CSC373 - Problem Set 2

Authors: Luke Bacchus, Naslin Rahman, Zhuoqian Li

February 3, 2021

Question 1

This algorithm is very similar to the Interval scheduling problem done in class except we have an circle of problems (similar to a clock). We can apply the same algorithm if there is only one job that intersects wuth the 12 am time slot. So we do this for each each one of these jobs.

The algorithm goes as follows:

1. For all intervals that contain the 12am slot put them into a set called S
2. Sort the remaining intervals in order of earliest finish time (EFT)
3. Create a new set A and select a job s from S and insert into the sorted intervals in step 2
4. for each interval i in sorted order, if it is compatible with A then add it to A.
5. Repeat steps 3 and 4 for each job in S
6. Now for each set we created, return the one with greatest length

The run time of this algorithm is O(n+ cnlogn + cn^2 + n) which is $O(n^2)$
This question was written and read by Luke, Naslin, and Zhuoqian.

Question 2

a. Our goal is to find the easiest trip which is the trip with the minimum toughness. The toughness of a trip is the greatest degree of difficulty among all portages that trip. Originally, in Dijkstra's algorithmn, we have d(v) = the sum of the weights between nodes/lakes on the path to node v. However, the proposed modification is to assign d(v) as the max of the weights (aka "difficulties") of the portages between lakes.

Below is the pseudocode:

```
function find(s, t, L, n):
    R = [s]
    d[s] = 0
    V = [1,2,...,n]

    for v = 1 to n:
        if v != s:
            if v in L[s]:
                v_weight = L[s][v][1]
                R.append(v)
                d[v] = v_weight
                p[v] = s
            else:
                R.append(inf)
                d[v] = inf
                p[v] = nil

    while R != V:
        not_R = V - R
        not_d = []

        for v in not_R:
            not_d[v] = d[v]

        u = not_d.get_index(min(not_d))
        R.append(u)

        for v = 1 to n:
            if v != u and v in not_R:
                if v in L[u]:
                    if max(d[u], L[u][v][1]) < d[v]:
                        d(v) =  max(d[u], L[u][v][1])
                        p(v) = u

    return d(t)
```

The first for loop would have a runtime of c*n, where c is a constant. The while loop would take at most n iterations, while the first for loop within the while loop would take m runtime, where $m <= n$. The second for loop in the while loop would take at most n iterations. Thus the worst case runtime of the modified algorithm is $O(c+cn+m+n^2)$ and thus $O(n^2)$.

b. In Dijkstra's original algorithm, d(v) is a sum of the weights. It assumes that when a new a new node, u, is added to R, for all nodes, v, in V - R whose R-path contain u, $d(v) = min(d(u) + w_uv, d(v))$. This relies on non negative weights since it relies on the shortest path to v being either the original path $s- > v$ or the path $s- > u +$ v. Negative weights would allow this assumption to be false since a negative weight connected to v elsewhere could lessen the weight of another path, not mentioned previously, making it possible to be the shortest path to v.

This assumption is not necessary since our modification takes the max weight of all the portages on the path. Thus, negative weights would not impact the outcome.

This question was written and read by Luke, Naslin, and Zhuoqian.

Question 3

a. Consider the actor heights of $900, 30, 15$, and the costume lengths of $50, 10, 5$. The optimal solution is matching $900 : 50$, $30 : 10$, and $15 : 5$ for a cost of $850+20+10 = 880$. However, Bombazzino's algorithm will match $15 : 10$, $30 : 50$, and $900 : 5$ for a cost of $5 + 20 + 895 = 920$. Thus, Bombazzino's algorithm is not optimal.

b. Let S_g be the solution that our greedy algorithm returns, and let S_o be the optimal solution. We will show that the cost of $S_g = S_o$, therefore S_g is optimal.

   To start, sort S_g and S_o by increasing actor height. Let h_1 to h_k be the distinct heights $h_1 > h_2 > ... > h_k$ of the actors. Starting at $i = 1$, consider the actors of height h_i, and let C_i be the set of costumes assigned to the actors of height h_i in solution S_g. Let D_i be the set of costumes assigned to the actors of height h_i in S_o. If $C_i = D_i$ than the cost of the assignment to the actors of h_i in S_g is the same as the cost of the assignment to the actors of height h_i in S_o and this holds for h_1 up to h_i.

   Say that $C_i \neq D_i$. Let a_i be an actor of height h_i in S_g, such that in S_o, a_i has a costume c_j that is not a member of the set of costumes C_i. Therefore, the length of c_j is less than the length of any costume in C_i, and there must also exist an actor a_j with height less than h_i such that a_j has been given some costume c_i where c_i is in C_i in S_o.

   Therefore, we have that the height of actor $a_i > a_j$, and the length of costume $c_i > c_j$. This is a contradiction, as then the optimal solution could be improved by swapping the costumes for a_i and a_j (in an optimal solution, there cannot be a pair of actors a_i a_j where a_i is taller than a_j and a_i is assigned a costume that is shorter than a_j's costume. We know this since $ai > aj$ and $ci > cj$, which implies that $ai - cj > ai - ci$. This tells us that $cost(ai\ cj) > cost(ai\ ci)$, which must be suboptimal, as swapping the costumes for a_i and a_j will strictly improve the cost of the solution). Therefore, we see that in the optimal solution C_i must be equal to D_i for all i, and therefore the cost is identical.

This question was written and read by Luke, Naslin, and Zhuoqian.