CMSC 27230: Problem Set 7

Owen Fahey

March 3, 2023

Problem 1

1. Bundles is in NP

Here is a simple non-deterministic polynomial time algorithm:

1. For each $i \in [n]$, assign p_i to a random $j \in [m]$.
2. Once all paintings have been assigned, initialize a counter $q = 0$. For each $j \in [m]$ where $S_j \subseteq A_j$ (where is A_j the set of paintings assigned to j), increase the counter by 1.
3. IF $q \geq k$, RETURN YES
4. ELSE, return NO

To see that this is a non-deterministic polynomial time algorithm for this problem, if it is possible for k customers to buy all paintings in their bundle, then there is some probability that the algorithm will return YES. Otherwise, the algorithm will always return NO.

This algorithm runs in polynomial time. First, we randomly generate a number, $r \in [m]$, n times. (If we assume the arithmetic operations are constant time, random number generation takes constant time. Otherwise, it takes time proportional to the bits of r which is still polynomial in m). We can check if a set A is a subset of another set B in $O(|A| \cdot |B|)$ time (naive approach). At most, each subset could be size n. Hence, a very loose upper bound on the subset checking part of the algorithm is $O(m \cdot n^2)$. Hence, this algorithm takes $O(m \cdot n^2)$.

2. Bundles is in NP–Hard

Recall that the independent set problem is NP-Hard and asks the following: Given a graph G, is there an independent set of size k in G? In other words, is there a set of vertices $I \subseteq V(G)$ such that $|I| = k$ and no two vertices in I are adjacent to each other?

We can reduce an instance of IndependentSet to an instance of Bundles in polynomial time. If Bundles was easy, then to solve IndependentSet, we could just transform it into bundles and solve it. Hence, it follows that Bundles is at least as hard as IndependentSet, i.e, Bundles is NP-hard.

Reduction:

Let (G, k) be an instance of IndependentSet and $T(G, k)$ be an instance of IndependentSet that has been transformed into Bundles. Let C be a set of customer in Bundles where $C = \{c_1, ..., c_m\}$. Let I be a solution to (G, k) where $I \subseteq V(G)$ such that $|I| = k$ and no two vertices in I are adjacent to each other. Let J be a solution to $T(G, k)$ where J is a set of customers such that $|J| = k$ and for all distinct $i, j \in J, S_i \cap S_j = \varnothing$ (where $S_i, S_j \subseteq [m]$ represent customer c_i's and c_j's respective bundles). Let $E(v_i, G)$ take a vertex v_i and a graph G and return all edges in G that connect to v_i.

In order to transform (G, k) into $T(G, k)$, do the following:

1. For each $v_i \in V(G)$, assign v_i to be c_i
2. For each $e_i \in E(G)$, assign e_i to be p_i
3. Initialize sets $S_1, ..., S_m$. Each S_j will be a set of paintings representing c_j's bundle. For each $e_i \in E(G)$ with vertices (v_j, v_k), add p_i to S_j and S_k.

In words, each vertex gets represented as a customer and each edge gets represented as a painting. Each vertex's bundle consists of all the edges that are connected to it.

Mappings:

1. Each v_i corresponds to c_i for $i \in [|V(G)|] = [m]$.
2. Each edge e_j corresponds to p_j for $j \in [|E(G)|] = [n]$.
3. For each $v_i \in V|G|$, each $e_j \in E(v_i, G)$ corresponds to each $p_j \in S_i$.

Runtime:

1. $|V(G)|$ steps to assign each v_i to c_i
2. $|E(G)|$ steps to assign each e_i to p_i
3. $2|E(G)|$ steps to assign each edge to the bundles of both its vertices

Hence, this reduction can be performed in $O(V|G| + E|G|)$ time. Since, each $v_i \in I$ corresponds to $c_i \in J$. If we have solution J for $T(G, k)$ then we can easily find a solution I for (G, k)

Proof of Validity

Claim: If I is a solution to (G, k), then J is a solution to $T(G, k)$.

Proof: If I is a solution to (G, k), then $|I| = |J| = k$. Furthermore, for each distinct $v_i, v_j \in I$, $E(v_i, G) \cap E(v_j, G) = \varnothing$. Based on the mappings we have above, the following must also be true: for each distinct $c_i, c_j \in J$, $S_i \cap S_j = \varnothing$. Hence, the criteria that satisfy $T(G, k)$ are met. □

Claim: If J is a solution to $T(G, k)$, then I is a solution to (G, k).

