CMSC 27230: Problem Set 6

Owen Fahey

February 27, 2023

Problem 1

(a) This problem is in NP. One non-deterministic polynomial time algorithm for this problem is:

```
1. Start at $s$. Keep track of visited vertices and the length of the current path
2. While the length of the path is less than $k$ to do the following:
       (i) Travel to a random node connected to the current node
       (ii) If we have visited this node, RETURN NO
       (iii) Update the list of visited vertices and the length of the current path
3. Once we have traveled $k$ times, check if the current node is equal to $t$.
       IF TRUE, RETURN YES
       IF FALSE, RETURN NO
```

To see that this is a non-deterministic polynomial time algorithm, if a graph G has no exactly-k length paths from s to t then we will output NO regardless of what we guess and if G has any k-length paths from s to t then there is some possibility we output YES. The steps in the while loop are performed in constant time at most k times, hence this algorithm is polynomial.

(b) This problem is in NP because it is in P.

Briefly describing the algorithm:

This problem can be solved with dynamic programming in $O(n^2)$. Let $L[i]$ take an index and give length of the longest increasing subsequence with x_i as the first number. We will solve each subproblem, $L[i]$, in descending order of i from $L[n]$ to $L[1]$. Our final answer will be:

$$\max_{i:1\leq i\leq n}\{L[i]\}$$

The recurrence relationship for $L[i]$ can be written as follows:

$$L[i] = max\{1, \max_{j:i<j\leq n; x_i<x_j}\{1+L[j]\}\}$$

This algorithm is guaranteed to find the correct answer because it finds the length of the longest subsequence starting at every index and returns the maximum. For each subproblem, the recurrence relationships looks for the largest subsequence to which x_i could be added to the beginning. If no such subsequence exists then, the largest subsequence with x_i as the first number is just x_i. It runs in polynomial time since for each subproblem with index i, we must check all subproblems between i and n. $\sum_{i=1}^{n}(n-i) = \frac{1}{2}n^2 - \frac{1}{2}n \to O(n^2)$.

(c) This problem is (likely) not NP because it is co-NP-hard for $k > 3$. Assume $k > 3$.

Argument:

1. Claim: k-coloring is NP-hard

   Recall that 3-coloring is NP-hard. We can reduce 3-coloring to k-coloring in polynomial time. In order to perform this reduction:

   Given a 3-coloring problem with graph G, make a new graph, G' that is a duplicate of G with $k - 3$ extra vertices connected to every other vertex (including each other). If G' is k-colorable then G is 3-colorable. To see why this is the case, observe that each added vertex in G' must have a unique color since each of these vertices is adjacent to every other vertex in the graph. Hence, since any 3-coloring problem can be reduced to a k-coloring problem in polynomial time and we know 3-coloring is NP-hard, we can conclude that k-coloring is NP-hard.

2. Claim: If $\chi(G) = k$, then G cannot be colored by $k - 1$ colors.

   If this were possible then k would not be the minimum number of colors needed to G. This would imply $\chi(G) \neq k$ which is a contradiction.

3. In order to know $\chi(G) = k$, we must know that

   (i) G can be colored by k-colors

   (ii) G cannot be colored by $k - 1$ colors.

   Observe that (ii) is the complement of the $(k - 1)$-coloring problem.

4. Since $(k - 1)$-coloring is NP-hard, we can conclude its complement is co-NP-Hard because of the fact that: a problem is co-NP-hard iff its complement is NP-hard. Since this problem relies on a the complement to k-coloring, we can conclude this problem is co-NP-hard.

   (An observation: If $k \leq 3$, then the complement to $k - 1$-coloring is not co-NP-hard)

(d) This problem is in NP. One non-deterministic polynomial time algorithm for this problem is:

1. Assign each player randomly to a team. After an assignment, update the number of players on the team and the team's total skill level.
2. Once finished:

```
IF the number of players on each team is not equal, RETURN NO
ELSE IF the total skill level of each team is not equal, RETURN NO
ELSE RETURN YES
```

To see that this is a non-deterministic polynomial time algorithm, if is not possible to divide the teams into two equal groups with the same total skill level then we will output NO regardless of what we guess and if this is possible, then there is some possibility we output YES.

This algorithm performs the following operations n times: (i) add 1 to the number of players on a team and (ii) add a player's skill to a running sum. Then it compares the number of players on and the total skill level of A and B. The time complexity of summing and comparing of total skill levels is strictly greater than the time complexity of summing and comparing the number of players. Hence, we will only worry about the time complexity of the former.

An upper bound on the number of steps involved adding two number is the total number of bits needed to store both numbers. 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.

