Algorithm Analysis and Design
Problemset 2

- ARYAMAN MANISH KOLHE
2022121002

Q1. Let G have n vertices and m edges, whose edge weights are all distinct. Give an algorithm to decide whether a given edge e is contained in a minimum spanning tree of G.

Let C_e = cost of edge e
Let $e = (u, v)$ such that $u, v \in V(G)$

Note how, since all edge weights are distinct, $\exists$ a unique MST in G.

Approach 1 - $O(m \log m)$

1 - sort all edges in ascending order of edge weights
2 - Build an MST, T, using Kruskal's algorithm, till the edge e is reached
3 - If adding e to T creates a cycle, then e is not in MST(G).
Else, $e \in MST(G)$

Proof of correctness -
The proof of correctness follows from the fact that Kruskal's algorithm is optimal (with both cycle and cut property).

Approach 2 - $O(m)$ $(\because m > n)$

We can come up with a more efficient solution using only the cycle property.

Cycle Property -

For any cycle C in G_1, if $e_m = (i, j)$ is the most expensive edge in C, then e_m does not belong to any minimum spanning tree of G_1.

Rough Idea - $G_1 = (V, E)$, $e = (u, v)$

1. Construct a graph $G_2 = (V, E - \{e\})$.
2. Use BFS starting from node u.
3. While doing BFS, consider edge e_i only if $C_{e_i} \leq C_e$.
4. If node v is present in the BFS tree of G_2, then we've indirectly found a cycle including u, v such that e is the maximum weight edge (since we only considered $e_i : C_{e_i} \leq C_e$). In this case, $e \notin MST(G_1)$
5. If node v is not present, then $e \in MST(G_1)$.

Algorithm -

```
discovered = list of length n. all values set to False
discovered[u] = true
Q = ϕ    // queue for BFS
Q.enqueue(u)
```

```
while Q is not empty :
    k = Q.dequeue()

    For each e_i = (l, k) ∈ E(G) :
        if c_ei ≤ c_e :   // c_ei = cost of edge e_i
            if discovered[l] == False :
                discovered[l] = True
                Q.enqueue(l)
            if l == v :
                return False
                // e is not in MST(G)

// v not encountered. e ∉ MST(G)
return True
```

Proof of Correctness: (inspired from Kruskal's Algo)

Let the edges in sorted (ascending) order be $e_1, e_2, \ldots, e_n$. Edge $e = e_k$ (to be checked)

$e_k = (u, v)$

If you start from u, and traverse along edges with weight $c_{e_i} \leq c_{e_k}$ then you can reach the vertices in e_i ($e_i \leq e_i \neq e_k$). Let these vertices be p_i. This means that until e_k is considered, we have built a subgraph H (of G), which has all p_i and all e_i ($e_i \leq e_i \neq e_k$). It does not matter whether some e_i form a cycle among themselves. We just want to see if v can be reached using only e_i's.

∴ if v can be reached, then $v \in \{p_i\}$, which means that considering e_k will be redundant (since we've already reached v in H). If $v \notin \{p_i\}$, then we can consider e_k to reach v from u.

Q.E.D

Q2. $G_1 = (V, E)$, simple, undirected,
$c_e \geq 0 \quad \forall \; e \in E$
T is an MST of G_1.
A new edge $e_k = (v, w)$ with $c_{e_k} = c$ got added to G_1. (for $v, w \in V$.)
Give an efficient algorithm to test if T remains MST(G_1). If not, then give an efficient algo to update T to make it an MST. *

Observation(s):
T is an MST(G_1). If you add e_k to T, then it will no longer remain a tree (Since $|E(T)| \neq |V(G_1)| - 1$). Let this graph be T'. (add e_k to T to get T')

By properties of trees,
T' has a cycle, C.
All we have to do is, find the max weighted edge in C and remove it to turn T' into an MST of G_1. [By Cycle Property of MST].

