CS 330 – Fall 2018, Assignment 6 Solutions

Question 1 (20 pts)**.** *Chapter 8, exercise 4 on pp. 506.*

Let RR denote the Resource Reservation Problem.

a.) RR is NP-complete. First, RR is in NP. A certificate for this is the set of processes that can be satisfied at the same time. We can check this in polynomial time, as we simply need to compare each two satisfied processes and check that they don't share resources.

Second, RR is NP-complete as there is a polynomial time reduction from Independent Set (IS) to RR.

Claim 1. *IS* $\leq_p$ *RR*

Proof. **Instance of IS** $\rightarrow$ **instance of RR:** An instance IS is the graph $G(V, E)$ and integer k. We define an instance of RR as follows. For each node $v \in V$ define a process p_v in RR. For each edge $e \in E$ define a resource r_e in RR. Assign to process p_v each resource corresponding to an edge adjacent to v. Thus, if R_v is the set of resources required for v, then $R_v = \{r_{e(u,v)} \mid e(u, v) \in E\}$.

proof that the IS instance has an optimal solution of size k **iff the RR instance has:** First, assume that there is an independent set $S \subseteq V$ of size k in G. Then the corresponding processes $P_S = \{p_v \mid v \in S\}$ can be satisfied at the same time. This is by definition of independence and the fact that resources correspond to edges in G. Since nodes in S don't share an edge, processes in P_S don't share a resource. Second, assume that there is a solution S_P to RR with k processes that are satisfied. That means that the resources of the processes in S_P do not overlap. But since resources correspond to edges in G this also implies that the nodes corresponding to processes do not share edges. Thus, $S = \{v \mid p_v \in S_P\}$ is indeed an independent set of size k in G. □

b.) For $k = 2$ this question simply asks whether there are two processes which don't have any resource in common. This can be decided in polynomial time. Simply, for every pair of processes p_1 and p_2 check whether they have any resource in common. If yes, then move on to the next pair, if no, then output them as solution. Since each process may require at most m resources, checking a pair of processes takes at most $O(m)$ time. There are $O(n^2)$ pairs, resulting in an $O(n^2m)$ algorithm.

c.) This problem can be reduced in polynomial time to the maximum matching problem which we know from class that it is polynomial. Hence, this version of RR is polynomial too. Define a bipartite graph $G = (U \cup V, E)$ as follows; let U correspond to the people among the resources and nodes in V correspond to a piece of equipment. There is an edge (u, v) if there is a process that requires person u to work on equipment v. We can see that a valid solution to RR defines a matching in this bipartite graph. There is a solution to RR of size k iff there is a matching of size k.

d.) This problem is NP-complete too Independent Set can be reduced to it. Notice that the reduction IS $\leq_p$ RR in part **a.)** holds for this case as well. Since resources in **a.)** correspond to edges, it is clear that each resource is requested by exactly two processes.

Question 2 (10 pts)**.** *Chapter 8, exercise 10 on pp. 510.*

(a) Strategic Advertising is in NP: given a specification of ad locations $\{v_1, v_2, \ldots v_k\}$, we can do a brute force check that each path does in fact contain one of the ad locations. With a worst-case path length of $|E| = m$ and linear scanning, certification is still O(mtk) time.

To show NP-hardness, we will show: Vertex Cover $\leq_p$ Strategic Advertising.

Take an arbitrary instance of the Vertex Cover problem: an undirected graph $G = (V, E)$ and a number k. Our Strategic Advertising graph G' will be a copy of G but with directed edges E', one for each edge in E, oriented arbitrarily. The paths P_i will be the set of m one-hop paths $u \rightarrow v$, one for each edge (u, v) in E'.

Claim 2. *There is a vertex cover of size k in G iff there is a set of k ad locations in G' that cover all paths $\{P_i\}$.*

Proof. $\rightarrow$: Assume there is a vertex cover (VC) of size k in $G = \{v_1, v_2, \ldots, v_k\}$. We will use this same set to cover all paths $\{P_i\}$. An ad appears on an arbitrary path P_i because it is a path $u \rightarrow v$, which corresponds to an edge (u, v) in G, and by definition of vertex cover, either u or v is in the VC, so either u or v contains an ad.

$\leftarrow$: Assume k ads are placed such that an ad appears somewhere along all paths $\{P_i\}$. Call this set of k vertices an Ad Cover (AC). We claim that this set is a vertex cover on G. Consider an arbitrary edge (u, v) in G. By our path construction, a directed path corresponding to this edge is some path P_i, and therefore, either u or v must appear in AC, and thus is also covered by the VC. □

(b) This is an example of "self-reducibility": converting a decision algorithm into a search algorithm.

First, make an initial call to the yes/no algorithm S on the provided input. If it answers no, then output no. This is correct, since S is correct.

If it answers yes, now we have to find the vertices. The key step is to efficiently test whether vertex i is a member of an Ad Cover of size k. If it is, then there must be a set of $k-1$ other vertices that cover all the paths that do not include vertex i, and conversely (key point), if it is not, then there is no set of $k-1$ other vertices that cover all the paths that do not include vertex i. This observation serves as our proof of correctness.

```
AC = { };     // initialize ad cover to empty set
P = {P_i};    // P is all paths, initially

foreach vertex v in 1 to n do:
    let Q = the set of paths in P that contain v;

    if S (G, P \ Q, k - 1) == yes:
        add v to AC;
        P = P \ Q;       // remove Q from P
        k = k - 1;

    if k == 0: break;    // found the set

output AC
```

The loop runs for n iterations. Identifying the set Q takes time O(tn), similar to what we did in (a). Also, we make one call to S per iteration. Work after the call returns is fast, O(t) to remove Q from P.

