COL351 Assignment 1

September 2021

2019CS10363 and 2019CS10355

1 Minimum Spanning Tree

(a) G is and Edge-weighted graph with n vertices and m edges and all the edge-weights in G are distinct.
We have to Prove that G has a unique MST.

Proof: By Contradiction. Let's assume the G does not have a Unique MST. Suppose there are two different MST's M_1 and M_2 which are differing by at least one edge-weight.
Let's assume the edge-weights of M_1 are $e_1, e_2, \ldots, e_{n-1}$ and that of M_2 are $E_1, E_2, \ldots, E_{n-1}$ and these edges are in increasing order. Let's assume that they differ at point x initially where $x <= n$ and
$\forall\ i \in [1, x-1]\ , e_i = E_i$ and assume $E_x > e_x$. Now take another graph $M_3 = M_2 \cup \{e_x\}$. Now as adding any edge in MST will form a cycle because by definition MST is the maximal connected *acyclic* graph so M_3 will definitely contain a cycle. *i.e.* M_3 is *acyclic*

Now we have to show at least one edge in the cycle of M_3 is greater than the edge weight of e_x. Lets assume there is no edge with edge-weight greater than e_x in the cycle of M_3 which implies that the cycle is formed by edges $\{E_1, E_2, \ldots, E_{x-1}\} \cup \{\mathrm{e_x}\}$ as only these edges have weights smaller than the edge weight of e_x . But here is a catch, as we initially assumed that M_1 is MST but if $E_1, E_2, \ldots, E_{x-1}, e_x$ forms a cycle then M_1 cannot be a MST as $E_1, E_2, \ldots, E_{x-1}, e_x$ are same as $e_1, e_2, \ldots, e_x$ because e_x was the first point of difference. So there must be an edge with edge weight greater than e_x and let assume it to be E_g, So now we have a graph $M_4 = M_3 \setminus \{E_g\}$ and it is *acyclic* as the only cycle is removed by removing edge and it has n vertices and same $n-1$ edges.

So it is Spanning Tree with total

$$cost(M_4) = cost(M_2) - wt(E_g) + wt(e_x)$$

and as

$$wt(E_g) > wt(e_x) \text{ so } cost(M_4) < cost(M_2)$$

and it is spanning tree also.

This contradicts our assumption that M_2 was the MST of G. Hence our assumption was wrong. This graph G with all distinct edges have a unique MST

1.b) Given, Graph $G = \{n, n+8\}$ i.e. the graph is an edge weighted graph with n vertices and $n+8$ edges.
We have to design an $O(n)$ running time algorithm to find it's MST.

We know that MST is the maximal connected *acyclic* tree in a graph. So, to find MST of a graph we have to find all the cycles in the graph and remove at least one edge from them and it should be the largest edge in that cycle because we are trying to find MST and want to minimize the total edge-weight sum. Hence, we will do **DFS** traversal in the graph and every time we found a cycle we will remove the largest edge from that cycle and at last our MST should have n vertices and n-1 edges in total. $TimeComplexityAnalysis$: as DFS traversal of any graph has time complexity $O(m+n)$ where m are edges and n are vertices. But here our number of edges are $n+8$ i.e. $O(n)$ so, final time complexity is $O(cn+b)$ where c and b are constants and $c > 0$ hence, it is $O(n)$ i.e. linear time complexity algorithm. Actually this c is 9.

Proof: Now, we have to show that this algorithm terminates finally and when it terminates it will definitely give us an MST of our initial graph G. This will terminate because DFS traversal in a finite graph terminates after visiting all vertices as it does not visits visited vertices again and again and our process of removing largest edge from any cycle will terminate after $m-n+1$ iterations because we are only removing edges whenever we find a cycle and each cycle will increase the number of edges by 1 in MST while number of vertices remain same . So, number of cycles that can be found at max are $m-n+1$ which is basically $n+8-n+1 = 9$ iterations. Now, we have to show that our algorithm on terminating will give a MST which can be shown by the fact that our algorithm will terminate only when there is no cycle in the graph as we are removing maximum weighted edge from every cycle of the graph and this will occur when remaining edges $= n-1$ where n are the vertices and these will remain same because we are not

removing vertices. So, the final graph will remain connected.

