CS170 - Homework # 5

SID #25949105

Problem #1

First name : NINH → NINHNINHNI

Last name : DO → DODODODODO

Using the algorithm in the textbook gives, for the first letter pair:

	N	-	N
	-	D	D
Cost :	1	1	1
Subprob:	INHNINHNI DODODODODO	NINHNINHNI PODODODODO	INHNINHNI ODODODODO

or $E(10,10) = \min\{1+E(9,10),\ 1+E(10,9),\ 1+E(9,9)\}$

In general:

$$E(i,j) = \min\{1+E(i-1,j),\ 1+E(i,j-1),\ 1+E(i-1,j-1)\}$$

We come up with the table:

		N	I	N	H	N	I	N	H	N	I
		1	2	3	4	5	6	7	8	9	10
D	1	1	2	3	4	5	6	7	8	9	10
O	2	2	2	3	4	5	6	7	8	9	10
D	3	3	3	3	4	5	6	7	8	9	10
O	4	4	4	4	4	5	6	7	8	9	10
D	5	5	5	5	5	5	6	7	8	9	10
O	6	6	6	6	6	6	6	7	8	9	10
D	7	7	7	7	7	7	7	7	8	9	10
O	8	8	8	8	8	8	8	8	8	9	10
D	9	9	9	9	9	9	9	9	9	10	10
O	10	10	10	10	10	10	10	10	10	10	10

Since my names don't have any letter matched then any way is the same, say, the diagonal one.

The total cost is 10. (edit distance)

Problem #2 : Weighted Set Cover

Main idea :

Use greedy algorithm : repeatedly select set s_i with minimal relative weight $\tilde{w}_i$, where the relative weight $\tilde{w}_i$ is defined as the ratio of weight w_i and the number of "uncovered" elements in s_i.

Pseudo code :

```
CoverSet = ∅ ; CoverElement = ∅
while not cover all element :
    Compute # of elements in s_i but not in CoverElement → n_i
    Compute relative weight w̃_i = w_i / n_i
    Add s_i w/ minimal relative weight w̃_i
```

Proof of Correctness :

Using the same strategy as in the lecture : Assume at the iteration i, there is at least $n-i$ elements left. This is conservative since we may be able to cover more than one element at a time. The total cost is k, so the cost of element at most $\frac{k}{n-i}$ per element in average, otherwise the total cost will exceed k after cover all $n-i$ element.

Thus, the cost for the whole process is at most

$$\frac{k}{n} + \frac{k}{n-1} + \frac{k}{n-2} + \cdots + \frac{k}{2} + \frac{k}{1} = k\left(\frac{1}{n} + \frac{1}{n-1} + \cdots + 1\right)$$

for $i = 0 \to n-1$. Taylor's expansion gives: $\ln n = 1 + \frac{1}{2} + \frac{1}{3} + \cdots$

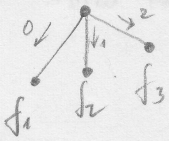
$\Rightarrow$ the sum $< k(\ln n + 1) \sim O(k \log n)$

Problem #3 Ternary Huffman

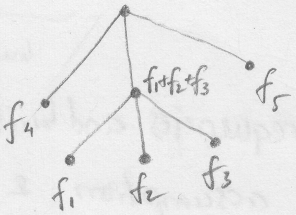
Main idea:

Similar to binary Huffman tree, the ternary one uses greedy algorithm: find the three smallest frequencies f_1, f_2, f_3, make them leaves and add the new frequencies $(f_1+f_2+f_3)$ into the priopridy queue. This frequency is a common parent node of the three old ones. Pop three smallest ones and build the tree up with leaves and node, until the priopridy queue is empty

$[f_1, f_2, f_3, \dots f_n] \rightarrow [(f_1+f_2+f_3), f_4, f_5, \dots, f_n]$



$\rightarrow [(f_1+f_2+f_3+f_4+f_5), f_6, f_7, \dots f_n]$



Pseudo code:

```
Q = [f1, f2, f3, ..., fn]      priority queue
while Q not empty:
    fi = deletemin(Q); fj = deletemin(Q); fk = deletemin(Q);
    create a node l with children fi, fj, fk
    Q.add(fi + fj + fk)
```

Proof of Correctness:

There are two invariants:

1. Each node has 3 children except the root node that may have 2 children.
2. The low frequency has depth $d \leq$ that of the high frequency.

