: Given a graph G and an integer k, decide whether G contains a vertex cover of at most k vertices.
>
> VC-O: Given a graph G, construct a minimum vertex cover for G

Prove: VC-D is in $\mathcal{P}$ if and only if VC-O is in $\mathcal{P}$.

Solutions. Suppose that the VC-O problem is in $\mathcal{P}$, i.e., is solvable in polynomial time. Thus, there is a polynomial time algorithm $\mathcal{A}_O$ that solves the VC-O problem. We construct a polynomial time algorithm $\mathcal{A}_D$ for the VC-D problem as follows: given an instance (G, k) of the VC-D problem, run $\mathcal{A}_O$ on G to obtain an minimum vertex cover C in G, then return "yes" if and only if $|C| \leq k$. The algorithm $\mathcal{A}_D$ is correct because the graph G contains a vertex cover of at most k vertices if and only if the minimum vertex cover of G contains no more than k vertices. Since the algorithm $\mathcal{A}_O$ runs in polynomial time, the algorithm $\mathcal{A}_D$ runs in polynomial time. Thus, the problem VC-D is solvable in polynomial time, i.e., it is in $\mathcal{P}$.

Conversely, suppose that the VC-D problem is in $\mathcal{P}$ so there is a polynomial time algorithm $\mathcal{A}'_D$ for the VC-D problem. We can construct a polynomial time algorithm $\mathcal{A}'_O$ for the VC-O problem, which is given in Figure 1

By our assumption, the algorithm $\mathcal{A}'_D$ runs in polynomial time. Because the algorithm $\mathcal{A}'_O$ calls the algorithm $\mathcal{A}'_D$ at most $n + n^2$ times (at most n times in step 1 and at most n times in step 3.1.1 for each i), the algorithm $\mathcal{A}'_O$ also runs in polynomial time.

To see the correctness of the algorithm $\mathcal{A}'_O$, first note that step 1 find the smallest integer i_0 such that $\mathcal{A}'_D(G, i_0) =$ yes. Thus, after step 1, $i = i_0$ is size of a minimum vertex cover of the graph G. In general, for each execution of step 3.1, we know that the minimum vertex cover is of size i, and we are looking for a vertex cover of i vertices in the graph G. In particular, if $\mathcal{A}'_D(G - v, i - 1) =$ yes, then the graph $G - v$ has a vertex cover of size $i - 1$, which, plus the vertex v, gives a minimum vertex cover of i vertices in the graph G. This means that the vertex v is in a minimum vertex cover of the graph G. As a result, if this is true at step 3.1.1, then we can include v in the minimum vertex cover C (at step 3.1.2), then recursively look for a (minimum) vertex cover of $i - 1$ vertices in the graph $G - v$. Step 3.1.2 of the algorithm thus correctly updates the graph G and the value i. In particular, at end of step 3, the set C contains

Algorithm $\mathcal{A}'_O$ Input: a graph G Output: a minimum vertex cover C in G 1. **for** $(i = 0;\ i \le n;\ i{+}{+})$ **do if** $(\mathcal{A}'_D(G, i) = \text{yes})$ break; 2. $C = \emptyset$; 3. **while** $(i > 0)$ 3.1 **for** (each vertex v in G) **do** 3.1.1 **if** $(\mathcal{A}'_D(G - v, i - 1) = \text{yes})$ 3.1.2 $C = C \cup \{v\}$; $G = G - v$; $i = i - 1$; 3.1.3 break; \\ break the **for**-loop 4. return(C).

Figure 1: The algorithm $\mathcal{A}'_O$ for VC-O

a vertex cover of i_0 vertices for the graph G, where i_0 is the value i after step 1, i.e., i_0 is the size of a minimum vertex cover of the input graph G. In conclusion, the algorithm $\mathcal{A}'_O$ runs in polynomial time and returns a minimum vertex cover C of the input graph G (at step 4). In other words, the VC-O problem is solvable in polynomial time, i.e., it is in $\mathcal{P}$.

This completes the proof of the question. □

2. Using the fact that the INDEPENDENT SET problem is $\mathcal{NP}$-complete, prove that the following problem is $\mathcal{NP}$-complete:

> CLIQUE: Given a graph G and an integer k, is there a set C of k vertices in G such that for every pair v and w of vertices in C, v and w are adjacent in G?

Solutions. We first show that CLIQUE is in $\mathcal{NP}$. An instance of the CLIQUE problem takes the form (G, k), where G is a graph and k is an integer. Consider the algorithm given in Figure 2.

