CS345: Design and Analysis of Algorithms

Tushar Vatsal(14766), Ankur Kumar(14109)

Theoretical Assignment 2

1 Hierarchical metric

Consider my complete graph G(V,E) where V is the set of all vertices, say
V: $\{a_1, a_2, a_3, ..., a_n\}$
and E is the set of all edges.

Algorithm 1: A Greedy Algorithm

```
Greedy_Tree {G}
if | G | = 2 then
    return a tree with h(root)=d(a_1, a_2) and its children as a_1 and a_2;
else
    Let a_1 and a_2 be two vertices with least weight edge;
    Remove a_1 and a_2 from V and insert a' in V;
    Remove edge (a_1, a_2) from E;
    Remove all edges from a_1 and a_2 and insert smaller of the two edges from any vertex to a_1 and a_2
     into a' in E;
    G' ← (V, E);
    T ← Greedy_Tree(G');
    Replace node a' in T by a tree with h(root) = d(a_1, a_2) and its children as a_1 and a_2;
    return T;
end
```

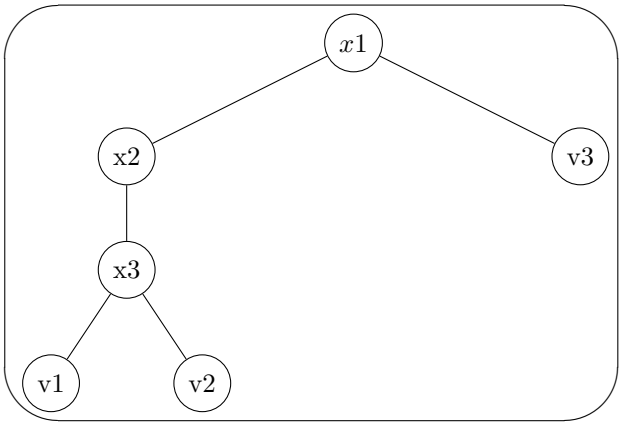
1.1 Details of Greedy Step and smaller instance G'

Search for edge with minimum weight, say(a_1, a_2). Remove a_1 and a_2 from V and insert a' in V where a' is a tree with h(root)=d(a_1, a_2) and its children as a_1 and a_2.
Now say V': $\{a', a_3, a_4, ..., a_n\}$
Also remove edge (a_1, a_2) from E. Now remove the larger of the two edges from a_1 and a_2 to other vertices and assign the smaller edges that vertex a' as one of their two vertices instead of a_1 or a_2. Call this set E'.
Now I am left with a new graph G'=(V',E'). This is again a complete graph of n-1 vertices. Hence this is my smaller instance G' which is a complete graph on n-1 vertices.

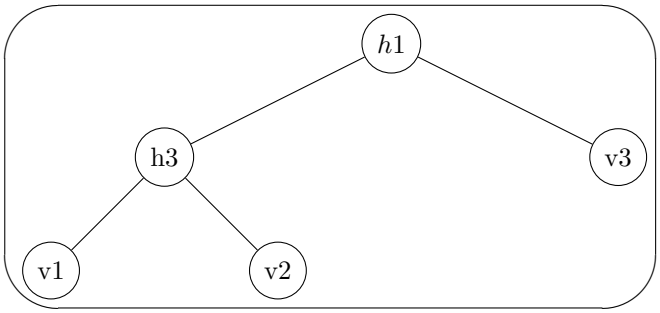
1.2 Theorem and its proof

Lemma 1. *The tree should be full binary tree*

Proof. Suppose the tree is not a full binary tree and is consistent with graph G and is also optimal. Consider such a tree T shown below :



Repalce this with tree T*:

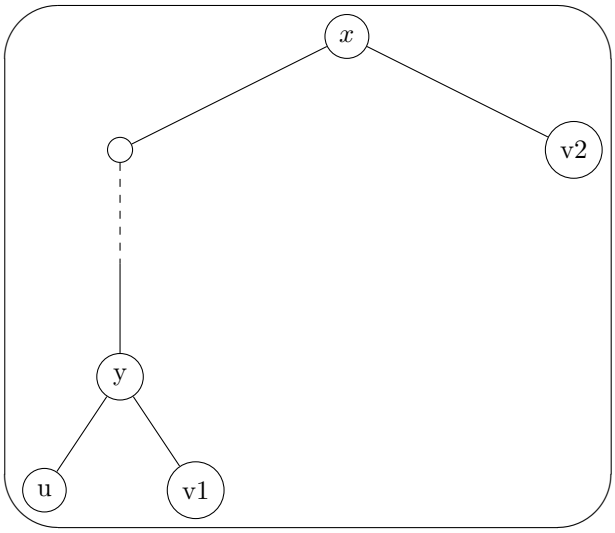


This is a full binary tree and is also consistent and optimal at the same time.