The cost of the tree is given by:

$$\text{Cost} = \sum_{1}^{n} f_i d_i \qquad \text{where } d_i \text{ is depth of } f_i$$

we prove this cost is minimum by using induction:

1. for $n = 3$: $\text{cost} = f_1 d_1 + f_2 d_2 + f_3 d_3$

   with $d_1 = d_2 = d_3 = 1 \rightarrow$ optimal

2. Assume the cost is minimum with $n = k$

   $[f_1, f_2, \cdots, f_k]$

3. for $n = k+1$, we can add an arbitrary new frequency:

   $[f_1, f_2, \cdots, f_k] \rightarrow [\underbrace{f_1, f_2, \cdots f_k}_{\text{build a tree}}, f_{k+1}]$

   $\uparrow$ f_{new}

we can take the first k frequencies and build a tree, this tree is optimal based on the assumption 2. The least frequency f_{k+1} is leaf having depth 1, since it has highest frequency

$$\Rightarrow \text{Cost} = \sum_{1}^{k+1} f_i d_i = \underbrace{\sum_{1}^{k} f_i d_i}_{\substack{\text{optimal} \\ \text{(minimum)}}} + \underbrace{f_{k+1}}_{\text{depth } d=1}$$

Now we see that if we swap f_{k+1} w/ any other leaf, the cost will increase since:

$$f_{k+1} d_i + f_i \geq f_i d_i + f_{k+1} \iff (f_{k+1} - f_i)(d_i - 1) \geq 0$$

thus, cost is optimal.

Problem # 4

Main Idea:

We sort the tasks in the increasing order of execution time and we choose the ones which have smaller time to do first, i.e. in increasing time order.

If both tasks have the same execution time, we choose the one with higher cost. The basic idea of this breaking tie is if we delay the task with hrghes cost, the total penalty will add up.

Proof of Correctness

Tasks : T_i Credit : V_i Cost : P_i Time : R_i $i = \overline{1, N}$

The award of task i is : $V_i - t_i \cdot P_i$, $t_i \geq R_i$

The total award is $\sum_{i=1}^{n} (V_i - t_i P_i)$

The credit does not change so to maximize the award, we have to minimize the delay time associated w/ cost. First, consider short → long task order. The table of penalty having $R_1 \leq R_2 \leq R_3 \ldots \leq R_n$ is as below

(T1)

Day 1 : $P_1 R_1$

Day 2 : $P_2 R_1 + P_2 R_2$

Day 3 : $P_3 R_1 + P_3 R_2 + P_3 R_3$

Day 4 : $P_4 R_1 + P_4 R_2 + P_4 R_3 + P_4 R_4$

...

Day n : $P_n R_1 + P_n R_2 + P_n R_3 + \ldots + P_n R_n$

In term of matrix :

$$\begin{bmatrix} P_1 & & & \\ P_2 & P_2 & & \\ P_3 & P_3 & P_3 & \\ & \ddots & & \\ P_n & P_n & \ldots & P_n \end{bmatrix} \begin{bmatrix} R_1 \\ R_2 \\ R_3 \\ \vdots \\ R_n \end{bmatrix} =$$

Now we try to swap tasks, if the total penalty increases, don't swap, otherwise swap it. We will see that the rule of swaping is as follow :

Day i : $P_i R_1 + P_i R_2 + P_i R_3 + \dots + P_i R_i$

Day j : $P_j R_1 + P_j R_2 + P_j R_3 + \dots + P_j R_j$

After swapping:

Day i : $P_j R_1 + P_j R_2 + P_j R_3 + \dots + P_j R_j$

Day j : $P_i R_1 + P_i R_2 + P_i R_3 + \dots + P_i R_i$

By adding terms vertically, the total penalty changes from: (before swapping)

(I) $R_1 \underbrace{\left(\sum_1^n P_k\right)}_{\text{all } P\text{'s term}} + R_2 \underbrace{\left(\sum_2^n P_k\right)}_{\text{missing } P_1} + \dots + R_i \left(\sum_i^n P_k\right) + \dots + R_j \left(\sum_j^n P_k\right)$

$\left(\sum_i^n P_k\right)$: missing $P_1 \to P_{i-1}$; $\left(\sum_j^n P_k\right)$: missing $P_1 \to P_{j-1}$

to: (after swapping)

(II) $R_1 \left(\sum_1^n P_k\right) + \dots + R_j \underbrace{\left(\sum_i^n P_k\right)}_{\text{missing } P_1 \to P_{i-1}} + \dots + R_i \underbrace{\left(\sum_1^n P_k\right)}_{\text{missing } P_1 \to P_j \text{ have } P_i}$

For example, if we swap task 2 and task 4, the penalty table turns into:

(T2)

Day 1 : $P_1 R_1$

Day 2 : $P_4 R_1 + P_4 R_4$

Day 3 : $P_3 R_1 + P_3 R_4 + P_3 R_3$

Day 4 : $P_2 R_1 + P_2 R_4 + P_2 R_3 + P_2 R_2$

...

Day $n-1$: $P_{n-1} R_1 + P_{n-1} R_4 + P_{n-1} R_3 + P_{n-1} R_2 + P_{n-1} R_5 + \dots$

Day n : $P_n R_1 + P_n R_4 + P_n R_3 + P_n R_2 + P_n R_5 + \dots$

If we add terms vertically, we will see exactly the series (II)
We want to minimize series (I). We see that the factors $(\sum_{k}^{n} P_k)$ are decreasing as it has one term P less than the factor $(\sum_{k}^{n} P_k)$ before and one term P more than the factor $(\sum_{k}^{n} P_k)$ after. So we have to associate factor $(\sum_{k}^{n} P_k)$ with the increasing order of factor R_i from left to right, and this is also the order of executing tasks: favor shortest time tasks.