For a Yes-instance $x = (G, k)$ of CLIQUE, i.e., if there is a set C of k vertices in G in which every pair of vertices are adjacent, then when the string y is this set C, the algorithm **VerifyClique**(x, y) will pass the tests in steps 1-2, and return in step 3 with an answer "Yes." On the other hand, if $x = (G, k)$ is a No-instance of CLIQUE, i.e., if there is no set of k vertices in which all vertices are pairwise adjacent, then the algorithm **VerifyClique**(x, y) for any y will either return in step 1 with a "No" answer (if y does not encode a set of k vertices in the graph G) or return in step 2 with a "No" answer (since the graph G does not have k pairwise adjacent vetices). That is, on a No-instance x of CLIQUE, the algorithm **VerifyClique**(x, y) returns "No" for all y. Finally, the algorithm **VerifyClique**(x, y) obviously runs in time $O(|x|^2)$ (assuming the graph G is given in an adjacency matrix). By the definition of problems in $\mathcal{NP}$, the algorithm **VerifyClique** shows that the problem CLIQUE is in $\mathcal{NP}$.

Now, we show that the INDEPENDENT SET problem is polynomial-time reducible to the CLIQUE problem, which will give the $\mathcal{NP}$-hardness of the CLIQUE problem. This reduction uses the notion of "complement graphs". Given an undirected graph $G = (V, E)$, the *complement graph* $\overline{G}$ of G is defined as $\overline{G} = (V, \overline{E})$, which has same set V of vertices, and the edge set $\overline{E}$ is defined as $\overline{E} = \{[u, v] \mid u, v \in V, u \neq v, \text{ and } [u, v] \notin E\}$.

The reduction algorithm takes as input an instance (G, k) of INDEPENDENT SET, constructs the complement graph $\overline{G}$ of G, then outputs $(\overline{G}, k)$ as an instance of CLIQUE. The algorithm

> **Algorithm. VerifyClique**$(x = (G, k), y)$
> Input: $x = (G, k)$, where G is a graph and k is an integer k, and a string y
> Output: verify if y is a solution to the Clique instance $x = (G, k)$
> 1. **if** (y is not a set of k vertices in G) return("No");
> 2. **if** (any two vertices in y are not adjacent) return("No");
> 3. return("Yes").

Figure 2: Clique is in $\mathcal{NP}$

obviously runs in polynomial-time (more precisely, in time $O(n^2)$). Moreover, it is easy to see that the graph G has a set C of k vertices in which no two are adjacent if and only if the same set C of k vertices in the complement graph $\overline{G}$ has all the k vertices pairwise adjacent. That is, (G, k) is a Yes-instance of Independent Set if and only if $(\overline{G}, k)$ is a Yes-instance of Clique. Thus, Independent Set $\leq_m^p$ Clique. Since the Independent Set problem is $\mathcal{NP}$-complete (thus, is $\mathcal{NP}$-hard), this reduction shows that the Clique problem is $\mathcal{NP}$-hard.

Summarizing the discussions, we conclude that the Clique problem is $\mathcal{NP}$-complete. □

3. Using the fact that the Partition problem is $\mathcal{NP}$-complete, prove that the following problem is $\mathcal{NP}$-complete:

> Knapsack: given n items of sizes $s_1, s_2, \ldots, s_n$ and values $v_1, v_2, \ldots, v_n$, resectively, a knapsack of size S, and a value objective V, can we select some of these items to fit into the knapsack so that the total value of the selected items is at least V?

Solutions. We first show that Knapsack is in $\mathcal{NP}$. An instance of the Knapsack problem is a tuple of $2n + 2$ integers $x = (s_1, v_1, s_2, v_2, \ldots, s_n, v_n; S; V)$. Consider the algorithm in Figure 3.

For a Yes-instance $x = (s_1, v_1, \ldots, s_n, v_n; S; V)$ of Knapsack, i.e., if there is a set A of items (note that the items are given by the integers $\{1, 2, \ldots, n\}$. Thus, a set A of items is a subset of $\{1, 2, \ldots, n\}$), which can fit into the knapsack (i.e., $\sum_{i \in A} s_i \leq S$) with the value at least V (i.e., $\sum_{i \in A} v_i \geq V$), then when the string y is this item set A, the algorithm **VerifyKnapsack**(x, y) will pass the tests in steps 1-2, and return in step 3 with an answer "Yes." On the other hand, if $x = (s_1, v_1, \ldots, s_n, v_n; S; V)$ is a No-instance of Knapsack, i.e., if there is no item set A that satisfies both $\sum_{i \in A} s_i \leq S$ and $\sum_{i \in A} v_i \geq V$, then the algorithm **VerifyKnapsack**(x, y) for any y will either return in step 1 with a "No" answer (if y does not encode a set of items) or return in step 2 with a "No" answer (since there is no item set A that satisfies both $\sum_{i \in A} s_i \leq S$ and $\sum_{i \in A} v_i \geq V$). That is, on a No-instance x of Knapsack, the algorithm **VerifyKnapsack**(x, y) returns "No" for all y. Finally, the algorithm **VerifyKnapsack**(x, y) obviously runs in time $O(|x|)$ (for first checking if y is a subset of $\{1, 2, \ldots, n\}$, then verifying the conditions for the size sum and value sum). By the definition of problems in $\mathcal{NP}$, the algorithm **VerifyKnapsack** shows that the problem Knapsack is in $\mathcal{NP}$.

