

CSOR W4231 Analysis of algorithms I  
Assignment 3

Xijiao Li (xl2950)

October 22, 2020

## Problem 1

We will use greedy algorithm here. The algorithm will be as follow:

```
Initialize an empty list A to store the placement of guards
for  $x_i$  in X
    if A is empty or  $x_i > A[-1]$ :
        place a guard  $a$  on  $1 + x_i$  and append  $a$  to the end of A
    else:
        continue to  $x_{i+1}$ 
```

Prove the optimality of the algorithm:

Let  $A = \{a_1, a_2, \dots, a_k\}$  denotes the solution generated by the above algorithm, and let  $O = \{o_1, o_2, \dots, o_m\}$  be an optimal solution.

Property 1:

Based on our strategy,  $a_1 = x_1 + 1$ . If the first placement  $o_1$  of  $O$  is on the right side of  $a_1$ , i.e.,  $a_1 < o_1$ , then it cannot guard  $x_1$ , since  $x_1 = a_1 - 1 < o_1 - 1$ . This leads to a contradiction, so  $o_1$  must be on the left side of  $a_1$ . Therefore, replacing  $o_1$  in  $O$  with  $a_1$  yields another feasible (and optimal) solution, since every  $x_i$  guarded by  $o_1$  can be guarded by  $a_1$ , too. Proof:

Given  $o_1 - 1 \leq x_i \leq o_1 + 1$ ,  
we have  $o_1 - 1 < a_1 - 1 = x_1 \leq x_i \leq o_1 + 1 < a_1 + 1$ ,  
thus,  $a_1 - 1 \leq x_i < a_1 + 1$

Therefore, there is an optimal solution whose first guard agrees with the greedy algorithm above.

Property 2:

Once we placed the first guard, suppose that it can guard  $x_1 \dots x_k$ . So the remaining guards must be placed to protect  $X' = \{x_{k+1} \dots x_i\}$ . This is a subproblem of the original one, and we can again use the above steps to prove that at each placement  $i$ , replacing the choice  $o_i$  of an arbitrary optimal solution with the choice  $a_i$  of greedy can yield another feasible optimal solution.

Therefore, we have proved the optimality of our greedy algorithm.

## Problem 2

a.

We will use greedy algorithm here. The algorithm will be simple as follow: each time we pick the task with smallest completion time to run, and repeat until we finish running all the tests.

Since repeatedly calculating and selecting the shortest task is redundant and takes lots of time, we can first sort the tasks, and then go through the sorted list of tasks once to run them one by one.

```
SCHEDULE(S)
    sort S in ascending order by completion time
    return S as our ordering for runing the tasks
```

Prove the optimality of the algorithm:

Let  $G = \{s_1, s_2, \dots, s_n\}$  denotes the tasks' ordering generated by the above algorithm, where  $s_i$  is the index of the task being schedule as the  $i$ th to run, and let  $O = \{o_1, o_2, \dots, o_n\}$  be an optimal solution of ordering.

Property 1:

Based on our strategy, we know that task  $a_{s_1}$  has the smallest completion time. Assume that  $o_i = s_1$  in  $O$  and  $o_i \neq o_1$ , so we can obtain a new ordering

$$O' = \{o_i, o_2, \dots, o_{i-1}, o_1, o_{i+1}, \dots, o_n\}$$

by simply swapping  $o_1$  and  $o_i$  in  $O$ . The goal is to minimize the average completion time  $1/n \sum_{i=1}^n c_i$ , so we calculate the average completion time of  $O$  and  $O'$ :

$$\begin{aligned} T_{avg}(O) &= 1/n [p_{o_1} + (p_{o_1} + p_{o_2}) + \dots + (p_{o_1} + p_{o_2} + \dots + p_{o_{i-1}} + p_{o_i}) + \dots + (p_{o_1} + \dots + p_{o_n})] \\ &= 1/n [np_{o_1} + (n-1)p_{o_2} + \dots + (n-i)p_{o_i} + \dots + 1 \times p_{o_n}] \\ T_{avg}(O') &= 1/n [p_{o_i} + (p_{o_i} + p_{o_2}) + \dots + (p_{o_i} + p_{o_2} + \dots + p_{o_{i-1}} + p_{o_1}) + \dots + (p_{o_i} + \dots + p_{o_n})] \\ &= 1/n [np_{o_i} + (n-1)p_{o_2} + \dots + (n-i)p_{o_1} + \dots + 1 \times p_{o_n}] \\ \text{so, } T_{avg}(O) - T_{avg}(O') &= n(p_{o_1} - p_{o_i}) + (n-i)(p_{o_i} - p_{o_1}) = i(p_{o_1} - p_{o_i}) > 0 \\ \text{given that } a_{o_i} \text{ has the smallest completion time, i.e., } p_{o_1} &> p_{o_i} \end{aligned}$$