Quantitatively, take table $(T_2) - (T_1)$:

$$ET_{2\to4} = \underbrace{R_4\left(\sum_{2}^{3} P_k\right)}_{\text{col 2 of T2 - col4 of T1}} + \underbrace{R_3(P_4 - P_2)}_{\text{col3 of T2 - col3 of T1}} - \underbrace{R_2\left(\sum_{3}^{4} P_k\right)}_{\text{col 4 of T2 - col2 of T1}} \quad \left(\begin{matrix}\text{extra time}\\ \text{after swapping}\end{matrix}\right)$$

We prove the above Extra-time is always positive given $R_2 \leq R_3 \leq R_4$

Two cases:

Case 1: If $P_4 > P_2$

$$\Rightarrow ET \geq R_4\left(\sum_{2}^{3} P_k\right) + R_2(P_4 - P_2) - R_2\left(\sum_{3}^{4} P_k\right)$$
$$\geq R_4(P_2 + P_3) - R_2(P_2 + P_3)$$
$$\geq (R_4 - R_2)(P_2 + P_3) \geq 0$$

Case 1: If $P_4 < P_2$

$$\Rightarrow ET \geq R_4\left(\sum_{2}^{3} P_k\right) + R_4(P_4 - P_2) - P_2\left(\sum_{3}^{4} P_k\right)$$
$$\geq R_4(P_4 + P_3) - P_2(P_4 + P_3)$$
$$\geq (R_4 - R_2)(P_4 + P_3) \geq 0$$

So swapping task 2 & task 4 increasing the total penalty.

Similary for i & j swapping, the ET is a little bit more complicated but pretty much the same:

$$ET_{i \leftrightarrow j} = R_j \left(\sum_{i}^{j-1} P_k \right) + R_{i+1} (R_j - R_i) + \dots + R_{j-1} (R_j - R_i) - R_i \left(\sum_{i+1}^{j} P_k \right)$$

We can consider two case to get either

$$ET_{i \leftrightarrow j} \geq (R_j - R_i) \left(\sum_{i}^{j-1} P_k \right) \geq 0$$

$$\text{or} \quad (R_j - R_i) \left(\sum_{i+1}^{j} P_k \right) \geq 0$$

This confirm that the task ordered in increasing execution time is optimal.

In the case we have two task with the same time, their swap results in (say, swap Task 3 & task 4 w/ $\begin{cases} R_3 = R_4 \\ P_3 > P_4 \end{cases}$)

$$ET_{4 \leftrightarrow 3} = R_4 P_3 - R_3 P_4 = R_4 (P_3 - P_4)$$

so we should take T_3 over T_4 so that their swap has $ET > 0$.

Final case: if some task are incontinuous, we see that it is even the worst case, since if we execute task i some days than switch to task j, then go back to task i. the ET is more than the ET of swapping $i \leftrightarrow j$ as job i is delayed R_j day more adding a little bit extra task j delay compared to the $i \leftrightarrow j$ swapping case.

Problem #5 Finding Maul

Main Idea:

Assume we have both matrix & adjacency list representations of the graph, we first loop through the vertices to find the rank of each vertex, where rank(S) is defined as the # of edges the vertex S has.
Put (S: rank(S)) into a dictionary or hash map HM if rank(S) ≥ 20 and put (S': rank(S')) into a set SS/queue if rank(S') < 20.
Go through each vertex S' of SS to update the rank of vertex S in HM, that is, reduce the rank of a vertex S by 1 if it shares an edge with vertex S' in SS, and move S to the set if rank(S) drops below 20.
The HM of crime-lords consists of those vertices whose rank ≥ 20.

Pseudocode:

```
1. for i = 1 → n :
       r(Si) = # edges of Si
       if r(Si) ≥ 20 : add (Si : r(Si)) to hashmap HM
       else: add (Si : r(Si)) to set SS
2. while ( SS is not empty)
       V = SS.pop()
       for each vi is neigbor of V :
           rank(vi) -= 1
           if rank (vi) < 20 :
               move vi into S
return all Si with r(Si) >= 20 (or return HM)
```

Proof of Correctness:

Set SS consists of those vertices whose are not crime-lords for sure.

Hashmap HM consists of those vertices who are potentially crime-lords.

When we go through V in SS, then going through neighbors of V, we disregard the edges representing "trivial" relationships by reducing rank(S)$_{S \in HM}$ by 1, where we define "trivial" relationship as relationship/edge having at least one non-crime lord vertex.

Indeed, assume there is any "trivial" rls of S in HM that we do not disregard. This rls/edge connect S-S' where S'$\in$ SS. We must have gone through S' already since we loop until SS is empty, so we must have reduced rank(S) by 1 due to this "trivial" rls.

Thus rank(S) represents the # of crime-lord relationship and in HM left only vertices whose rank is >=20, otherwise they has been moved to the set SS and been processed.

$\Rightarrow$ HM is the collection of crime-lords.

Run time:

The for loop has runtime O(|V|)

The while loop has runtime O(|E|) at worst since we just care about the relationship/edge.

This is based on the constant time to access vertices and update their rank using hashmap HM.

$\Rightarrow$ O(|V| + |E|)