CS345 Theoretical Assignment 1

Ayush Agarwal, 13180

M.Arunothia, 13378

Contents

1 Neister Tree

1.1 Overview

Given a complete graph G(E,V), we construct neister tree of set X by using nodes in $\{G\backslash X\}$ in a brute force manner. Neister tree on X is a set Z so that $X \subset Z \subset V$, together with a spanning tree T of G[Z] such that weight of MST(G[Z]) is minimum.

1.2 Maximum value of Z

Lemma 1: Any node n $\in$ $\{Z\backslash X\}$ will have **degree** >= 3 in MST(G[Z]).
Proof:

- **degree**(n) can't be 0 since it $\in$ MST(G[Z]) which is fully connected.
- **degree**(n) can't be 1 because a Spanning Tree can be constructed by just dropping n, whose weight will be less than MST(G[Z]). In that case Z won't be neister tree.
- Suppose node n $\in$ Z
  X is connected to only 2 nodes (say x and y) in M=MST(G[Z]). Using Triangle Inequality we can say that,
  $\boldsymbol{\omega}(n,x)+\boldsymbol{\omega}(n,y) >= \boldsymbol{\omega}(x,y)$
  Now, we can have a Spanning Tree S in which there is an edge between x and y, and n is absent.
  weight(S)= $\textbf{weight}(M) - \boldsymbol{\omega}(n,x) - \boldsymbol{\omega}(n,y) + \boldsymbol{\omega}(x,y) <= \textbf{weight}(M)$
  Contradiction.

Hence we can say that $\textbf{degree}(n) >= 3$ in MST(G[Z]).

Lets calculate the maximum value of Z. Suppose $\mid Z\backslash X \mid = p$.
In a tree, $\sum_V \textbf{degree}(v) = 2E$. In case of MST(G[Z]), minimum degree of p and k vertices are 3 and 1 respectively.

$$\implies 3p + k <= 2(p + k - 1) \qquad \because E = V - 1 \text{ in a tree}$$
$$\implies p <= k - 2$$

1.3 Pseudo-Code

- **n** $=\mid V \mid$
- **vertexSet(G, k)**: returns a set of k distinct vertices from G which is different from previously returned sets
- **min_weight**: holds the minimum weight of all the trees seen so far.
- **neisterTree**: Holds the set of vertices(Z) which can be potential Neister-Tree.

```
1:  procedure NeisterTree($G, X, V$)
2:      min_weight ← MST(G[X])
3:      neisterTree ← X
4:      $T \leftarrow \{G\backslash X\}$
5:      for $i$ in 0 to $k-2$ do
6:          for $j$ in 1 to ${}^{n-k}C_i$ do
7:              $Z \leftarrow X \cup$ vertexSet($T, i$)
8:              $temp \leftarrow$ MST($G[Z]$)
9:              if temp $\le$ min_weight then
10:                 min_weight ← temp
11:                 neisterTree ← Z
12:             end if
13:         end for
14:     end for
15:     return neisterTree
16: end procedure
```

1.4 Time Complexity

For every ith iteration, time req. is

$${}^{n-k}C_i * \textbf{MST(Z)} <= {}^{n-k}C_i * n^2$$

Summing up time for every iteration,

$${}^{n-k}C_0 * n^2 +^{n-k} C_1 * n^2 +^{n-k} C_2 * n^2 + ... +^{n-k} C_{k-1} * n^2 +^{n-k} C_k * n^2$$
$$= {}^{n-k+1}C_{k+1} * n^2$$

$\leq {}^{n}C_{k+1} * n^2$
$\leq n^{k+1} * n^2$
$\leq n^{\mathcal{O}(k)}$
So time Complexity is $\boldsymbol{\mathcal{O}(n^{\mathcal{O}(k)})}$

1.5 Justification

This algorithm uses brute force to cover all cases. Basically it iterates over every possible subset of $\{V \backslash X\}$ of cardinality less than k-1, calculating the weight of the MST thus formed and takes minimum of them. The tree with the minimum weight is our answer.