In the worst case, every player has a skill level of 2^{100n} which takes $100n + 1$ bits to represent. The largest possible total skill level is $n \cdot 2^{100n} = 2^{100n+log(n)}$ which takes $100n + log(n) + 1$ bits to represent. Hence, an upper bound on the number of steps needed to add some player's skill level to a total skill level is $200n + log(n) + 2 \rightarrow O(n)$. Since we perform $n + 1$ addition/comparison operations, we have $(n + 1) \cdot O(n) \rightarrow O(n^2)$. Hence, the time complexity of the algorithm is $O(n^2)$.

Problem 2

Setup:

This problem is a matching problem that can be reduced to a max flow problem. Let us set up a directed graph by doing the following:

1. Create a start node s and a sink node t
2. For $j \in [n]$: create node v_j to represent the j^{th} volunteer
3. For $j \in [n]$: create an edge going from s to v_j with capacity c_j
4. For $i \in [k]$: create node τ_k to represent the k^{th} job type
5. For $j \in [n]$: for $i \in [k]$: create an edge going from v_j to τ_i with capacity b_{ij}
6. For $i \in [k]$: create an edge going from τ_i to t with capacity a_i

Example:

Let us set this graph up for the example we are given. Let Margaret be v_1, Nancy be v_2 and Owen be v_3. We should get as follows:

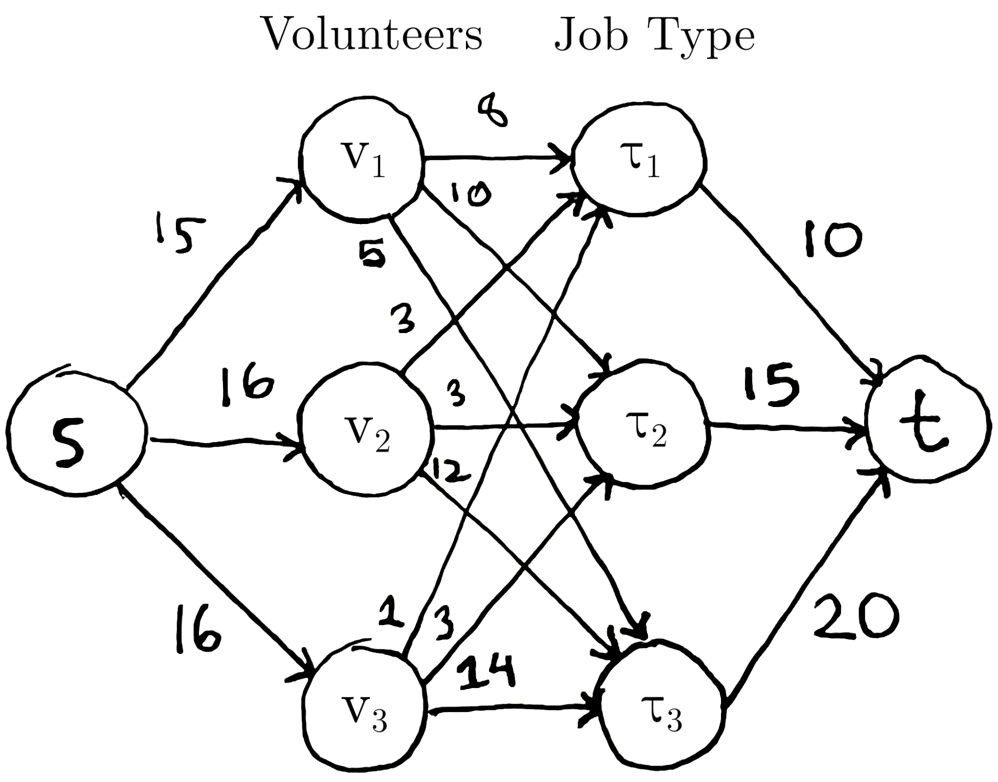

Figure 1: Setting up the graph on the example

Explanation of Correctness:

This graph has an integer-valued flow of at least z if and only if there is an arrangement of z jobs that satisfies all the of the constraints. Suppose we have an arrangement of z jobs where no more than a_i jobs are done for job type i, no volunteer does too many jobs of a given type and no volunteer does too many jobs.

For every job of type i down by volunteer v_j, we can route one unit of flow along the path $s \to v_j \to \tau_i \to t$. Combining these flows, we obtain a flow of value z that satisfies the condition that for every vertex $v \in V(G)\backslash\{s,t\}, f_{in}(v) = f_{out}(v)$.

Given an integer-valued flow, we make the following assignments:

> Let f_{ji} be the flow on edge $v_j \to \tau_i$. For each j, for each i, if the edge $v_j \to \tau_i$ exists, assign f_{ji} jobs of type i to volunteer j. If the edge $v_j \to \tau_i$ does not exist, assign 0 jobs of type i to volunteer j.

Some observations:

1. Since the total capacity into each v_j is c_j, the maximum possible flow going out of v_j is c_j. Hence, v_j cannot be assigned more than c_j total jobs.
2. Since the capacity between v_j and τ_i is b_{ij}, the maximum possible flow from v_j to τ_i is b_{ij}. Hence, v_j cannot be assigned more than b_{ij} jobs of type i.
3. Since the capacity out of τ_i is a_i. We cannot assigned more than a_i jobs of type i to volunteers. In other words, we do not over-assign jobs.

Thus, to find an optimal way to assign jobs, we run Ford-Fulkerson on the graph to obtain an integer-value max flow and make assignments as described above.

Runtime analysis:

The time complexity of this algorithm can be broken down into:

1. Constructing the graph
2. Running Ford-Fulkerson

Constructing a graph with edges and vertices will take O($|V| + |E|$). The number of vertices is as follows:

$$\begin{aligned} |V| &= |\{s\}| + |\{t\}| + n + k \\ &= 2 + n + k \end{aligned}$$

The number of edges is as follows:

$$|E| = n + k + nk$$

Hence, the amount of time it takes to construct the graph is equal to $O(2+2n+2k+nk) \to O(nk)$

The runtime of Ford-Fulkerson is $O(F \cdot |E(G)|$ which depends on max flow F. The max flow through this graph is the total amount of jobs that needs to get done which is equal to the quantity m. Hence, $O(F \cdot |E(G)| = O(m \cdot O(nk)) \to O(mnk)$.

Problem 3

1. This problem is in NP

One non-deterministic polynomial time algorithm for this problem is:

```
1. Promote $k$ random people to head volunteer
2. For each event:
        Check to see if there is one head volunteer
        IF NOT, RETURN NO
3. RETURN YES (We only get here if every event has at least one head volunteer)
```

To see that this is a non-deterministic polynomial time algorithm, if it is not possible to promote k head volunteers and cover all events, then we will output NO regardless of what we guess and if it is possible to promote k head volunteers and cover all events then there is some possibility we output YES. This algorithm clearly runs in polynomial time since it takes $O(k)$ to promote k people at most $O(nk)$ to check all the events for a head volunteer.

2. This problem is NP-hard

Recall that vertex cover is NP-hard and asks the following: 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 vertex cover to event supervision in polynomial time. If event supervision was easy, then to solve vertex cover, we could just transform it into event supervision and solve it. Hence, it follows that event supervision is at least as hard as vertex cover, i.e, event supervision is NP-hard.

Reduction: Let G be a graph in a vertex cover problem of size k. For $n \in |E(G)|$, let (u_n, v_n) be the n^{th} edge in G (the specific order does not matter). For each $(u_n, v_n) \in E(G)$, let $j_{u_n} = u_n$ and $j_{v_n} = v_n$ be volunteers who will be at event n (note: vertex, u_n, might be the same as u_m or v_m for $m \in |E(G)| : m \neq n$). In other words, given (G, k), an instance of vertex cover, let the vertices be the volunteers and the edges correspond to the events where k is the number of volunteers we want to promote. Let H be set of k head volunteers that solves this event supervision problem (if a solution exists). I claim that such a set H exists if and only if there exists a solution to the original instance (G, k) of vertex cover. For notational purposes, let $T(G, k)$ denote the transformed version of vertex cover. Then, since vertices are mapped bijectively (the n vertices get mapped to n volunteers) we can say $T(V)$ for $V \subseteq V(G)$ to refer to the volunteers that correspond to those in V.

This reduction can be done in polynomial time. We need to represent each edge as an event and each vertex as a volunteer. Hence, the time complexity of the reduction grows polynomially with the number of edges and vertices in the graph.

Showing the reduction is valid: Assume there is a set V of size k that solves vertex cover for graph G. For every $(u, v) \in E(G)$, $u \in V$ or $v \in V$ (or both). Since the events in $T(G, k)$ correspond to edges in (G, k), it follows that every event is of course covered by at least one vertex in $T(V)$, since V is a vertex cover so by definition, every edge is touching at least one vertex in V — this corresponds to the fact that every event has at least one head volunteer. Conversely, if V is a set of k head volunteers of $T(G, k)$ such that every event has at least one head volunteer, then $T^{-1}(V)$ is the vertex cover. Specifically, every event is supervised by at least one head volunteer, so by design, every edge in G is covered by at least one vertex in $T^{-1}(V)$. Essentially, I have used many words to show the obvious fact that (G, k) is a YES instance of vertex cover iff $T(G, k)$ is a YES instance of event supervision. In order to solve both problems, at least one of the two vertices/volunteers per edge/event must be part of the covering set/the set of head volunteers. Hence, we have shown that event supervision is at least as hard as vertex cover. Therefore, event supervision is NP-hard.

3. This problem is NP-complete because it is in NP and NP-hard.