Total time: O(n) calls to S and O(tn^2) other steps.

[Important note: It is not correct to test vertices one at a time without reducing k along the way. Vertex 1 might be part of one Ad Cover of size k, and vertex 2 might be part of a different Ad Cover of size k, but they might never jointly appear in the same ad cover of size k!].

Question 3 (10 pts)**.**

Suppose that someone gives you a black-box algorithm A that takes an undirected graph $G = (V, E)$, and a number k, and behaves as follows.

- If G is not connected, it simply returns G is not connected.
- If G is connected and contains a simple path of length at least k, it returns yes.

- If G is connected and does not contains a simple path of length at least k, it returns no.

Suppose that the algorithm A runs in time polynomial in the size of G and k. Show how, using calls to A, you could then solve the Longest Path Problem in polynomial time: Given an arbitrary undirected graph G, and a number k, does G contain a simple path of length at least k?

Solution. In this problem we don't need to do any reduction as we are not proving complexity. Let $G(V, E)$ be the graph that we want to test whether it contains a path of length k. We can see that if G is connected, then the algorithm on G as an input will return $A(G, k) ==' yes'$ iff G contains the path and will return *'no'* otherwise. The only "incorrect" answer we receive is when G is disconnected. We can circumvent this by passing the connected components of G to A and return *'yes'* if any of the components returns *'yes'*. We find the components by running any algorithm on G suitable for this task, for example BFS or DFS.

```
DisconnectedLongestPaths(G(V,E),k):
 While |V|>0:
     v = random node in V
     G_v = BFS(V,E,v)  //run BFS on remaining graph to find conn. comp. of v
     if A(G_v,k) == 'yes':
         return 'yes'
     V = V \ V_v
     E = E \ E_v
  return 'no'
```

Running time. Observe that we have at most n iterations. Thus, there are at most $O(n)$ calls to A which we know is polynomial. We can give a more efficient evaluation for the calls to BFS; since the running time to find the component of node v only depends on the size of that component, the running times sum up to $O(n + m)$. We denote the running time of A by $O(A)$. Then the overall running time of the algorithm is $O(nA + n + m)$.

Question 4 (10 pts)**.**

A company has two trucks, and must deliver a number of packages to a number of addresses. They want both drivers to be home at the end of the day. This gives the following decision problem: You are given a set of locations L, with for each pair of locations $v, w \in L$, a distance $d(v, w)$, a starting location $s \in L$, and an integer k. Find two cycles, that both start in s, such that every location in L is on at least one of the two cycles, and both cycles have length at most k? Show that this problem is NP-complete.

Solution. We will refer to this problem as the Truck Problem (TP). First, we need to check whether TP is in NP. For this, the certificate will be the two specific truck routes. And the Certifier algorithm is to trace both routes (at most n steps) and check that their length is not more than k and that they touch all required locations.

To show NP-completeness, we reduce the Traveling Salesperson (TSP) problem *to* TP. TSP consists of a weighted graph $G(V, E, w)$, where nodes correspond to cities and edge weights correspond to the distance between the cities, and an integer k. TSP returns 'yes' if there is a tour of weight at most k that touches all cities. Note, that since the salesperson needs to go to every city, the problem does not change if we specify a starting point.

Generating the TP instance. Now, given this instance of TSP we have to define the appropriate instance of TP. Notice that the two problems are quite similar, except that TP has two routes instead of one. We can alleviate this by adding an extra cycle of weight k to the starting point of the trips. This way one of the trucks has to go along the cycle. Thus, the final construction is the following. Take the instance of TSP, i.e. the weighted graph $G(V, E, w)$ and the number k and define the following TP instance: pick a node $s \in V$ at random as the starting point. Add

$k-1$ new nodes to G, index them $v_1, v_2, \ldots v_{k-1}$. Add an edge (s, v_1) with weight 1, another edge (v_{k-1}, s), and edges (v_i, v_{i+1}) for every $i = 1 \ldots k-2$ all with weight one. We denote the enhanced graph by G'.

Proof that the two instances have related solutions. We claim that the TP problem with the input of graph G', node s and integer k will return 'yes' iff TSP on the input of G and k returns 'yes'. To prove this, assume that TSP returns 'yes', thus there is a tour through G of length at most k that traverses every node. In G' we will define the two truck routes as follows; the first truck will travers the cycle that we added to node s. The second truck will start in s and traverse the route that is the solution to the TSP instance. It is clear that both truck routes have length at most k. Now assume, there is a solution to TP on this instance. That is, there are two truck routes, each of length at most k, such that they contain each node in the graph. We may assume that one of these two routes consist of going around the added cycle (See proof for this at the end). Then it is clear, that the other route contains all the original nodes in V. This second route is the tour that is the solution for TSP.

(To prove that there exists a solution to TP were one of the truck routes contains the entire added cycle is the following. Assume, that both truck routes contain some part of the cycle. This is possible if each truck starts going around the cycle, but then turns back at a certain node. Assume the first truck turns back at node v_i. We can make two observations; 1. the remaining part of the cycle (not covered by the first route) has weight $k - \left(w(s, v_1) + \sum_{j=1}^{i-1} w(v_j, v_j + 1)\right)$, 2. the part of the first route that is not on the cycle has the same weight. Hence, we can reroute the two trucks, so that the first route consists of the cycle, and the second route consists of the non-cycle part of the two original routes.)