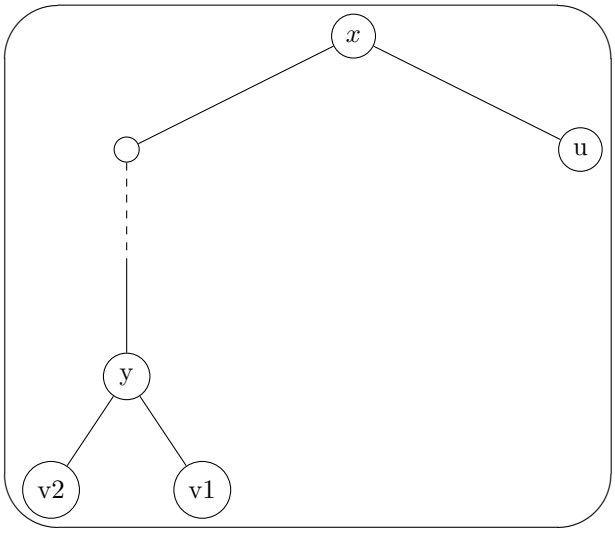
□

Theorem 2. *There exists a tree T* which is consistent with G and also optimal in which vertices a_1 and a_2 appear as siblings and their distance in tree is d(a_1, a_2) where edge (a_1, a_2) is the least weight edge.*

Proof. Consider a tree T(G) which is consistent and optimal and a_1 and a_2 do not appear as siblings.



Here x is the lowest common ancestor of a_1 and a_2
Say $x = \text{LCA}(a_1, a_2)$
also $\text{h}(x) \leq d(\text{a}_1, a_2)$, further $\text{h}(y) \leq d(\text{a}_1, u) \leq d(\text{a}_1, a_2)$
Consider any vertex in subtree of left child of x say p. Distance between p and $a_2 \leq \text{h}(x)$. After replacing u with a_2 and assigning every internal node a value of $\text{h}(x)$ will give me another optimal solution say T*(G) shown below where $\text{h}(v)$ in T(G) $\leq \text{h}(v)$ in T*(G) $\forall\ v \in S$
where S: {set of all internal nodes} $\cup$ {root node}
Thus $\exists$ an optimal and consistent solution where a_1 and a_2 appear as siblings and their distance in tree is $\text{d}(a_1, a_2)$

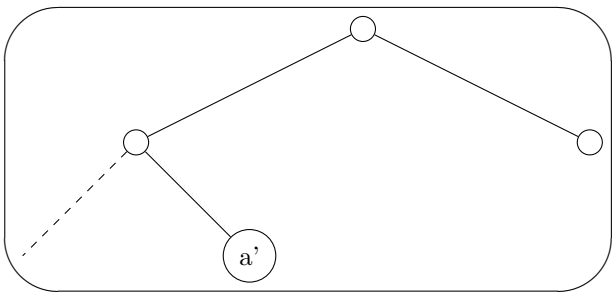


In the above figure $\text{h(y)} = \text{h}(x)$ □

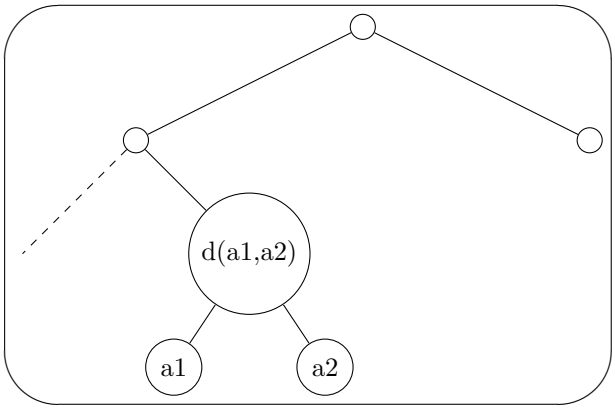
Now define Opt(G) = $\sum_x h(x)$
where S: set of all non-leaf nodes in Greedy_TreeG

Theorem 3. *Opt(G) = Opt(G') + d(a_1, a_2) where terms has its usual meaning as used earlier*

Proof. Derive Greedy_Tree{G} from Greedy_Tree{G'} :
Greedy_Tree{G'} :

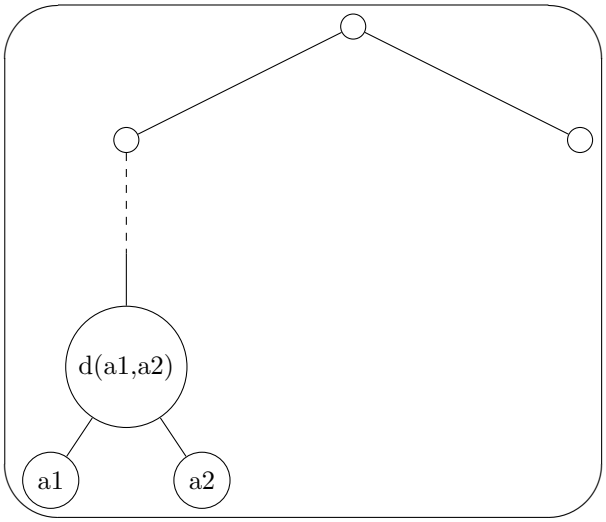


By simple construction we can get Greedy_Tree{G}
Greedy_Tree{G}

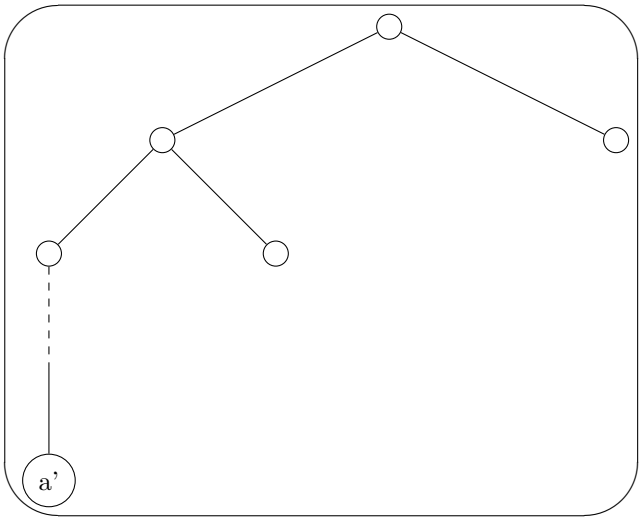


By construction it is obvious that Opt(G) = Opt(G') + d(a_1, a_2)
$\Rightarrow$ Opt(G) $\leq$ Opt(G') + d(a_1, a_2) ———— (i)

To derive Greedy_Tree(G') from Greedy_Tree(G)
Use Theorem 2 that a_1 and a_2 appear as siblings and their distance is d(a_1, a_2)
Let T* be a binary tree corresponding to Greedy_Tree(G) shown below:



Replace the nodes a_1, a_2 and their parent node through $a^{'}$. This will give Greedy_Tree{G'}



with Opt(G') = Opt(G) - d(a_1, a_2)
$\Rightarrow$ Opt(G') $\leq$ Opt(G) - d(a_1, a_2)
$\Rightarrow$ Opt(G) $\geq$ Opt(G') + d(a_1, a_2) ———- (ii)
Using (i) and (ii) we get Opt(G) = Opt(G') + d(a_1, a_2) □

1.3 Implementation of the algo

Initially build an adjacency list for the given graph G. In this adjacency list, linked list corresponding to each vertex should be sorted according to weight of edge in increasing order.
Such an adjacency list can be built in O($n^2 logn$) time.
Suppose any vertex a_i in the adjacency list has x_i neighbours.

$$\Rightarrow \sum_{i=1}^{n} x_i = O(n^2)$$

Now linked list corresponding to each vertex in the adjacency list can be sorted in O($x_i log x_i$) time.
Total time complexity for building this adjacency list: O($\sum_{i=1}^{n} x_i log x_i$)
=O($x_1 log x_1 + x_2 log x_2 + ... + x_n log x_n$)
=O($log(x_1^{x_1} x_2^{x_2} .... x_n^{x_n})$)
$\leq O(log(\sum_{i=1}^{n} x_i)^{\sum_{i=1}^{n} x_i})$
=O($n^2 log(n^2)$)=O($2n^2 logn$)=O($n^2 logn$)
Also build an adjacency list where linked list corresponding to each vertex, say a_i stores the weight of all edges in an order of the vertices, say $\{a_1, a_2, a_3, ..., a_n\}$
Note: In both the adjacency lists we will have to maintain a pointer between two edges,say (a_i, a_j) and (a_j, a_i). This can be done in O(n^2) time while building unsorted adjacency list.
Note: After building this unsorted adjacency list, build the sorted adjacency list. In this way the pointer between edges (a_i, a_j) and (a_j, a_i) will be restored in the sorted adjacency list. Also maintain a pointer between the respective edges in unsorted and sorted adjacency lists. This tool will help me while removing edges.
Once we built this adjacency list with all its linked list sorted we can implement the algorithm written above in the form of a pseudo-code in O($n^2 logn$) time.
Now consider all those operations mentioned in the pseudo-code written above one by one
First operation: To find the edge with minimum weight in G. This can be achieved in O(n) time since it requires a single scan of the first column of the adjacency list.
Second operation: To remove vertices a_1 and a_2 where (a_1, a_2) is the least weight edge and insert vertex $a^{'}$ where $a^{'}$ is defined above. This can be achieved in O(n) time.
Note: Removing vertices a_1 and a_2 in the adjacency list will also remove their corresponding linked list so there is no need to remove those edges as they have already been removed.
Third operation: To create the sorted linked list for $a^{'}$: In the unsorted adjacency list remove a_1 and a_2 and insert $a^{'}$ with its linked list containing weights which is smaller of the two edges from a_1 and a_2. This can be done in O(n) time. Then sort this linked list and add this in sorted linked list to vertex $a^{'}$. This can be achieved in O($nlogn$) time. We also have to remove the edges (a_j, a_i) where j $\in \{1, 2, ..., n\}$ and i $\in \{1, 2\}$. This can be removed in unsorted adjacency list in O(n) time since we have a pointer between edges (a_i, a_j) and (a_j, a_i) in both the adjacency lists and since we retain a pointer between the respective edges in sorted adjacency list and unsorted adjacency list we will also delete it from the sorted adjacency list.
Fourth operation: To recurse this function on a smaller instance of G named $G^{'}$ with n-1 vertices. This will be achieved in T(n-1) where T(n) is time taken to achieve Greedy_Tree(G).Thus total time complexity can recursively be related as:
T(n) = $O(nlogn)$ + T(n-1)

Thus T(n) = $O(nlogn) + O((n-1)log(n-1) + ... + O(1)) \leq O(n(nlogn)) = O(n^2logn)$
Thus this whole algorithm can be implemented in $O(n^2logn)$

2 A novel algorithm

Algorithm 2: An inspirational algorithm

$E_S = \phi$;
while *G is not empty* **do**
  Pick any vertex v from G;
  if $degree(v) \leq \sqrt[2]{n}$ **then**
    Add all edges incident on v to E_S ;
    Remove v and all edges incident on v from G ;
  else
    Remove any edge from G between any two vertices of set P(set of neighbours of v);
    Remove edges from G such that any neighbour of vertices in P(except v) has only one neighbour in P;
    Add all edges incident on v or vertices in P to E_S ;
    Remove v, vertices in P and all edges incident on v or vertices in P from G ;
  end
end
return E_S;

2.1 Explanation

According to the algorithm we choose any vertex, say v. We look at its degree at that time in the remaining graph G.

2.1.1 if part: $degree(v) \leq \sqrt[2]{n}$

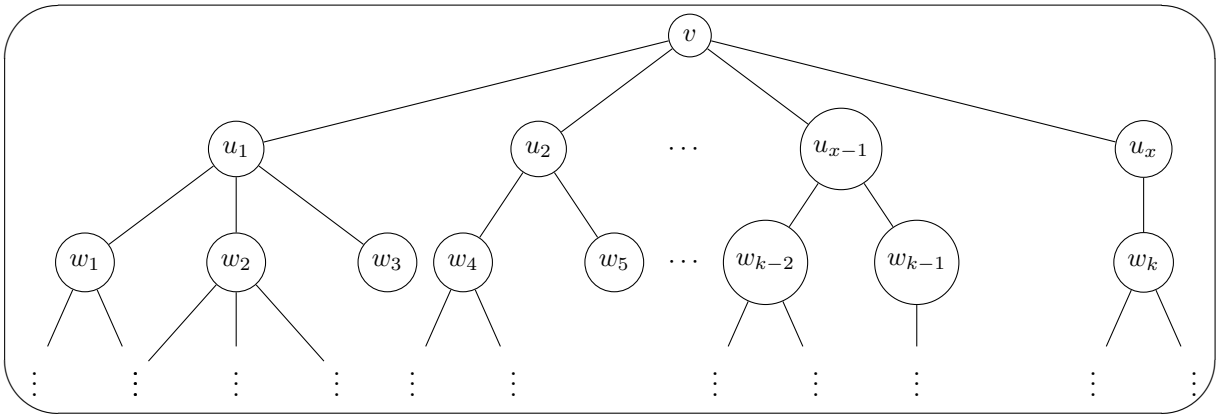
We simply look at the neighbour of v. We add all the edges incident on v to E_S. We remove vertex v and all edges incident on it from G.

Lemma 4. *During processing of v, we add atmost $\sqrt[2]{n}$ edges to E_S.*

Proof. Since $degree(v) \leq \sqrt[2]{n}$ and we add all the edges incident on v to E_S, the lemma holds. □

2.1.2 else part: $degree(v) > \sqrt[2]{n}$

Look at the following picture where
$x = degree(v)$
u_i : neighbour of v $\forall 1 \leq i \leq x$
w_j : neighbour of u_i except v $\forall 1 \leq j \leq k$



There may be edges between any u_i and u_j. We are deleting any such edge in this part. Also consider any w_z. There may be multiple neighbours of w_z in the set of u_is. We are also deleting any edge of kind (u_i, w_z) such that w_j has exactly one neighbour in the set of u_js. There may be edges between any w_y and w_z. We are not deleting any such edge. We are deleting any edge only in graph G. After all these removals we add the edges incident on any vertex u_i or v to set E_S and remove these edges from G. Also we remove all the vertices u_i and v from G.

Lemma 5. *For each pair of vertices u_i and u_j(these vertices are as shown in figure), there exists a path of at most 3 edges in H.*

Proof. We have added all the edges incident on v to E_S. Therefore all the u_zs are still connected with v via one edge. We go from u_i to v and from v to u_j. Hence we get a path of 2 edges between u_i and u_j. □

Lemma 6. *For each pair of vertices u_i and w_j(these vertices are as shown in figure), there exists a path of at most 3 edges in H.*

Proof. While removing edges of kind (u_y, w_z) from G we made sure that w_z has exactly one neighbour in the set of u_ys. Then we added all the remaining edges incident on any u_y to H. Therefore any vertex w_z has exactly one neighbour in the set of u_ys in H. For the lemma, suppose w_j has u_l as its neighbour in H. From Lemma 1, there exists a path of exactly 2 edges between u_l and u_i in H. Adding edge (u_l, w_j) to this path we get a path of 3 edges from u_i to w_j in H. □

Lemma 7. *While processing v, if edge (p, q) was removed from G then there exist a path of atmost 3 edges from p to q in H.*

Proof. We deleted only two kind of edges while processing v, (u_i, u_j) and (u_i, w_j). We showed in Lemma 1 and Lemma 2 that we have a path of atmost 3 edges in H for any such removed edge in G. Therefore the lemma holds. □

Lemma 8. *While processing v, atmost $\sqrt[2]{n}$ edges(on an average) per processed vertex(during processing of v) was added to E_S.*

Proof. We added all edges of type (v, u_i) $\forall 1 \leq i \leq x$ to $E_S \Rightarrow x$ edges.
We added all edges of type (u_i, w_j) after making sure that each w_j has exactly one neighbour in the set of u_is. Suppose there were k vertices in set w_js, we added only $k < n - x$ edges(one per vertex w_j) to E_S. Therefore while processing v we added less than n edges in total. Since we processed $x + 1$ vertices(v and all u_is) and $x > \sqrt[2]{n}$, we added atmost $\sqrt[2]{n}$ edges per processed vertex. □

2.2 Proof of Correctness

Since we are removing atleast one vertex and all of its edges in each iteration of the algorithm, the algorithm is guaranteed to terminate in at most n iterations. To prove that the distance between any pair of vertex(u,v) in H is at most 3 times the distance in G, it suffices to prove that for each(u,v) in G but not in H, there exists a path of atmost 3 edges connecting u and v in H(because then for each edge in the original path, we have a path of at most 3 edge summing this path we will get a path of at most 3 times the distance in G).

Theorem. *For each edge $(u, v) \in E$, $(u, v) \notin E_S$, there exists a path of atmost 3 edges connecting u and v in H.*

Proof. If an edge $(u, v) \in E$, $(u, v) \notin E_S \Rightarrow$ edge(u, v) would have been processed while processing a vertex with degree greater than $\sqrt[2]{n}$. Therefore the Lemma holds from Lemma 4. □

Theorem. *Size of subgraph H is $O(n^{\frac{3}{2}})$.*

Proof. From Lemma 1 and Lemma 5, we add atmost $\sqrt[2]{n}$ edges to E_S per vertex on an average. Therefore, for n vertices we add atmost $n^{\frac{3}{2}}$ edges. Hence, the lemma holds. □

2.3 Fastest Implementation

Our input will be in the form of adjacency list. We will store one additional field at each node in the adjacency list. Consider an edge(u, v) in G. In adjacency list, array index u will point to a list containing node v and array index v will point to a list containing node u. We will keep a pointer also in node v in array index u which will point to node u in array index v. Similarly we will do this at each node. This will felicitate deletion of an edge in $O(1)$ time. Each insertion can be done in $O(1)$ time trivially. Also we will keep an array $Distance[\,]$ of size n. This array will be initialized to ∞.

2.3.1 if part: $degree(v) \leq \sqrt[2]{n}$

We simply add all the edges incident on v to E_S, $O(1)$ time per edge. We then remove all edges of v from graph G, $O(1)$ time per edge. Total no. of edges = $degree(v)$

$$Time\ complexity = degree(v)$$

Algorithm 3: Process-like-BFS(G,v)

```
CreateEmptyQueue(Q);
Distance(x) ← 0;
Enqueue(x,Q);
while Not IsEmptyQueue(Q) do
    v ← Dequeue(Q);
      for each neighbour w of v do
        if Distance(w) =∞ then
          Distance(w) ← Distance(v) + 1
          if Distance(w) = 1 then
            Enqueue(w)
          end if
          Add edge(w,v) to E_S
          Remove edge(w,v) from G
        else
          Remove edge(w,v) from G
        end if
      end for
    Remove v from G;
end
```

2.3.2 else part: $degree(v) > \sqrt[2]{n}$

We have seen that we need to remove all the edges of type (u_i, u_j) and (u_i, w_j). Considering the set of vertices u_is to be first level, set of vertices w_js to be second level and v to be root, a BFS from root upto two levels only will serve our purpose. During this BFS we have to add only tree edges to E_S and remove all the non-tree edges from G.

We see that each vertex processed takes time of the order of its degree at that time. Hence total time taken for one call to Process-like-BFS() takes time of the order of sum of degree of all the vertices processed in the call. Additionally we will again have to initialise array Distance[ ] to ∞. We can keep an array of vertices visited in the call to Process-like-BFS() and change entry corresponding to these vertices only. Number of these vertices will be less than the sum of degree of all these vertices. Therefore a call to Process-like-BFS() and maintaining array Distance[ ] takes time of the order of sum of degree of all the vertices processed in the call. Therefore,

$$Time\ complexity\ for\ processing\ vertex\ v = O(degree(v))$$

From 2.3.1 and 2.3.2

$$Time\ complexity\ for\ algorithm = \sum_{v \in G} O(degree(v)) = O(m + n)$$