2 Unique-path graph

2.1 Overview

Given there exists a vertex u which has a path to every other vertex, we approach by first finding out vertex u. Now, we apply a single DFS from u and keep extra tracker of *Back_edge_to* to check for unique-path nature of the graph.

2.2 Algorithm

- Start
- Mark all vertices unvisited.
- Start DFS from every vertex that is unvisited and mark them visited whenever they get visited in the process. Also estimate start and finish times of every vertex.
- Let u = Vertex with maximum finish time.
- Start a DFS from vertex u. Exit with false if any cross edge, forward edge, or more than one back edge from any sub-tree is encountered as in these cases the graph cannot be unique-path graph.
- Exit with true.
- Stop.

2.3 Pseudo Code

Run a DFS and find the vertex v with maximum Finish Time. Now Run the following starting from v.

```
procedure DFS($v$)
    UniquePathGraph ← true
    Visited[v] ← true
    D[v] ← count++
    backEdge[v] ← Null
    for each edge $(v, w)$ do
        if Visited[v] = false then
            $DFS(w)$;
            t ← backEdge[w]
            if t<> Null && Finished[t]=false then
                backEdge[v] = t
            end if
        else if Finished[w] then
            UniquePathGraph ← false
            break;
        else if backEdge[v] = Null then
            backEdge[v] ← $w$
        else
            UniquePathGraph ← false
            break;
        end if
    end for
    Finished[w] ← true;
    F[v] ← count++
end procedure
```

2.4 Time Complexity :

- Locating the vertex u takes a single DFS for every unvisited vertex - O(m+n)
- From u we make a single DFS - O(m+n)
- Therefore, overall it is O(m+n) algorithm.

2.5 Proof of Correctness

3 A real life application of Directed Acyclic Graphs

3.1 Overview

This problem is approached by exploiting the topological ordering of DAGs. By the definition of root and exit, they will appear on the leftmost and rightmost ends of the ordering.

3.2 Algorithm

- Start.
- Sort the vertices to get their topological ordering. Let $path$ be an array, defined as - $path[i]$ stores the number of distinct paths from root to the i^{th} node in the topological ordering. Initialize all entries of this array with 0. and $path[0] = 1$
- Let $i = 1$, $node_1$ denotes the first vertex after root in order. Run the following step till $i <> n - 1$
- For all incoming vertices to this i^{th} node : edge $(node_j, node_i)$ exists
  - assign edge weight as $path[i]$
  - $path[i] += path[j]$
- Stop.

3.3 Pseudo Code

```
Assign_edge_weight(V,E)
{
        node[ ] = topological_sort(V,E)
        path[ ] = 0
        path[0] = 1
         $i = node[1]$ // $node[0]$ will be $root$
         while($i <> n - 1$)
                For all $(node[j], node[i]) \in E$
                        $w(node[j], node[i]) = path[i]$
                        $path[i] += path[j]$
        return
}
```

3.4 Time Complexity

- Topological sorting - single DFS (using Finish Time) = $O(m + n)$
- O(while_loop) = O(sum of in degrees in the graph) = $O(m + n)$
- Hence, overall time for a query is $O(m + n)$.

3.5 Proof of Correctness

3.5.1 What is to be proved? or Claim

After i iterations, $nodej \in \{1, 2, .., i\}$ will have $path[j]$ to be the number of distinct paths that start from the root and end in $node[j]$. Each of these paths will be assigned a unique path id between 0 and $path[j] - 1$.