∴ We can use the same algorithm to solve both the above problems efficiently. (*)

Algorithm : O(m+n)

discovered : list of length n, set to False
discovered[v] = True
parent = list of length n, set to NULL
Q = ∅ // for BFS

While Q is not empty :
    u = Q.dequeue()
    flag = False
    For each edge e_i = (u_2, u) ∈ E(G) :
        if discovered[u_2] == False :
            discovered[u_2] = True
            Q.enqueue[u_2]
            parent[u_2] = u
        else if discovered[u_2] = True and parent[u] != u_2 :
            // cycle found
            flag = True
            break

    if flag == True
        break
e_max = e_k // assume that max wt. edge is the new one
C_max = C_ek
current = W
While current != V :
    e_i = (current, parent[current])
    C_ei = cost(e_i)
    if C_ei > C_max :
        C_max = C_ei
        e_max = e_i
    current = parent[current]

if $e_{max} == e_k$:

print "T is still an MST of G"

else :

$T\_new = (T - \{e_{max}\}) \cup \{e_k\}$

print "T is no longer an MST of G"

print "Updated MST is :", T_new

Proof of Correctness :

T was an MST of G.

∴ No. of edges in T = $|E(T)| = |V(G)| - 1$

After adding $e_k = (v, w)$, to T, a cycle got formed, and to make T the MST, we will have to remove the maximum weighted edge in the cycle (By the cycle property)

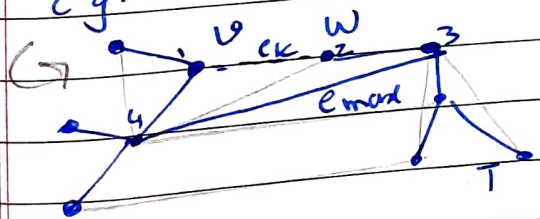
We essentially replace e_{max} with e_k, and hence preserve the no. of edges.

$|E(T\_new)| = (E(T) - 1) + 1$ // [This works because ∃ a unique path between any 2 nodes of a tree.]

This algorithm is $O(m+n)$ (BFS), and is efficient, because we'll have to traverse T, or do cycle detection to solve the problem. ∴ Algo is $\Theta(m+n)$.

Q.E.D

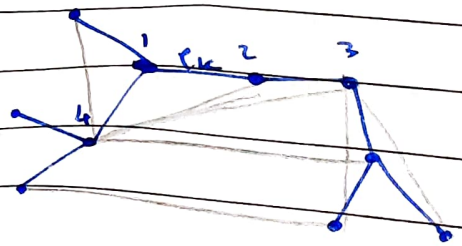
E.g.



cycle = 1, 2, 3, 4 after adding e_k

$(3, 4) = e_{max}$

After algo -



= MST = G

Q3. Given a list of n natural numbers, $d_1, d_2, \ldots, d_n$, show how to decide in polynomial time whether there exists an undirected simple graph $G = (V, E)$, whose node degrees are precisely the numbers $d_1, d_2, \ldots, d_n$.

Observation :

If the list was stored in array, $D = [d_1, d_2, \ldots, d_n]$, then we can keep reducing $D[i]$, $D[j]$ by one each (add edge between v_i, v_j).

The graph exists if $D[i] = 0$ for all i at the end.

Approach 1 - DP based approach

There must exist a sequence of pairs with which we can get $D[i] = 0$ $\forall i$.

Use 2D - DP to keep track of which edge is added in the adjacency matrix representation of this graph.

However, this solution is memory inefficient.

Approach 2 -

Use a greedy algorithm.

If we sorted D in descending order, we can add an edge between $D[i]$, $D[i+1]$. This greedy choice is safe since choosing $D[i]$, $D[j]$ $(j \neq i+1)$ will not help exhaust all edges (all elements in D should be 0 at the end).

Algorithm -

```
//quick sort test for descending order
function quicksort-desc(A, lower, higher):
    if lower < higher
        p = partion(A, lower, higher)
        quicksort-desc(A, lower, p-1)
        quicksort-desc(A, p+1, higher)

function partition(A, lower, higher)
    pivot = A[higher]
    i = lower - 1
    For j = lower to higher - 1
        if A[j] > pivot : // descending order
            i = i + 1
            swap(A[i], A[j])
    swap(A[i+1], A[higher])
    return i+1

V = [0, 1, ..., n-1]
E = ∅          // Gr = (V, E), construct Gr on the way
D = [d0, d1, ..., dn-1]
quicksort-desc(D, 0, n-1)
possible = True // flag that assumes it is possible to make Gr

i = 0
p = dictionary that keeps track of the vertex index based on D.
    If you do not want to generate Gr, then this can be ommitted.
```