Proof: If J is a solution to $T(G, k)$, then $|J| = |I| = k$. For each distinct $c_i, c_j \in J$, $S_i \cap S_j = \varnothing$. Based on the mappings we have above, the following must also be true: $v_i, v_j \in I$, $E(v_i, G) \cap E(v_j, G) = \varnothing$. Since this fact means that no two vertices in I connect to the same edge, this means that criteria for (G, k) is satisfied by I.

Claim: I is a solution to (G, k) iff J is solution to $T(G, k)$.

Proof: Let P mean "I is a solution to (G, k)" and let Q mean "J is a solution to $T(G, k)$". By our previous to claims, we have $P \implies Q$ and $Q \implies P$, meaning we have $P \iff Q$. □

$\therefore$ Bundles is at least as hard as IndependentSet meaning Bundles is NP-hard. □

3. Bundles is NP-Complete because it is in NP and NP-hard

Problem 2

1. TechTree is in NP

Here is a non-deterministic polynomial time algorithm:

1. Initialize an empty set T
2. For each $i \in [2, n]$, choose at random whether to add technology i to T (i.e. whether to research i)
3. IF $|S| > m$, return NO
4. IF S does not contain all desired technologies, return NO
5. Otherwise, return YES

To see that this is a non-deterministic polynomial time algorithm for this problem, if it is possible for all technologies in T to be researched while only researching at most m technologies, then it is possible that the algorithm returns YES. Otherwise, the algorithm always return NO.

This algorithm runs in polynomial time. Choosing to research or not technology can be done in constant time via random number generation. We perform this step $n - 1$ times $\rightarrow O(n)$. The time complexity of (3) is $O(1)$. An upper bound on time complexity of (4) is $O(|S| \cdot |T|)$ if we do naively search through S for every element element in T. Both $|S|$ and $|T|$ are bounded by n. Hence, an upper bound on (4) is $O(n^2)$. Thus, the algorithms takes $O(n^2)$.

2. TechTree is NP-hard

Recall that the NP-hard version of Vertex Cover we did in class asked: Given a graph G, is there a vertex cover of size k in G? In other words, is there a set of vertices $V \subseteq V(G)$ such that for every edge $(u, v) \in E(G)$, either $u \in V$ or $v \in V$?

We can reduce an instance of VertexCover to an instance of TechTree in polynomial time. If TechTree was easy, then to solve VertexCover, we could just transform it into TechTree and solve it. Hence, it follows that TechTree is at least as hard as VertexCover, i.e., TechTree is NP-hard.

<u>Reduction</u>

For convenience, I am going to rename the starting technology, v_1 to v_0. Here are the rules from TechTree that get updated:

1. The definition of T updates to $T \subseteq [n]$
2. The definition of R updates to $R \subseteq [n]$
3. Condition 1 updates to:

   > For every $i \in R$, there is a path P from v_0 to v_i such that for every vertex v_j in P, $j \in R \cup 0$ (Given the starting technology v_0, you can iteratively research all of the technologies $\{v_i : i \in R\}$ without needing any other technologies).

Let (G, k) be an instance of VertexCover where G is the graph and k is the target size of the vertex cover. Let $T(G, k)$ be an instance of VertexCover that has been transformed into an instance of

TechTree. Let g be $|V(G)|$ given a (G, k). Let $S \subseteq [g]$ represent a solution to (G, k) where $|S| = k$ and for all $i \in S$, $v_i \in V$. (In words, let (G, k) take S, the set of indices of all vertices in V, as a solution).

In order to transform (G, k) into $T(G, k)$ we do the following:

1. Create a directed graph G' with the same vertices as G, $\{v_1, ..., v_g\}$.
2. Add in vertex v_0 and for $i \in [g]$, add in a directed edge from v_0 to v_i
3. For each edge e_j in G with vertices (u_j, v_j):
    i. Add vertex v_{g-1+j}
    ii. Add in an directed edge from u_j to v_{g-1+j}
    iii. Add in an directed edge from v_j to v_{g+j}
4. Set the target technologies T to be the set of all added vertices in G'
5. Let $m = k + |E(G)|$

Mapping:

1. Each vertex $v_i \in V(G)$ directly corresponds to $v_i \in \{v_1, ..., v_g\}$
2. Each edge $e_j \in E(G)$ corresponds to $v_j \in \{v_{g+1}, ..., v_{g+j}\}$
3. $T = \{v_{g+1}, ..., v_{g+j}\}$
4. If R solves $T(G, k)$ then we can arrive at an S which solves (G, k) by removing all indices in R of vertices that corresponds to edges in G. More specifically $S = R \backslash \{v_{g+1}, ..., v_{g+|E(G)|}\}$.