Now, we show that the Partition problem is polynomial-time reducible to the Knapsack problem, which will give the $\mathcal{NP}$-hardness of Knapsack. The reduction algorithm takes as input an instance $x = (a_1, a_2, \ldots, a_n)$ of Partition, and outputs a tuple $y = (s_1, v_1, \ldots, s_n, v_n; S; V)$ as an instance of Knapsack, where $s_i = v_i = a_i$ for $1 \leq i \leq n$, and $S = V = (\sum_{i=1}^n a_i)/2$. The algorithm obviously runs in polynomial-time (more precisely, in time $O(n)$).

Algorithm. VerifyKnapsack$(x = (s_1, v_1, \ldots, s_n, v_n; S; V), y)$
Input: a KNAPSACK instance $x = (s_1, v_1, \ldots, s_n, v_n; S; V)$, and a string y
Output: verify if y is a solution to the KNAPSACK instance x

1. **if** (y is not a subset of $\{1, 2, \ldots, n\}$) return("No");
2. **if** ($\sum_{i \in y} s_i > S$ or $\sum_{i \in y} v_i < V$) return("No");
3. return("Yes").

Figure 3: KNAPSACK is in $\mathcal{NP}$

We show that x is a Yes-instance of PARTITION if and only if y is a Yes-instance of KNAPSACK.

Suppose that $x = (a_1, a_2, \ldots, a_n)$ is a Yes-instance of PARTITION. Then, we can divide the set $\{1, 2, \ldots, n\}$ into two disjoint subsets L and R such that $\sum_{l \in L} a_l = \sum_{r \in R} a_r$. In this case, we must have $\sum_{l \in L} a_l = \sum_{r \in R} a_r = (\sum_{i=1}^n a_i)/2$. Now consider the item subset L (again a subset of items is given by a subset of $\{1, 2, \ldots, n\}$). Then we have $\sum_{l \in L} s_l = \sum_{l \in L} a_l = (\sum_{i=1}^n a_i)/2 = S$ and $\sum_{l \in L} v_l = \sum_{l \in L} a_l = (\sum_{i=1}^n a_i)/2 = V$. Therefor, L is a item set that satisfies both $\sum_{l \in L} s_l \leq S$ and $\sum_{l \in L} v_l \geq V$, i.e., the instance $y = (s_1, v_1, \ldots, s_n, v_n; S; V)$ is a Yes-instance of the KNAPSACK problem.

On the other hand, if $y = (s_1, v_1, \ldots, s_n, v_n; S; V)$ is a Yes-instance of KNAPSACK, i.e., if there is an item set L that satisfies both $\sum_{t \in L} s_t \leq S$ and $\sum_{t \in L} v_t \geq V$, then since $s_t = v_t = a_t$ for all t and $S = V = (\sum_{i=1}^n a_i)/2$, we must have

$$(\sum_{i=1}^{n} a_i)/2 = V \leq \sum_{l \in L} v_l = \sum_{l \in L} a_l = \sum_{l \in L} s_l \leq S = (\sum_{i=1}^{n} a_i)/2.$$

Thus, the subset L of $\{1, 2, \ldots, n\}$ satisfies $\sum_{l \in L} a_l = (\sum_{i=1}^n a_i)/2$. If we let $R = \{1, 2, \ldots, n\} - L$, then obviously we also have $\sum_{r \in R} a_r = \sum_{i=1}^n a_i - \sum_{l \in L} a_l = (\sum_{i=1}^n a_i)/2$. That is, the set of integers $\{a_1, a_2, \ldots, a_n\}$ can be divided into two subsets L and R that have the same sum. This shows that $x = (a_1, a_2, \ldots, a_n)$ is a Yes-instance of PARTITION.

This completes the proof that PARTITION is polynomial-time reducible to KNAPSACK. Since PARTITION is $\mathcal{NP}$-complete (thus, is $\mathcal{NP}$-hard), the problem KNAPSACK is $\mathcal{NP}$-hard.

Summarizing the discussions, we conclude that the KNAPSACK problem is $\mathcal{NP}$-complete. □