While i < n and D[i] != 0 :

```
    For   j = 1, 2, ..., D[0] :
        if i+j >= n or D[i+j] == 0
            possible = False
            break
        D[i+j] = D[i+j] - 1
        add (P[i], P[i+j]) to E.

    if possible == False :
        break

    D[i] = 0
    i = i + 1
    quicksort-desc ( A, i, n-1 )
    // update P according to A
```

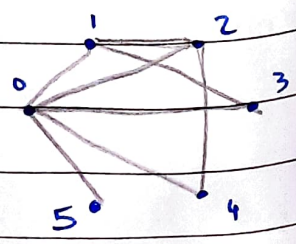
if i == n : break

```
if possible == True :
    print "It is possible, and we have constructed G:", G
else :
    print "It is not possible to construct G."
```

Example -

		0	1	2	3	4	5
P							
D		5	3	3	2	2	1
⇒		0	2	2	1	1	0
⇒		0	0	1	0	1	0
P	sort	0	1	2	4	3	5
D		0	0	1	1	0	0
⇒		0	0	0	0	0	0

G :



Time Complexity Analysis:

We've chosen quick sort so that when it is run in the while loop, it is sorting a "nearly sorted array".

In the worst case scenario,

$D = [k, k, \ldots, k]_{1 \times n}$

The first for loop will iterate k times.
The next time it will iterate $k-1$ times and so on, so we get the series:

$$k + (k-1) + (k-2) + \cdots + 1$$

↳ $D[n-1]$.

$= \frac{k(k+1)}{2}$ elementary steps and k can't be more than n (graph)

∴ The algorithm is $O(n^2)$, which is polynomial.

Proof of correctness:

WLOG, let $D = [d_0, d_1, \ldots, d_{n-1}]$, ∋ $d_0 \geq d_1 \geq d_2 \geq \ldots \geq d_{n-1}$

In our algorithm we took an edge b/w $(0,1)$ then $(0,2)$, and so on till d_0 became 0. (last one being d_k), $k = d_0$.

If we took one from some d_i, $i > k$ instead of d_k, then one node j from $0 \leq j \leq k$ will not get exhausted, and will return an incorrect result.

∴ The greedy solution is optimal. (proved by contradiction, that no better choice exists).

Q.E.D.

Q4. $G = (V, E)$, $n = |V|$, each pair of nodes is joined by an edge. $\therefore$ G is fully connected.

Weight $(i, j) = w_{ij}$ such that
$w_{i,k} \leq w_{i,j} + w_{j,k}$ (triangle inequality)

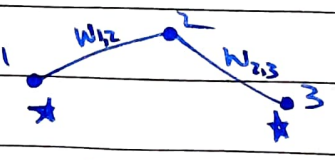
For $V' \subseteq V$, $G[V']$ = subgraph with nodes V' and valid edges from E.

Set of k terminals $= X \subseteq V$
T is a Steiner tree on X, such that
$T = (Z, E_T) \Rightarrow X \subseteq Z \subseteq V$.
Show that the problem of finding a minimum weight Steiner tree on X can be solved in $n^{O(k)}$ time.

OBSERVATIONS -

What is a Steiner Tree -
- Generalization of the MST problem.
- for $k = 2$, it reduces to shortest path problem
- for $k = n$, it reduces to MST on G.

Why $X \subseteq Z$ and not $X = Z$
for $X = \{1, 3\}$



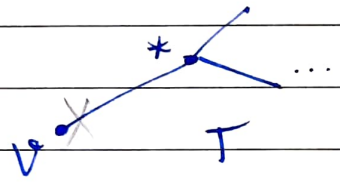
T ~~contains~~ has $Z = \{1, 2, 3\}$

Finding T is hard since we don't know $Z - X$ beforehand.

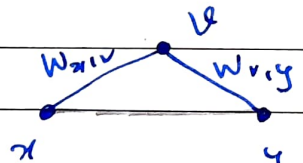
Nature of $Z - X$;

Claim 1: Each node in $|Z - X|$ has degree at least equal to 3.

Proof :
If degree of $v \in Z - X$ is 1 then; v is a leaf in T, which makes it redundant (increases the weight of T).



If degree of $v \in Z - X$ is equal to 2, then we get the following ~~graph~~ subgraph in T :



$x, y \in X$
$v \in Z - X$

But $w_{x,y}$ (edge (x, y)) exists in G_T, since G_T is a complete graph.
And by the triangle inequality,
$w_{x,y} \leq w_{x,v} + w_{v,y}$.
so by removing v, we can reduce the weight of T.