Runtime:

$|V(G')| = |v_0| + |V(G)| + |E(G)| = 1 + |V(G)| + |E(G)|$

$|E(G')| = |v_0| \cdot |V(G)| + 2 \cdot |E(G)| = |V(G)| + 2|E(G)|$

$|E(G')| + |V(G')| = 1 + 2|V(G)| + 3|E(G)|$

We can construct G' in $O(|V(G')|+|E(G')|) = O(1+2|V(G)|+3|E(G)|) \rightarrow O(|V(G)|+|E(G)|)$

Initializing T requires adding $|E(G)|$ values to a list. Initializing m requires summing k and $|E(G)|$ which is polynomial in k and $|E(G)|$.

Hence, this reduction can be performed in polynomial time.

Proof of Validity

Claim: If S is a solution to (G, k), then R is a solution to $T(G, k)$.

Proof: In order to be solve $T(G, k)$, we need to research every edge technology represented by $v_{g+1}, ..., v_{g+|E(G)|}$. There are $|E(G)|$ edges technologies. Since, we can only research a total of $m = k + |E(G)|$ technologies, we can only research $m - |E(G)| = k$, non-edge technologies (i.e. vertex technologies). To reach any technology corresponding to an edge, we first must research

a technology corresponding to one of the vertices of that edge. Every vertex technology can be researched immediately. Hence, we are constrained as follows:

We can only research k vertex technologies.

We must be able reach every edge technology from the vertex technologies we research.

Observe that these are the exact same constraints that (G, k) satisfy. To restate this, in order to solve $T(G, k)$, it must be the case that we can research k vertex technologies such the set of corresponding vertices in G connect to all edges in G. Since these constraints are the same as an instance of VertexCover, we can conclude that if S solves (G, k) then R solves $T(G, k)$. □

Claim: If R is a solution to $T(G, k)$, then S is a solution to (G, k)

Proof: If R solves $T(G, k)$ then we can reach every edge technology from k or less vertex technologies. This entails that k or less vertices in G connect to every edge in G. Hence, this set of vertices that correspond solve this instance of vertex cover. Hence, S is a solution to (G, k). □

Claim: S is a solution to (G, k) iff R is a solution to $T(G, k)$

Proof: Let P mean "S is a solution to (G, k)" and let Q mean "R is a solution to $T(G, k)$". By our previous to claims, we have $P \implies Q$ and $Q \implies P$, meaning we have $P \iff Q$. □

$\therefore$ We have shown TechTree is at least as hard as VertexCover meaning TechTree is NP-hard. □

3. TechTree is NP-complete since it both in NP and NP-hard

Problem 3

Section (a)

This problem can be solved using dynamic programming.

Setup: Let $dp[i][k][q]$ take $i \in [0, n]$, $k \in [0, a_{start} + \sum_{i \in [n]} a_i]$ and $q \in [0, b_{start} + \sum_{i \in [n]} b_i]$. If it is possible to have k amount of currency A and q amount of currency B after trade j, $dp[j][k][q]$ returns 1; otherwise, $dp[i][k][q]$ returns 0. Our subproblems will be $dp[i][k][q]$ for all values of j, k and q. As an initialization step, set $dp[0][a_{start}][b_{start}]$ to 1 and all other $dp[0][k][q]$ to 0.

Recurrence relation and solution:

$$dp[i][k][q] = \max\{dp[i-1][k][q], dp[i-1][k-a_i][q-b_i]\}$$

Note: If we try to access an out of bounds subproblem (i.e. where $k < 0$ or $q < 0$) return 0.

Solve the subproblems in ascending order of $i \in [n]$

For each subproblem, if $dp[i][k][q] = 1$, record the last accepted trade. (To be specific, record trade i, if the subproblem $dp[i][k][q] = 1$ and is solved by taking $dp[i-1][k-a_i][q-b_i]$ as the max). We can use information about last trades to recreate how we ended up with any value of k and q.

