Aashish Yadavally
Spring 2021

Homework 3

Question 1

First, let us assume that the graph has two distinct minimum spanning trees T_1 and T_2.

Let there be an edge from vertex u to vertex v such that it is present in T_1 and not present in T_2. Removing these edge splits T_1 into two components, namely C_u, C_v, each containing the vertices u and v respectively.

Let (x, y) be a light edge crossing the cut (C_u, C_v). If $(x, y) \neq (u, v)$, then edge weight of (x, y) is less than that of (u, v) and the spanning tree $T_1 - (u, v) \cup (x, y)$ has a better cost.

This can not be true! Hence, (u, v) is the unique light edge crossing the cut (C_u, C_v) that doesn't belong to T_1.

Now, consider a path p from u to v in T_2 such that it starts in T_u and ends in T_v. Hence, there should be atleast one edge crossing the cut (T_u, T_v) such that it's edge weight is greater than that of (u, v). If we add (u, v) to T_2, we get a cycle composed of (u, v) and the path p. By removing any edge from the cycle, we again get a minimum spanning tree. Hence, $T_2 \cup (u, v) - edge$ is a spanning tree, and with the above definition, has a lower cost than T_2. However, this is a contradiction as well.

Thus, by proof of contradiction, we can prove that for every such cut, there is a *unique* minimum spanning tree.

Counter-example to the converse:

Let us consider a graph with three vertices t, u, v such that the edge weights of (t, u), (t, v) and (u, v) are 1, 1, 2 respectively. This graph has an MST containing the edges (t, u) and (u, v). However, the cut t to (u, v) doesn't have a unique light edge crossing it.

Question 2

Let us create a weighted, undirected graph with $N + 1$ nodes where one is a source node and the remaining N nodes each represents a house.

Between any two houses i and j in the city, add edges with edge-weight C_{ij} representing the cost to build the path from house i to house j.

Add an edge between the source node and a vertex i with an edge-weight W_i representing the cost to build a well at house i.

Find a minimum spanning tree T of this graph. The weight of the minimum spanning tree T for the graph thus constructed gives the minimum amount of money needed to supply every house with water.

Homework 3

Question 3

Consider a simple graph G which has four vertices A, B, C, D and the edges {(A, B), (B, C), (C, D), (D, A)} with the edge-weights 1, 1, 1, 2 respectively. Let us partition G into subsets G_1 and G_2 containing the edges {A, D} and {B, C} respectively.

The MST of G_1 has a weight of 2 and that of G_2 has a weight of 1. The minimum edge weight crossing the cut (A, B) has an edge weight 1. The spanning tree formed by the proposed algorithm is D-A-B-C with a weight of 4. However, the weight of MST of the graph G is 3. Thus, the proposed algorithm fails to obtain an MST.

Hence, based on the above counter-example, I can claim that the algorithm **fails**.

Question 4

(a) A_1, A_2, A_3, A_4 have 73, 44, 100, 55 elements respectively.

Merging A_2 and A_4 to get list A_{24} with 99 elements, worst case for the number of comparisons = 44 + 55 - 1 = 98

Merging A_1 and A_{24} to get list A_{124} with 172 elements, worst case for the number of comparisons = 73 + 99 - 1 = 171

Merging A_3 and A_{124} to get list A_{3124} with 272 elements, worst case for the number of comparisons = 100 + 172 - 1 = 271

Total worst case number of comparisons = 98 + 171 + 271 = 540

(b) First, we need to select two sorted lists with the smallest number of elements.

Let us assume that the sizes of the m lists are stored in a min-heap. The time to construct the heap is $O(m)$.

The time to retrieve two smallest elements from this min-heap by popping them is $O(2 * log(m))$.

Finally, the time complexity to merge the two lists corresponding to these sizes is $O(n_1 + n_2)$ and the worst-case number of comparisons = $n_1 + n_2 - 1$, where n_1 and n_2 are the sizes of the retrieved lists.

This is performed until the min-heap is emptied. By following this procedure, it can be ensured that the total number of comparisons while merging m sorted lists is minimum.

Correctness of the algorithm:

Let us consider the lists L_i with sizes n_i for $1 \leq i \leq m$. In the algorithm, instead of selecting two lists L_i and L_j with minimum sizes from among the given lists for merging, let us select L_i and L_k first. The worst-case number of comparisons for merging L_i and L_k to form L_{ik} is $n_i + n_k - 1$ and that for merging L_i and L_j to form L_{ij} is $n_i + n_j - 1$.

From the assumption, $k > j$. Thus, $n_i + n_k - 1 > n_i + n_j - 1$. Let the worst-case number of comparisons for merging the remaining $m-2$ lists by selecting the minimum of those lists to form L_p be p. Therefore, finally, the worst-case number of comparisons to merge L_p with L_ij is less than that to merge with L_ik as $|L_p + n_i + n_j - 1| < |L_p + n_i + n_k - 1|$.

Therefore, by selecting the two lists with minimum sizes at each step, we can conclude that we will end up with minimum worst-case number of comparisons.

Question 5

Let us consider a code which is not monotonically increasing with respect to the codeword length.

Now, let us rearrange the codewords such that they are monotonically increasing such that the one which is the shortest is assigned to the alphabet with the highest frequency. Repeat this until each codeword is assigned to an alphabet. Since all the codewords are being assigned optimally, the total cost is lowered. This is true for any possible given code.

Thus, it can be concluded that if we order the characters in an alphabet so that their frequencies are monotonically decreasing, then there exists an optimal code whose codeword lengths are monotonically increasing.

Question 6

(a) It is given that the cost of retrieving a file is the sum of the sizes of the files scanned plus the size of the file retrieved.

Thus to retrieve a file f_{ik}, cost $= \sum_{j=1}^{j} m_{ij}$.

Therefore, total cost $= \sum_{k=1}^{n} f_{ik} = \sum_{k=1}^{n} (n - k + 1) m_{ik}$.

(b) Any permutation in non-decreasing order of $m_i's$ will have minimum cost. Let us consider a greedy algorithm which greedily picks the file f_i in non-decreasing order of their size m_i. Thus, by ordering the files on the tape in this order, the total cost will be minimize.

Correctness of the algorithm:

To prove the correctness of the above-mentioned greedy algorithm, let us consider any other file ordering in which the memory sizes of the files are not in non-decreasing order.

We know that the cost to retrieve a file f_{ik} is $\sum_{j=1}^{j} m_{ij}$.

Let us consider two files f_{ik} and f_{il} with memory sizes m_ik and m_il respectively where m_ik ¿ m_il (i.e. not conforming to a non-decreasing order), and $k < l$. If we swap the files f_{ik} and f_{il} in the ordering i, we notice that the cost to retrieve file f_ik in the new

ordering is decreased by $m_i l$. Thus, the cost of retrieving the file is minimized, hence proving that ordering the files in non-decreasing order of their memory sizes greedily decreases the total cost.

Question 7

For a graph G_i with vertex set V and edge set $E_i = \{e_1, ..., e_i\}$, c_i is the number of connected components in the graph.

Initially put each vertex v_i into a bucket i and set $c_0 = n$. Thus, there are n buckets in all.

For $j = 1, 2, ..., m$ suppose edge $e_j = (u, v)$. If u and v are in two different buckets, then set $c_j = c_{j-1} - 1$ and merge these two buckets into one. Otherwise, set c_j to be c_{j-1}.

In the above algorithm, finding the bucket containing the vertex for each of the m edges takes $O(m)$ time. Also, for each of the n vertices, the pointer to the beginning of the bucket is updated at most $logn$ times. Thus, the total cost of all the merges is $O(nlogn)$. Hence, the running time of the algorithm is $O(m + nlogn)$.

Question 8

Sort all the activities according to the ending time f_i in a non-decreasing order: $f_1 \leq f_2 \leq .. \leq f_n$. Let the lecture halls be $H_1, H_2, ...$ Now, we can make a schedule with the following greedy algorithm:

```
for i = 1 to u do begin
    ok <- 0; j<- 1; m<- 1;
    while ok = 0 do
        if activity Ai can be scheduled in hall Hj
            then assign(i) <- j and ok <- 1
        else
            j <- j + 1
```

Let us assume that the above algorithm used m halls. Consider some activity A_i such that it was the first scheduled activity in lecture hall H_m. Now, A_i was scheduled in H_m because all the other $m - 1$ halls were busy, i.e. at the time A_i is scheduled to begin, there are m activities occuring simultaneously. Under such circumstances, the algorithm IS SUPPOSED to use atleast m halls. Thus, the given algorithm is optimal.

The total time complexity of the algorithm is dominated by that taken to sort the activities based on their finish time. Thus, the time complexity of the algorithm is $O(nlgn)$.