$\therefore$ Degree of all $v \in Z - X$ is greater than or equal to 3.

From claim 1, we can obtain an upper bound on the set $Z - X$.

Claim 2 : $|Z - X| \leq k$ Proof :

Assume that the steiner tree (X) has p nodes, and q edges

By Euler's Handshaking lemma,

$$\sum_{v \in T} d(v) = 2q \qquad d(v) = \text{degree of vertex } v.$$

Also, T is a tree, $q = p - 1$.

$$\therefore \sum_{v \in T} d(v) = 2(p-1) = 2p - 2$$

Claim 3 :

Another property of trees is that the number of nodes with degree ≥ 3 is lesser than or equal to the number of leaves.

Proof : PROOF BY CONSTRUCTION

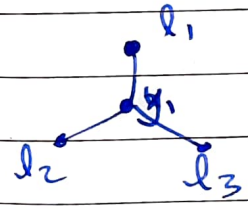
If no. of nodes with degree = 3 is 1. (let it be y_i)

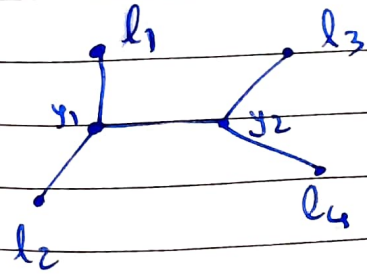
↳ Let this set be Y.

• Set of leaves = L.

$|L| \geq |Y|$ holds.

$(3 \geq 1)$





$4 \geq 2$

$\therefore |L| \geq |Y|$ holds.

If you replace any leaf with a node y_i, to make sure that $d(y_i) \geq 3$, we have to introduce at least one more leaf, which ~~support~~ build on top of the previous two examples.

(Alternatively, we can prove by contradiction by saying that if $|L| < |Y|$, then we get a cycle.)

Adding nodes of degree = 2 does not affect the length of L and Y, therefore, all trees can be constructed with this construction.

Q.E.D.

Back to Claim 2:

We have $\sum_{v \in T} d(v) = 2p - 2$, and claim 3, and claim 1.

$|Z - X|$ has only nodes v s.t. $d(v) \geq 3$.

∴ ~~leaves and~~ ~~terminals~~ ~~cannot~~ ~~exceed~~ Also, nodes of degree 1 are leaves (and therefore ~~are~~ marked), and nodes of degree 2 are also terminals (degree < 3).

terminals

∴ $|Z - L| \leq k$

∴ $|Z - X|$ is at most k.

Q.E.D (claim 2)

We have now reduced the problem space to all sets Z which contain X as a subset, and $|Z-X|$ is at most k.

$\therefore \quad |Z - X| \leq k$

$\Rightarrow \quad ||Z| - |X|| \leq |Z - X| \leq k$

$\Rightarrow \quad ||Z| - |X|| \leq k$

$\Rightarrow \quad ||Z| - k| \leq k$

$\Rightarrow \quad |Z| \leq 2k$

$\therefore$ Among n vertices, we need to find an MST in ${}^{n}C_{2k}$ subsets of V.

Case 1 : $2k > n$

We'll have to test for all possible trees, and find the Steiner tree with X

Case 2 : $2k \leq n$

$${}^{n}C_{2k} = \frac{n!}{(n-2k)!(2k)!} \leq \frac{\overbrace{n \cdot n \cdot n \cdots n}^{(k \text{ times})}}{(n-2k)!\,(2k)!}$$

$$\leq n^{O(k)}$$

Also, for each $n^{O(k)}$ testcase, we need to spend $2k \cdot \log(2k)$ time to find an MST.

$2k \log(2k) \leq n \log n \leq n\sqrt{n} = n^{3/2} \quad (2k \leq n)$

and $n^{3/2} \Rightarrow O(n^{3/2})$

$\therefore$ The algorithm is $O(n^{O(k)} \cdot n^{3/2}) = O(n^{O(k)})$

Q.E.D

References :

Q4. Discussed the possibilities of a tighter bound, and how $2k\log(2k)$ affects $O(n^{6k})$ with Yatharth Gupta (2021101093)