3.5.2 Proof by Induction

Induction is carried out on the iterator i

3.5.3 Base Cases

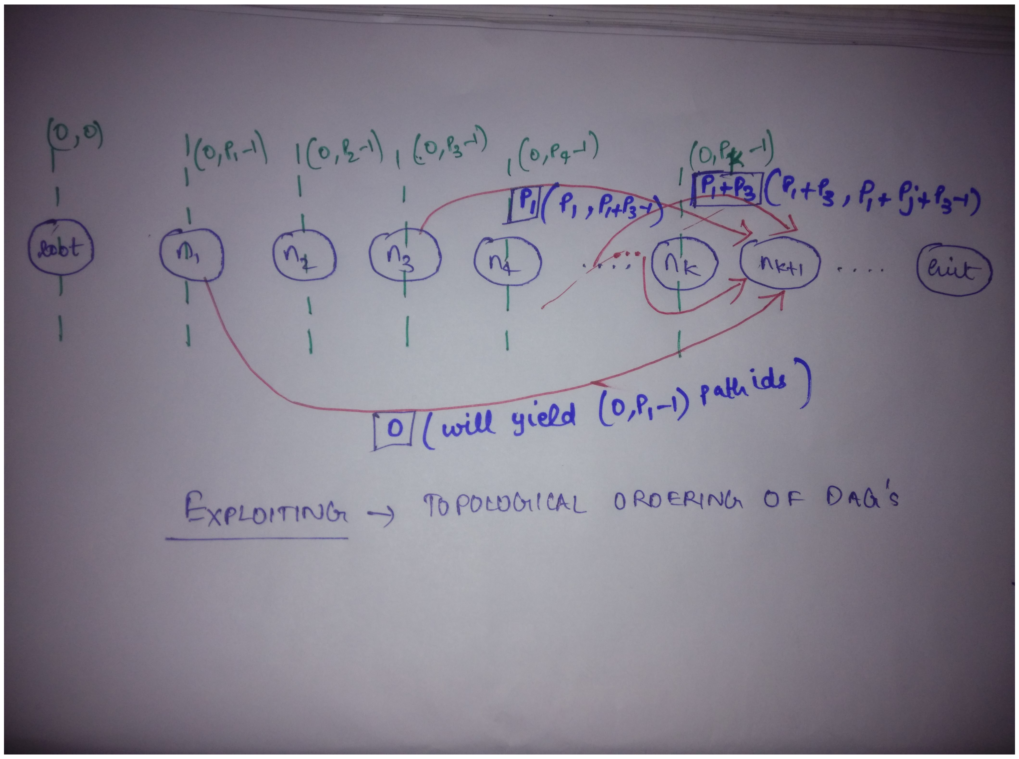
- $(i = 1)$ - Only one edge from root to this $node_1$.
- $w(root, node_1) = path[1] = 0$
- $path[1] = path[0] + 0 = 1$
- Hence, claim satisfies base case.
- Notice, if there was no edge from root also, the claim will still hold.

3.5.4 Hypothesis

Let us assume that the Claim is true for all $i <= k$ where $k > 1$ and both are integers.

3.5.5 Inductive Step

Let us prove that claim for $k + 1$ is true.


(0,0)
(0,P1-1)
(0,P2-1)
(0,P3-1)
(0,P4-1)
(0,Pk-1)
P1 (P1, P1+P3-1)
P1+P3 (P1+P3, P1+Pj+P3-1)
root
n1
n2
n3
n4
nk
nk+1
exit
0 (will yield (0,P1-1) path ids)
EXPLOITING → TOPOLOGICAL ORDERING OF DAG's

- if $(node_j, node_{k+1}) \in E$ then $j < k + 1$ as they are in topological order.
- From inductive hypothesis we know
  - $path[j]$ - The total number of distinct paths from root to $node_j$.
  - Every path from $root$ to $node_j$ have unique path-ids $\in [0, path[j] - 1]$.
- $w(node_j, node_{k+1}) = path[k+1]$ will help maintain distinct path-ids as the paths that were encountered to reach $node_{k+1}$ would have been assigned path-ids between 0 and $path[k+1] - 1$. This new set of paths to $node_{k+1}$ via $node_j$ will get path-ids in $(path[k+1], path[k+1] + path[j])$.
- Notice that every such j will add $path[j]$ number of distinct paths to reach $node_{k+1}$ from the root. Hence, $path[k+1] + = path[j]$ updates that in the loop.
- Hence, the claim holds for $k + 1$.
- Thus, proved by induction.