And we know by definition that any connected sub graph of any graph G with n vertices and $n-1$ edges is a spanning tree of G. Let's say this sub graph we obtained is G' and $G' = \{v, e\}$ where $|v| = n$ and $|e| = n-1$. Now we have to show that G' is the MST of our initial graph G. Assume It is not MST of our graph and there is another spanning tree in G i.e (G'' with smaller total weight than G'. We know that there will be a unique MST of G from above problem because the graph has all distinct edges. Then, there must be an edge in G'' which is smaller in weight than at least one edge in G' because $weight(G'') < weight(G')$. let that edge be e_{g1} in G'' and $weight(e_{g2}) < weight(e_{g1})$ where e_{g1} is an edge in G'. Now we can clearly see that $G'' \cup \{e_{g2}\}$ will contains a cycle because G'' was the maximum *acyclic* graph. But this edge e_{g2} is not present in G' which we got by our algorithm i.e. it was removed from the cycle but it is not possible as our algorithm only removes maximum weight edge from any cycle and there is at least e_{g1} in that cycle with more weight than e_{g2} and without that G'' is not possible. Hence, under no circumstances our spanning tree obtained by algorithm i.e. G' can be reduced further to form any other spanning tree with less weight than G'.
This concludes the correctness of our algorithm and shows that this linear algorithm will terminate and will finally provide a MST of the given graph G.

b) Algorithm

G = {V,E} Parent[] = -1 *$|V|$
Vis[] = 0 *$|V|$
Function dfs(v, G, parent)
Vis[v] = true;
For u in adjacency list of G
If !vis[u]
Parent[u] = v;
Dfs(u,G,parent)
Else
OUREDGE = Null
Maxweight = wt(u,parent[u])
While((newpar =parent[u]) !=v) do
Maxweight = max(wt(parent[newpar],newpar),maxweight);
If(maxweight $\equiv wt(parent[newpar], newpar))\{OUREDGE = (parent[newpar], neu$
$Newpar = parent[newpar]$
$G = (V, E \setminus \{OUREDGE\}$
$ReturnG$;
End

Function finalMST(G)
While(connected(G) and —V—-1¡—E—¡—V—+8){ v= null
For (vertex c : V)
If (vis[c] = false)
v = c
If(cycle in DFS(v))
G = DFS(v, G,parent)

Return final G

2) Huffman Encoding

2.a) Given n letters with their frequencies equal to the first n $fibonacci$ numbers. The optimal binary Huffman encoding for these n letters.

Initial Cases : If $n = 1$, in this case there is only 1 possible encoding i.e. 0 or 1 for the only letter with frequency 1. So, length of optimal Huffman encoding is 1. If $n = 2$, in this case also we give encoding 0 and 1 to both of these letters and so, length of the optimal huffman encoding for both of these letters with frequency 1 is 1.
Applying huffman encoding algorithm to n letters with frequency {1,1,2,3,5,8, ... } i.e. frequency of n^{th} letter is n^{th} $fibonacci$ number. We are going to add all the n letters in a priority Queue and every time if possible,we are going to remove two smallest frequency letters and will add a new element with frequency sum of both these. This process will terminate after $n-2$ iterations if there are total n letters because after that only 2 element will be remaining in the Priority Queue.
Statement: After $m(m <= n-2)$ iterations of removing two smallest frequency elements and adding a new frequency element with frequency the two smallest elements in priority Queue are $(m+2)^{th}$ $Fibonacci$ number and $(m+3)^{th}$ $fibonacci$ number - 1. Let's denote i^{th} $fibonacci$ number by $f(i)$. Then the two least frequency elements are $f(m+2)$ and $f(m+3)$ -1.
Base case: In 1^{st} iteration (possible only when $n >= 2$) the two smallest frequencies are 1,1 of 1st and 2^{nd} letter respectively. In next step pop both of these from priority Queue and push (1+1) = 2 in it. Next, (possible only $n >= 3$) we have two smallest frequencies as 2 and 2 that are basically $f(3)$ and $f(4)-1$. Inductive hypothesis : After a $(a <= n-2)$ steps of popping and pushing assume the two smallest frequency elements are $f(a+2)$ and $f(a+3)-1$. Inductive step: In the $(a+1)^{th}$ step we again remove two smallest frequency elements and add a new frequency element with frequency the addition of above two elements. As we have already used all $fibonacci$ number frequency elements till $a+1$ i.e. in previous a iterations we have used two least frequency elements every time so, at a total we have used all the numbers $f(i)$ where $1 <= i <= a+1$.

By our induction hypothesis, two smallest frequencies after m iterations are completed are $f(a+2)$ and $f(a+3)-1$. So, now in the next i.e. $(a+1)^{th}$ step we pop out both these and add a new frequency

vector with frequency $(f(a+2)+f(a+3)-1) = f(a+4)-1$. Now all the remaining frequencies are $f(a+3), f(a+4), \ldots, f(n)$ and the new added one i.e. $f(a+4)-1$. On ordering these frequencies in increasing order we get the frequencies are $f(a+3), f(a+4)-1, f(a+4), \ldots, f(n)$. $(f(a+4)-1 > f(a+3)$ because $f(a+2) > 1$ and here $a >= 0$. Hence, we can conclude that after $(a+1)^{th}$ iteration are $f(a+3), f(a+4)-1$ which we have proved using the principle of mathematical induction.

