# Sample Solution for Problem Set 11

Data Structures and Algorithms, Fall 2022

February 14, 2023

## Contents

# 1 Problem 1

**(a)** Here is the counterexample. Let activities $S = \{a_1, a_2, a_3, a_4\}$ be

$$a_1 = (s_1, f_1) = (1, 2), a_2 = (s_2, f_2) = (3, 7), a_3 = (s_3, f_3) = (6, 9), a_4 = (s_4, f_4) = (8, 15).$$

The greedy solution is $\{a_1, a_3\}$. The optimal solution is $\{a_1, a_2, a_4\}$.

**(b)** It is correct. The proof is similar to the one in the slides.

**(c)** Here is the counterexample. Let activities $S = \{a_1, a_2, a_3, a_4, ..., a_{11}\}$ be

$$a_1 = (s_1, f_1) = (1, 4), a_2 = (s_2, f_2) = (5, 8), a_3 = (s_3, f_3) = (9, 12), a_4 = (s_4, f_4) = (13, 16),$$

$$a_5 = a_6 = a_7 = (3, 6), a_8 = (7, 10), a_9 = a_{10} = a_{11} = (11, 14).$$

The greedy algorithm first select $a_8$, which conflict with the only optimal solution is $\{a_1, a_2, a_3, a_4\}$. Notice that the remaining activities contains the non-selected activities that is not compatible with previously selected activities.

**(d)** Here is the counterexample. Let activities $S = a_1, a_2, a_3$ be

$$a_1 = (s_1, f_1) = (1, 5), a_2 = (s_2, f_2) = (2, 3), a_3 = (s_3, f_3) = (4, 5).$$

The greedy solution is $\{a_1\}$. The optimal solution is $\{a_2, a_3\}$.

## 2 Problem 2

### (a)

**Algorithm** Sort the activities by processing time in increasing order. Process the activities in this order.

**Complexity** The running time of the algorithm is the same as the sort algorithm, which can be $O(n \log n)$.

**Correctness** Let $d_1, d_2, ..., d_n$ denote the processing order. The average of completion time is

$$\frac{1}{n} \sum_{i=1}^{n} \sum_{j=1}^{i} d_j = \frac{1}{n} \sum_{i=1}^{n} (n - i + 1) \cdot d_i.$$

Here, $d_1$ is added the most times, then $d_2, d_3, ..., d_n$.

As a result, $d_1$ must be the activity that costs minimum time, and then $d_2, d_3, ..., d_n$. If there exists $i < j$ that satisfies $d_i < d_j$. Swap the $i$-th activity and the $j$-th activity, we simply get a new order with less average completion time.

### (b)

**Algorithm** Sort the activities by releasing time. Use a min-heap, which compares the remaining processing time of the activities. Each time an activity comes, we add it into the min-heap and choose the activity with least remaining processing time in the min-heap to do. Once an activity is finished, if the min-heap is not empty, choose the activity with least remaining processing time in the min-heap to continue doing.

**Complexity** Consider the complexity of sorting and maintaining min-heap, the running time is still $O(n \log n)$.

**Correctness** The correctness proof is almost the same as in (a). Assume at time $t$, we choose to do a task with $d_i$ remaining processing time instead of a task with $d_j$ time, where these two tasks are released before time $t$. If $d_i > d_j$, we simply get a new order with less average completion time by first process the task with $d_j$ remaining processing time.

## 3 Problem 3

Instead of grouping together the two with lowest frequency into pairs that have the smallest total frequency, we will group together the three with lowest frequency in order to have a final result that is a ternary tree. The analysis of optimality is almost similar to the binary case.

# 4 Problem 4

**Algorithm** The minimum number of colors needed to proper color $X$ is the maximum number of intervals that satisfies their intersection is not empty.

```
1  func color(L, R, n)
2      Sort L
3      Sort R
4      i = 1
5      j = 1
6      ans = 0
7      cur = 0
8      while i <= n do
9          if L[i] <= R[j]
10             i++
11             cur++
12         else
13             j++
14             cur--
15         if cur > ans
16             ans = cur
17     return ans
```

**Analysis** The running time of the algorithm is $O(n \log n)$.

# 5 Bonus Problem

## (a)

**Algorithm** Simple greedy algorithm. Suppose the remaining element set is $R$. Keep picking the subset $S'$ with minimal $\frac{c(S')}{|R \cap S'|}$.

**Analysis** We prove that if optimal solution costs $OPT$ , then the simple greedy algorithm costs at most $O(OPT \cdot \ln n)$.

Let $R_t$ denote the remaining elements after $t$ iteration. Let $n_t = |R_t|$. Thus, we have $n_0 = n$. This $n_t$ remaining elements can be covered by some sets which together costs $k$.

Thus, there must exist a subset $S'$ that satisfies $\frac{c(S')}{|R_t \cap S'|} \leq \frac{OPT}{n_t}$. Suppose after $t'$ iterations, we have $n_{t'} = 0$. Then, we have the total cost $C$ satisfies

$$C \leq \sum_{i=1}^{t'} (n_{i-1} - n_i) \cdot \frac{OPT}{n_{i-1}} \leq OPT \cdot (\frac{1}{n} + \frac{1}{n-1} + ... + \frac{1}{2} + 1) \leq O(OPT \cdot \ln n).$$

It is easy to verify that the running of the above algorithm is polynomial with respect to the input length.

## (b)

Based on the analysis of the algorithm, the corresponding problem instance can be given easily as follow.

- Define the ground set $U = \{1, 2, ..., n\}$.
- Define the collection of the subsets $S = \{S_1, S_2, ..., S_n\} \cup \{\hat{S}\}$, where $S_i = \{i\}$ and $\hat{S} = \{1, 2, ..., n\}$.
- Define the cost function as follow. For each $1 \leq i \leq n$, $c(S_i) = \frac{K}{2(n-i+1)}$. $c(\hat{S}) = K$.

The solution produced by the simple greedy algorithm is $\{S_1, S_2, ..., S_n\}$ which cost $\Omega(K \log n)$. However, the optimal solution is $\{\hat{S}\}$, which costs $K$. Thus, our algorithm produces a solution that costs $\Omega(OPT \cdot \ln n)$, which proves that the approximation ratio is asymptotically tight.

## 6 Problem 5

Omitted.

# 7 Problem 6

## (a)

Using disjoint set. First, initialize each vertex is in a disjoint set. Then, traverse through all the edges. For each edge $e = (u, v) \in E$, if $l_e \leq L$, merge the sets they are in. To determine whether there is a feasible route from $s$ to $t$, we query whether vertex $s$ and vertex $v$ is in the same set.

It is easy to verify that the running time of this algorithm is $O(|V| + |E|)$.

## (b)

Using a variant of Dijkstra's algorithm. The only difference betweem our algorithm and the Dijkstra's algorithm in the slides is that, instead of updating $v$ as

$$v.dist = u.dist + w(u, v)$$

, we use

$$v.dist = \min(u.dist, l_e)$$

, where $e = (u, v)$. It is eassy to verify that the running time of this algorithm is $O((|V| + |E|) \log |E|)$.