This leads to a contradiction with the assumption that  $O$  is the optimal ordering. Therefore, we have proved that  $a_{o_1}$  must be the task with the smallest completion time, meaning that the index of first task ( $o_1$ ) in optimal solutions always agree with the index of first task ( $s_1$ ) in the greedy algorithm above.

Property 2:

Once we selected the first task, we need to schedule the remaining tasks so as to minimize their average completion time. This is a subproblem of the original one, and we can again use the above steps to prove that every time when we need to select a task to be the  $i$ th one to run, we need to pick the one with the smallest completion time among the remaining tasks, in order to achieve the optimal ordering eventually.

Therefore, we have proved the optimality of our greedy algorithm.

The running time of this algorithm will simply be the time of sorting the tasks, which will be  $O(n \log n)$  if we use merge sort.

b.

We will use greedy algorithm here.

```
SCHEDULE(S)
    sort S in ascending order by releasing time
    Initialize a list T to store distinct releasing times
    Run RELEASE-TIME(S, T)
    Initialize a Min-Priority Queue Q
    set i <- 1
    for each  $t_j$  in T:
        while  $r_i \leq t_j$ :
            insert  $a_i$  into Q based on  $p_i$ 
            set i <- i+1
        if  $t_{j+1}$  is null:
            break
        else:
            set duration <-  $t_{j+1} - t_j$ 
            while duration > 0:
                extract min  $a_k$  from Q
                if  $p_k < duration$ :
                    run and finish  $a_k$ 
                    set duration <- duration -  $p_k$ 
                else:
                    run  $a_k$  for duration time
                    set  $p_k <- p_k - duration$ 
                    set duration <- 0
                insert  $a_k$  into Q based on  $p_k$ 
            while Q is not empty:
                extract min  $a_k$  from Q and run

RELEASE-TIME(S, T)
    for  $a_i$  in S:
```

```

if i = 1 or ri != T[-1]
append ri to T

```

Same as part a, this problem also exhibits the greedy property which can be exploited by running activities in “shortest remaining processing time” order, since now we involve the preemption. At the beginning, “shortest remaining processing time” equal to  $p_i$  for any  $a_i$ . Use a priority queue which prioritizes based on the activity with the shortest remaining time. Then we choose from the queue and do the shortest activity until finish all of them. Each time when a new activity comes up, insert it into the queue. Also, preempt the current activity, recalculate its remaining time, and put it back into the min queue. Then we again choose from the queue and do the shortest activity.

The proof of optimization is the same as part a: at each release time, we meet a same subproblem as part a, while now activities = currently available activities, completion time = remaining completion time. If we do not schedule using the greedy algorithm based on remaining processing time, then we will be able to swap two time slots which would then improve the sum of the completion times, and leads to a contradiction. For example, assume you have two activities at time, where activity  $i$  has processing time  $x$  remaining and  $j$  has  $y$ , where  $x > y$ . Assume for the purposes of contradiction that the optimal answer has activity  $j$  running before activity  $i$ . then  $c_j = a + y$  and  $c_i = a + y + b + x$ . The average completion time is  $O = \frac{2a+2y+b+x}{2}$ . However if activity  $i$  running before activity  $j$ , the average completion time is  $O' = \frac{2a+2x+b+y}{2} < O$ , which leads to the contradiction. As a result, the activity with the lowest time remaining should be done first.

We have already proved that at each step, choosing the one with the smallest remaining completion time will lead to the optimal solution, thus we have proved that our above algorithm can achieve the overall optimization.

Since at each release time we may need to preempt the current activity and insert it into the Min Queue with its new remaining completion time, which has the worst case  $O(n)$  running time. In the worst case when all the activities have different releasing time, so we need to do  $n$  insertions and the running time will be  $O(n^2)$ . So the overall time complexity is  $O(n \log n)$  (for sorting) +  $O(n^2)$  (for inserting) +  $O(n)$  (for extracting min  $a_i$ ) =  $O(n^2)$

### Problem 3

1.

i.

Consider the following activities:

$$\begin{aligned}a_1 &= \{s_1 = 1, f_1 = 4, w_1 = 2\}, \\a_2 &= \{s_2 = 1, f_2 = 2, w_2 = 1\}, \\a_3 &= \{s_3 = 2, f_3 = 3, w_3 = 1\}, \\a_4 &= \{s_4 = 3, f_4 = 4, w_4 = 1\}\end{aligned}$$

Using “Select an activity of largest weight” we will get  $\{a_1\}$ , and the total weight is  $w_1 = 2$ . However, the optimal solution is  $\{a_2, a_3, a_4\}$  with the total weight being  $w_2 + w_3 + w_4 = 3$ .

ii.

Consider the following activities:

$$\begin{aligned}a_1 &= \{s_1 = 1, f_1 = 3, w_1 = 1\}, \\a_2 &= \{s_2 = 3, f_2 = 5, w_2 = 1\}, \\a_3 &= \{s_3 = 2, f_3 = 4, w_3 = 2\}, \\a_4 &= \{s_4 = 4, f_4 = 6, w_4 = 2\}\end{aligned}$$

Using “Select an activity with the earliest finishing time” we will get  $\{a_1, a_2\}$ , and the total weight is  $w_1 + w_2 = 2$ . However, the optimal solution is  $\{a_3, a_4\}$  with the total weight being  $w_3 + w_4 = 4$ .

2.

We will use Bottom-up Dynamic Programming to solve this problem.

We first sort the activities by their finish time. Without the loss of generality, we number the activities and their start/finish time from 1 to  $n$  after they are sorted, i.e., in the following algorithm,  $a_1 \dots a_n$  is not the original random order, but the sorted order;  $a_1$  is the activity that finish first with start time being  $s_1$ , finish time  $f_1$ .

```
SCHEDULE(n)
    Run CAL-OPT(A0, S0, F0, W0)
    // get OPT, W, P
    Initialize an empty list R
    Run FIND(n)
```

```

        return R

CAL-OPT(A0, S0, F0, W0)
    Initialize an empty list OPT with length n+1
    Sort activities by their finish time
    // get A[a1...an], S[s1...sn], F[f1...fn], and W[w1...wn]
    Run LAST-ACT(A)
    // get P
    set OPT[0] <- 0 // init state
    for i = 1 to n
        set j <- LAST-ACT(i)
        set OPT[i] <- max(wi + OPT[j], OPT[i-1])
    output OPT, W, P

LAST-ACT(A)
    Initialize an empty list P with length n
    for i = 1 to n:
        set target <- si
        set P[i] <- B-SEARCH(F, target, 1, i-1)
    output P

B-SEARCH(A[1...k], target, low, high)
    if (high < low)
        return low
    mid = (low + high) / 2
    if (A[mid] > target)
        return B-SEARCH(A, target, low, mid-1)
    else if (A[mid] < target)
        return B-SEARCH(A, target, mid+1, high)
    else
        return mid

FIND(j)
    if j == 0:
        return R
    else if wj + OPT[j] > OPT[j-1]:
        append j to R
        FIND(P(j))
    else:
        FIND(j-1)

```

Justification:

Here,  $P(j)$  denotes the “last compatible activity” of  $a_j$ , i.e., the activity  $a_i$  with the biggest finish time  $f_i \leq s_j$ . We find this using binary search.  $\text{OPT}(j)$  is value of optimal solution to the subproblem consisting of activities requests 1, 2, ...,  $j$ .

So we have two conditions:

Case 1: Optimum selects activities  $j$ :

- so now we can't use incompatible jobs  $\{P(j) + 1, P(j) + 2, \dots, j - 1\}$

- we include  $j$  into the optimal solution to problem consisting of remaining compatible activities  $1, 2, \dots, P(j)$

Case 2: Optimum does not select  $j$ .

- optimum is the same as the optimal solution to problem consisting of remaining compatible jobs  $1, 2, \dots, j - 1$

The relation is:

$$OPT(j) = \begin{cases} 0, & \text{if } j = 0 \\ \max(w_j + OPT(P(j)), OPT(j - 1)), & \text{otherwise} \end{cases}$$

Which is exactly what our algorithm does.

Running time:

The sorting takes  $O(n \log n)$  time if we use merge sort. The LAST-ACT takes  $O(n \log n)$  time since it uses binary search inside a for loop which has  $n$  iterations. The FIND has  $k \leq n$  recursive calls, since in the worst case  $j = j - 1$  in each step and it goes through  $n$  all the way down to 1 to find the solution set (e.g., in the case when no activity overlaps with any other one). So the overall run time is  $O(n \log n) + O(n \log n) + O(n) = O(n \log n)$ .

## Problem 4

a.

```

LPS(S)
    Initialize a table OPT with n × n size
    for i from 1 to n:
        set OPT[i][i] <- 1
    for len from 2 to n:
        for i from 1 to n-l+1:
            set j <- i+len-1
            if X[i] == X[j]:
                if len == 2:
                    set OPT[i][j] <- 2
                else:
                    set OPT[i][j] <- 2+OPT[i+1][j-1]
                else:
                    set OPT[i][j] <- max(OPT[i+1][j], OPT[i][j-1])
    Run FIND-LPS(S, 0, n, OPT) // return list LPS
    Output LPS

FIND-LPS(X, l, r, T)
    if r < l:
        return []
    else if r = l:
        return [X[l]]
    else if X[l] = Y[r]:
        return X[l] + FIND-LPS(X, l+1, r-1, T) + X[l]
    else if T[l+1][r] > T[l][r-1]:
        return FIND-LPS(X, Y, l+1, r, T)
    else
        return FIND-LPS(X, Y, l, r-1, T)

```

For an input string  $S = s_1s_2\dots s_n$ , let  $O(i, j)$  denotes the length of the longest palindrome subsequence on the substring  $s_i s_{i+1} \dots s_j$ , and we have the recurrence

$$O(i, j) = \begin{cases} 0, & \text{if } j < i \\ 1, & \text{if } j = i \\ 2 + O(i + 1, j - 1), & \text{if } j > i \text{ and } s_i = s_j \\ \max(O(i + 1, j), O(i, j - 1)), & \text{if } j > i \text{ and } s_i \neq s_j \end{cases}$$

The base cases correspond to all string with length 1, which are palindromes with length 1. If the string is at least length 2, then we check if the first and last characters are equal. If this is true, we take both characters to be part of the palindrome and recursively check the inner substring. Otherwise, we take

the optimal solution of the two subproblems, i.e., the maximum of the LPS of two substrings that ignore the first and last character respectively. Thus, the length of the longest palindromic subsequence of  $S$  is  $O(1, n)$ .

Instead of using recursion, we use dynamic programming, since some subproblems are duplicated. We first make a table  $\text{OPT}$  of size  $n \times n$ , where  $\text{OPT}[i][j]$  stores the value of  $O(i, j)$ . We initialize all  $\text{OPT}[i][i]$  with 1. Then we start to fill out the table starting from all substring with  $\text{len} = 2, 3, \dots, n$ . Each time, we decide whether the first character  $X[i]$  is the same as the last character  $X[j]$ . If yes, we append the character  $X[i]$  at the end of  $\text{OPT}[i][j]$ ; if not, we just copy  $\text{OPT}[i+1][j-1]$ . In this way, we can obtain the palindrome subsequences of maximum length.

To get the palindromes, we use backtrack.  $\text{FIND-LPS}(S, l, r, \text{OPT})$  will return half of the LPS of  $S_l \dots S_r$ . The base case are 1.  $r < l$ , where we return an empty list to begin with; 2.  $r = l$ , where we return a list that contains only  $S_l$ . Otherwise, we check whether  $S_l = S_r$ . If they are equal, we append this character at both the begin and the end of  $\text{FIND-LPS}(S, l+1, r-1, \text{OPT})$ , which is the list containing LPS of  $S_{l+1} \dots S_{r-1}$ . If they are not equal, we compare  $\text{OPT}[l+1][r]$  and  $\text{OPT}[l][r-1]$  to select from the optimum of two subproblems.

The time complexity of calculating the  $\text{OPT}$  table is

$$\sum_{l=2}^n \sum_{i=1}^{n-l+1} c = O(n^2)$$

and that of finding the LPS is  $O(n)$  since in the worst case, every recursive call we have either  $l + 1$  or  $r - 1$ , meaning that we proceed by 1 character, so we have at most  $n$  recursive calls, each with constant time. So overall, the running time is  $O(n^2)$ .

## Problem 5

In this question, let  $p$  denotes “graph contains a cycle of odd length” and  $q$  denotes “graph is not bipartite”.

a.

Prove that  $p \rightarrow q$ .

We prove this by proof of contradiction: assume that a graph  $G$  contains a cycle of odd length and the graph is bipartite.

If  $G$  is bipartite, let the vertex partitions be  $P_1$  and  $P_2$ . Suppose that  $G$  contains an odd cycle  $C = v_1e_1v_2e_2\dots e_{2k+1}v_1$ . Assume that  $v_1$  is in  $P_1$ ; then we know  $v_2$  must be in  $P_2$ , and it is connected to  $v_1$  by  $e_1$ . So, we can see step by step that  $e_{2i+1}$  is preceded by a vertex in  $P_1$  and proceeded by a vertex in  $P_2$  for all  $i \in [0, k]$ . But  $e_{2k+1}$  is proceeded by  $v_1$ , which is a vertex in  $P_1$  and therefore cannot also be a vertex in  $P_2$ . This leads to a contraction with our assumption. Thus we have proved that if a graph  $G$  contains a cycle of odd length, the graph is not bipartite.

b.

Prove that  $q \rightarrow p$ .

We will prove this by proving its contrapositive,  $\neg p \rightarrow \neg q$ , which is “if a graph  $G$  does not contain a cycle of odd length, it is bipartite”. To prove this, we only need to show a algorithm that can find the partition for any graph  $G$  that does not contain a cycle of odd length.

```
PARTITION(G):
    mark all the vertices in G as -1
    for v in G:
        if v is not marked -1:
            continue to next vertex
        else:
            Initialize a list C to store the protential odd cycle
            if MARK(v, 0, C) == false:
                return false
            output G with marked vertices
    return true

MARK(v, num, C)
    if v is marked as 1-num:
        output C
        return false
    else if v is marked as num:
        return true
    else:
```

```

mark v as num
for all the neighbor s of v:
    MARK(s, 1-num, C.append(v))

```

Justification:

We first mark all nodes as -1, meaning that the node is not been visited. We pick a arbitrary vertex  $v_1$  to begin our algorithm by marking it as 0, meaning that we put it in the partition  $P_1$ . Then, we do DFS to mark all the vertices that are linked one by one using 1 and 0 alternatively, in order to ensure that any two adjacent vertices are not marked with the same number (i.e., they are put into different partition set).

If we encounter a node that has already been marked with the same number as its predecessor, then we know that this current link forms an odd cycle, so we return false and output the odd cycle. If we successfully color all the vertices in  $G$ , then we know there is no odd cycle. We return true and output  $G$  with all the vertices being marked. So therefore we found a partition: all the vertices marked 0 will be in the partition  $P_1$  and all the vertices marked 1 will be in  $P_2$ . Using this algorithm, we have prove  $\neg p \rightarrow \neg q$ . Since we are doing DFS, which was told in lecture, and its time complexity is  $O(e + n)$ , since we go through every node in the for loop and go through every edge in the recursive call MARK, each with constant time.

c.

This is basically the same with b. We can show that by first converting the problem to the scenario of b, and then solve it using the above algorithm. We first initialize a graph  $G$  with vertices  $[x_1..x_n]$ . Then we go through the set of non-equality constraints  $x_i \neq x_j$ , and link  $x_i, x_j$  in  $G$ , i.e., put  $x_j$  into **neighbor**( $x_i$ ) and put  $x_i$  into **neighbor**( $x_j$ ). Then we have finished converting and obtain a graph with  $n$  vertices and  $m$  edges. Now, we just run the algorithm in part b on this graph; if it returns false and outputs an odd cycle, then there is no satisfying assignment, since there is a contradiction. If it returns true and outputs  $G$  with  $[x_1..x_n]$  marked, then we the graph is bipartite and we found a satisfying assignment: all the vertices marked as 0 are in  $P_1$ , and will be assigned 0, and all the vertices marked as 1 are in  $P_2$ , and will be assigned 1.

Justification:

Given the definition of bipartite, we know that any two vertices  $x_i, x_j$  in one partition do not have an edge in-between them. Since edge means non-equality constraint, so there is no non-equality constraint for  $x_i, x_j$ . The running time for converting the problem in a graph is  $O(n + m)$ , since we go through each variable and each constraint once, each in constant time. So the overall running time is  $O(n + m) + O(n + m) = O(n + m)$ .

## Extra Credit

1.

$$\alpha(v) = w(v) + \sum_{u \in \text{children}(v)} \beta(u)$$

$$\beta(v) = \sum_{u \in \text{children}(v)} \max(\alpha(u), \beta(u))$$

Justification:

For every node  $v$ , we have two conditions:

Case 1: Optimum selects node  $v$ :

- so now we can't use incompatible nodes  $u \in \text{children}(v)$ .

- we include  $w(v)$  into the optimal solution to problem consisting of remaining compatible nodes, which is  $\sum_{u \in \text{children}(v)} \beta(u)$ , because  $\beta(u)$  denotes the situation that  $u$  is not selected.

Case 2: Optimum does not select  $v$ .

- optimum is the same as the optimal solution to subproblem consisting of remaining compatible nodes  $u \in \text{children}(v)$ . Notice that even though we do not select  $v$ , we may not select  $u$  as well. We select  $u$  only if  $\alpha(u) > \beta(u)$ .

2.

Algorithm:

```

MIS-TREE(T)
    Root the tree T at a node r
    Initialize two list  $\alpha$  and  $\beta$  each with size n
    for each node v in T (using post-order traversal)
        if v is a leaf node:
            set  $\alpha(v) \leftarrow w(v)$ 
            set  $\beta(v) \leftarrow 0$ 
        else:
            set  $\alpha(v) \leftarrow w(v) + \sum_{u \in \text{children}(v)} \beta(u)$ 
            set  $\beta(v) \leftarrow \sum_{u \in \text{children}(v)} \max(\alpha(u), \beta(u))$ 
    Initialize a list MIS
    Run FIND-MIS(r,  $\alpha$ ,  $\beta$ , true, MIS)
    Output MIS,  $\max(\alpha(1), \beta(r))$ 

    FIND-MIS(v,  $\alpha$ ,  $\beta$ , include, MIS)
    if  $\alpha(v) > \beta(v)$  and include = true:
        append v to MIS
        set include  $\leftarrow$  false
    else:
        set include  $\leftarrow$  true

```

```

for u in children(v):
    run FIND-MIS(u,  $\alpha$ ,  $\beta$ , true, MIS)

```

Justification:

The basic logic of calculating the maximum weight of the MIS is explained in part 1. Instead of using recursion, we use dynamic programming, since some subproblems are duplicated. We first make two lists  $\alpha$  and  $\beta$  each with size  $n$ . Since we are using bottom-up DP, to evaluate  $\alpha(v)$  and  $\beta(v)$  we need to have computed values of all children and grandchildren of  $v$ , so we need to do the post-order traversal of this tree. In this way, we can calculate  $\alpha(v)$  and  $\beta(v)$  for all nodes, and  $\max(\alpha(r), \beta(r))$  will be the final answer, where  $r$  is the root node of  $T$ .

To find the MIS itself, we use backtracking. `FIND-MIS(v,  $\alpha$ ,  $\beta$ , include, MIS)` will decide whether to include the current node  $v$  in the optimum MIS: only if  $\alpha(v) > \beta(v)$  and the flag `include` is `true`, we include the node  $v$ , and set the flag to `false`; otherwise, we set it to `true`. Then we call `FIND-MIS` recursively on all the children of  $v$ . Finally we can get all included nodes in MIS.

The time complexity of calculating  $\alpha$  and  $\beta$  is  $O(n)$ , since we use post-order traversal and go through every node once, each in constant time. The time of finding the MIS is  $O(n)$ , too, since we go through every node once from root to leaves, each with constant time. So overall, the running time is  $O(n)$ .