Statement: In the optimal binary tree encoding of $n(n >= 2)$ letters with frequencies the corresponding *fibonacci* numbers the length of optimal coding of two letters with frequencies 1 is n. So, the answer of the question asked about optimal encoding is $n-1$. Proof: let $L(k)$ denote the length of the encoding corresponding to the two smallest frequency elements with frequency 1 after k iterations of the above algorithm.
We know that the above algorithm will stop only after $n-2$ iterations and after $(n-2)^{th}$ iteration the last two remaining frequency elements are assigned '0' or '1' i.e. encoding of length 1. This implies $L(n-2) = 1$ and also at every step we are removing two elements and are adding 1 element so, its length of encoding is decreasing by 1 every time (as we have to form an optimal encoding and adding the added frequency element at same depth will unnecessarily increase the length of optimal coding so we will add it at the place of parent of two least frequency elements and this will decrease the length of the new encoding by 1).
So, basically $L(k) = L(k+1)+1$. So, from both $L(k) = L(k+1)+1$ and $L(n-2) = 1$ we can easily conclude that $L(0) = n-1$. i.e. the length of the encoding of two least frequency elements with frequency 1 is $n-1$.

2.b) We want to compress a file with 16-bit characters such that maximum character frequency is strictly less than twice the minimum character frequency. To show that compression obtained by huffman encoding is same as ordinary fixed-length encoding. This problem is independent of number of bits. Hence, we will prove it for any number of bits let's say n.

Statement: If we want to compress a file with n bit characters with frequencies let's assume $a_1 <= a_2 <= \ldots <= a_{2^n}$ where the maximum character frequency is less than twice the minimum character frequency, then the size of encoding obtained Huffman coding is same as ordinary fixed length encoding.
Proof by induction:
Base case: If $n = 1$, then only two possibilities of encoding the characters with 0 and 1. Hence, length of encoding obtained by Huffman encoding is 1.
Induction Hypothesis: Let's assume the claim is true for all k less than m i.e. $k >= 1$ and$k < n$. **Inductive step**: now our file is having m-bit characters and hence 2^n characters with frequency let's assume $a_1 <= a_2 <= a_3 <= \ldots a_{2^n}$. And we also know that $a_{2^n} < 2 * a_1$ and if this is the case then we can say that for any i, j, k in 1 to $2^n a_i + a_j > a_k$ because $a_i + a_j >= 2 * a_1$ as well as $2 * a_1 > a_k$ for any k in 1 to m.

Hence in our process of Huffman encoding where we remove two minimum elements from a priority Queue and add a new frequency element with frequency having sum of both frequencies if minimum elements one thing is clear that first of all, all the elements from a_1 to a_{2^n} will be popped in first $(2^n)/2 = 2^(n - 1)$ iterations. And after that we are left with frequency elements $a_1 + a_2 <= a_3 + a_4 <= \ldots <= a_{(2^n)-1} + a_{2^n}$. Now this can also be reduced to similar smaller problem as $a_{(2^n-1)} + a_{2^n} < 2 * (a_1 + a_2)$ as we know that $a_{(2^n-1)} < a_{2^n} < 2 * a_1$ and $a_{2^n} < 2 * (a_1) < 2 * (a_2)$, hence adding both these will give us us

that $a_{(2^n-1)} + a_{2^n} < 2 * (a_1 + a_2)$. Hence after 2^{n-1} iterations the priority Queue will be changed to new $n-1$ bit character with 2^{n-1} possible character bit frequencies be $b_1 <= b_2 \ldots <= b_{2^n-1}$ where we can say that $b_i = a_{2i-1} + a_{2i}$

Hence by our hypothesis, The length of optimal coding of all these elements will be same and equal to ordinary fixed-length encoding and equal to $n-1$. Hence length of optimal coding of characters in huffman encoding is (length of optimal coding in encoding of priority Queue after 2^{n-1} iterations)+1 which is $n - 1 + 1 = n$. Hence, for any file containing n-bit (16-bit) characters, the length of encoding obtained by Huffman encoding and ordinary fixed length encoding are same.

3) Graduation Party of Alice

3.a):Given List of n people,List of pair of people who know each other.
Constraints: It has to be largest subset, each person at least know 5 other people and don't know at least 5 other people.
Formulation : Undirected graph $G = \{vertices, edges\}$
vertex : people
Edge : only when friends
Algorithm approach : In this problem after forming the Undirected graph we are basically iterating the graph and then check if the number of edges on a vertex are greater than 5 and also the number if edges on any vertex are more than $n - 6$ (one is the vertex itself) in both cases we are going to remove that vertex from our graph and form a new subgraph and after completion of one such iteration there may be cases where these constraints may be imposed on some other vertices. So, at max we have to remove all the vertices and for that we have to do this process of removing of edges and updating graph n number of times where n is the total number of friends of Alice.