Check all subproblems $d[n][\alpha][\beta]$ where $\alpha \geq a_{target}$ and $\beta \geq b_{target}$. If any of these subproblems equal 1, then it is possible to end with at least both a_{target} of A and b_{target} of B and we can trace back the proper sequence by using the record of last trades.

Discussion of Correctness:

This algorithm is correct because it checks all possible amounts of currency A and B one could possibly have after every trade.

An important observation is that if it is possible to have some amount of currency A and currency B after trade i then it will also be possible to have the same currency amounts after any trade j where $j > i$ (since we can maintain any amount of currency by declining trades). This is represented in the first term in the max function in the recurrence relationship.

Recall that (a_i, b_i) is the amount each currency changes after accepting trade offer i. If we have amounts k and q of currency A and B after trade $i-1$, then we will have amounts $k+a_i$ and $q+b_i$ if we accept trade i. Hence, if $d[i][k][q] = 1$ and we accept trade i, $d[i+1][k+a_{i+1}][q+b_{i+1}] = 1$ (for $i \in [n-1]$). This fact can be used to gain information about $d[i][k][q]$ from $d[i-1][k][q]$.

If $d[i][k][q] = 1$ but $d[i-1][k][q] = 0$, trade i must necessarily have been accepted. This means that before trade i, we had $k - a_i$ currency A and $q - b_i$ of currency B. Hence, $d[i-1][k-a_i][q-b_i]$ must have been equal to 1. This is the second term of the max function in the recurrence relation.

(As a note, it is possible that both the first and second terms in the recurrence relation equal to 1. This just means there are multiple ways to end up with k and q after trade i.)

Runtime Analysis:

The number of subproblems will be equal to $n \cdot (a_{start} + \sum_{i \in [n]} a_i) \cdot (b_{start} + \sum_{i \in [n]} b_i)$. Recall that $B = \max \{a_{start}, b_{start}, a_{target}, b_{target}, max_{i \in [n]}\{|a_i|\}, max_{i \in [n]}\{|bi|\}\}$. For both $(a_{start} + \sum_{i \in [n]} a_i)$ and $(b_{start} + \sum_{i \in [n]} b_i)$, $(n+1) \cdot B$ is an upper bound. Hence, the number of subproblems is equal to $n \cdot (n+1) \cdot B \cdot (n+1) \cdot B \rightarrow O(n^3 \cdot B^2)$. Each subproblem requires checking the values of 2 previous subproblems which can be done in $O(1)$ time. Hence, the total runtime is $O(n^3 \cdot B^2)$ meaning this algorithm is $poly(n, B)$.

Section (b)

1. CurrencyTrader is in NP.

Here is a simple non-deterministic polynomial time algorithm:

> For each trade i, randomly choose to accept i. If after any trade, the amount of currency A or B possessed is less 0, return NO. After the final trade, if the amount of currency A and B are both above the target values, return YES. Otherwise, return NO.

To see that this is a non-deterministic polynomial time algorithm for this problem, if there exists a sequence of decisions where the final currency amounts are above the target value, then there is some small probability this algorithm returns YES. If not, this algorithm will always returns NO.

Runtime: We must perform n steps that involve the following operations:

1. Randomly choose to accept or decline
2. Add a value to a sum and subtract a value from a sum
3. Check if two sums are less than zero

The time complexity of both (1) and (3) is $O(1)$. The time complexity of (2) depends on the size of the values and the size of the sums. An upper bound on the number of steps involved adding two number is the total number of bits needed to store both numbers. If we assume that B is at most $2^{poly(n)}$, then the number of bits need to represent any value will be $poly(n) + 1$. The size of either sum will be at most $n \cdot B$ thus requiring $(n \cdot poly(n)) + 1$ bits. Hence, an upper bound on the time complexity of operation (2) is $poly(n) + 1 + (n \cdot poly(n)) + 1 \rightarrow O(n \cdot poly(n))$. Performing these operations n times takes $O(n^2 \cdot poly(n))$.

Finally, as a last step, we must compare our final values with the target values.

The time complexity of comparing two numbers is also the total number of bits since in the worst case scenario, the i^{th} bit of each number gets compared. Since the number of bits needed to represent the target values and the final sums is polynomial in n (as we just explained), this step is also polynomial in n. Therefore, this algorithm takes $O(n^2 \cdot poly(n))$.

2. CurrencyTrader is NP-hard if B can be exponentially large

Recall that the NP-hard version of Knapsack we did in class asked: Given n non-negative weights $w_1, ..., w_n$ and a capacity C, is there a subset $S \subseteq [n]$, such that $\sum_{i \in S} w_i = C$.

We can reduce an instance of this version of KnapSack to an instance of CurrencyTrader in polynomial time. If CurrencyTrader was easy, then to solve KnapSack, we could just transform it into CurrencyTrader and solve it. Hence, it follows that CurrencyTrader is at least as hard as KnapSack, i.e., CurrencyTrader is NP-hard.

Reduction

Let (W, C) be an instance of KnapSack where W is the set weights, $\{w_1, ..., w_n\}$ and C is capacity. Let $T(W, C)$ be an instance of KnapSack that has been transformed into an instance of CurrencyTrader. In order to transform (W, C) into $T(W, C)$ we do the following:

1. Let $a_{start} = 0$, $b_{start} = C$, $a_{target} = C$ and $b_{target} = 0$.
2. Let n trade offers be made. For each trade offer $i \in [n]$, let $a_i = w_i$ and $b_i = -w_i$.

Accepting the i^{th} trade offer will represent accepting w_i. Hence, a solution to $(W, Q), S \subseteq [n]$ (where S is a set of indexes of chosen weights) corresponds to a solution to $T(W, Q), D \subseteq [n]$ (where D is a set of indexes of accepted trades). In other words, $S = D$.

We can perform this reduction in polynomial time as we just have to assign $4 + 2n$ variables.

Proof of Validity

Claim: If S is a solution to (W, C), then D is a solution to $T(W, C)$.

Proof: Assume S is a solution to (W, C). This means that $\sum_{i \in S} w_i = C$. For $i \in S$, if we accept the corresponding trade offer $i \in D$, then our final currency values will be:

$$a_{final} = a_{start} + \sum_{i \in D} a_i = 0 + \sum_{i \in D} w_i = C$$

$$b_{final} = b_{start} + \sum_{i \in D} b_i = C + \sum_{i \in D} -w_i = C - C = 0$$

Hence, we will have at least as much currency A and B as our targets meaning D solves $T(WC)$:

$$a_{final} = C \geq a_{target} = C$$

$$b_{final} = 0 \geq b_{target} = 0$$

Therefore, if there S solves (W, C), D solves $T(W, C)$. □

Claim: If D is solution to $T(W, C)$, then S is a solution to (W, C).

Proof: Assume D solves $T(W, C)$. This entails the following:

$$a_{final} = a_{start} + \sum_{i \in D} a_i \geq a_{target}$$

$$a_{final} = 0 + \sum_{i \in D} w_i \geq C$$

$$b_{final} = b_{start} + \sum_{i \in D} b_i \geq b_{target}$$

$$b_{final} = C + \sum_{i \in D} -w_i \geq 0$$

We can rewrite things as follows to :

$$\sum_{i \in D} w_i \geq C \rightarrow 0 \geq C - \sum_{i \in D} w_i$$

We can set up an inequality that will allow us to bound the sum of the weights:

$$C - \sum_{i \in D} w_i \leq 0 \leq C + \sum_{i \in D} -w_i$$

$$C - \sum_{i \in D} w_i \leq 0 \leq C - \sum_{i \in D} w_i$$

$$C \leq \sum_{i \in D} w_i \leq C$$

$$\sum_{i \in D} w_i = C$$

$$\sum_{i \in S} w_i = C$$

Hence, S solves (W, C) since the sum of weights with indexes in S is equal to capacity. □

Claim: S is a solution to (W, C) iff D is solution to $T(W, C)$.

Proof: Let P mean "S is a solution to (W, C)" and let Q mean "D is a solution to $T(W, C)$". By our previous to claims, we have $P \implies Q$ and $Q \implies P$, meaning we have $P \iff Q$. □

Therefore, we have shown CurrencyTrader is at least as hard as KnapSack meaning Currency-Trader is NP-hard.

3. CurrencyTrader is NP-Complete if B can be exponentially large since it is in NP and NP-hard