Time Complexity Analysis: Our algorithm runs in $O(n^3)$ time where n is the number of friends of Alice. Outer for loop runs the process of removing the vertices n number of times and inside that we chose every pair of two random vertices and check if there is an edge in between the vertices or not and then do specific actions. So, overall complexity is $O(n^3)$.

Algorithm Correctness proof: We have to show that this algorithm terminates. For that we say that it may be possible that Alice does not invite any of her friends and for that we have to remove all the friends from the graph and hence in that case our outer for loop will do the job and it will terminate automatically as it is just running n iterations.

Along with this we have to show that the set f people returned by our algorithm follow the constraints and also it is the maximum set if people for that we can say that in one iteration of outer loop we are removing at most one people and there can be possibility that Alice invites no friend. Hence, in that case our outer iteration will run n number of times and will remove all such friends which have degree less than 5 or number of friends not connected more than 5. Hence, all such friends will be removed which don't follow the constraint.

Optimality: To prove that the solution obtained by the algorithm is optimal.
Contradiction : lets assume that there exist some set of friends F" which is maximal and the set of friends obtained by our algorithm is F'. Then we can say that there is at least one such friend let v which is in the set F" but not in F'. Now we know that if v is in optimal set then it implies that the number of people v knows in the party should be $>= 5$ and also number of people v don't know in the party should also be $>= 5$.But, v is not in set F' which implies at some iteration, v was contradicting at least one of the above two constraints. Hence, our algorithm returns maximal subset of friends following the constraints.

Here, our proof of correctness concludes and then we can say that our algorithm will return maximal set of friends of Alice such that number of people any people knows in partly is at least 5 and number of people they don't know is also at least 5 .

Algorithm

3a

Function maximumSubset(noOfFriends, pairs)
Vertices = noOfFriends;
N = $|vertices|$
Edges = [];
Friends =[];
Nonfriends = [];
For pair = (u, v) in pairs do
E = E $\cup$ pair
Friends[u] = Friends[u] +1 and Friends[v] = Friends[v]+1

Graph = {V ,E};
For v in vertices
Nonfriends[v] = n-friends[v] -1
For I =0 to I = $n - 1$ do
For source in vertices do
If friends[source]<5 or nonfriends[source]>5 do
For dest in V do
If (source,dest) in E of Graph do
Friends[dest] = friends[dest]-1
Else do
Nonfriends[dest] = nonfriends[dest]-1
Vertices = vertices\{source}
Friends = friends\ {source}
Nonfriends = nonfriends\{source}
Return vertices;

End function

b) Given that Alice invited her n_0 friends to the party having different ages. So we will be creating a list of her n_0 friends in non-decreasing order of their ages. After that we will then greedily allot them tables (capacity of each table is given 10) so that the difference of maximum age of person in table and minimum age of person is at most 10 years.

Required Time Complexity: A greedy algorithm to solve this problem in $O(n_0)$ time assuming the age of each person is an integer in the range [10, 99]

Approach: As the age range of Alice's friends is given so we can make a List of size 100 for storing the frequency of her friends ages. For doing this, we have to iterate the List of her friends which is of size n_0 so it takes $O(n_0)$ time.
After this we update the list of her friends with help of frequency of ages list. This will take $O(n_0)$ time. So the overall complexity of algorithm is $O(n_0)$
Claim 1: The algorithm terminates in finite steps. As the function MinimumTables has a for loop which runs at most the size of the list which is finite i.e. n_0. So the algorithm terminates when all the friends are visited.
Claim 2: The number of tables returned from the function MinimumTables is minimal.// By Contradiction. Say the number of table T returned by the function is not minimal meaning we can further reduce number of tables as there might be some vacant seats in table. Let T_i be the table having vacant seats and T_{i+1} is table filled with people $p_1, p_2, \ldots, p_k, k <= 10$. Now as T_i has vacant seat. We can try to accommodate p_1 in this vacant seat. But the given condition was that difference of the maximum age of person in table and minimum age is at most 10. So p_1 - minimum ages of people in table T_i i.e.
$p_1 - minimumAge(T_i) > 10$. Hence there is contradiction to the statement of algorithm. So we can say that there is no way of getting less

than T tables which is already minimal.

Algorithm

Procedure :

MinimumTables (ages, n_0)
frequency <= [0]*100
i <= 0
while i < n_0 do
frequency[ages[i]] <= frequency[ages[i]] + 1
i <= i+1

M = [0]*n_0
k <= 0
for j in 10 to 100 do
while frequency[j] > 0 do
M[k] <= j
frequency[j] <= frequency[j] - 1
k <= k+1

i <= 0
while i < n_0 do
st <= i
vac <= 0
while st < n_0 and vac < 10 and M[st] $\leq$ M[i] + 10 do
vac <= vac + 1
st <= st + 1

i = st
T <= T + 1

return T
End Procedure: