CS345 Theoretical Assignment 3

Abhibhav Garg 150010
Abhisek Panda 150026

September 23, 2017

1 Question 1

Given a DAG $G = (V, E)$, task is to assign integral weights to the edges so that every path from $s(source)$ to $t(exit\ vertex)$ has a unique pathID. pathID is defined as the sum of weights of all edges in the path.

1.1 Algorithm:

We follow a bottom-up procedure to find the required edge weights. The steps are discussed below:

- The given graph is first topologically sorted(since its a DAG, a topological sorting exists).s appears as the first vertex and t as the last in the sorted graph. All other vertices are ignored.
- Consider each vertex(say u) in order of their appearance in topological sort, starting from s. Let X_i denote the vertices such that $(X_i, u) \in E$. Let the number of paths from s to X_i be denoted by n_i. Since a bottom up approach is followed, all n_i's are already known.
- The edges (X_i, u) is assigned weights as follows:

Algorithm 1 Assigning weights to edges ending at u:

```
function AssignWeights(G, u)
    U ← {x | (x, u) ∈ E}
    incr ← 0
    for x_i in U do
        n_i ← Number of paths from s to x_i
        w(x_i, u) = incr
        incr+ = n_i
    end for
end function
```

- So the entire algorithm becomes:

Algorithm 2 Assigning weights to all edges in G

```
function FinalWeights(G, u)
    G' ← TopologicalSort(G)
    for u in order of appearance in G' do
        AssignWeights(G, u)
    end for
end function
```

1.2 Proof of Correctness:

At any iteration step through the topologically sorted graph, let the vertex being considered be u. Let $U = \{x \mid (x, u) \in E\}$. We will prove the correctness by induction on weights of edges considered till u.

Induction Hypothesis: Applying the above algorithm for any vertex u yields the correct weights so that every path has a unique ID in the range of 0 to $n-1$, where n is total number of paths from s to u.

Base Case: When $u = s$, U is either s or an empty set. So if s has a self loop type edge, it is assigned a weight 0, otherwise the function returns. Hence, the induction holds for base case.

General case:
Let n_i correspond to number of paths from s to x_i. By induction hypothesis, all edges considered till now are assigned weights such that each path from s to x_i has a unique pathId between 0 to $n_i - 1$.

- Consider edge (x_1, u). Assign it a weight 0.
- Consider edge (x_2, u). The paths from s to x_2 have unique IDs between 0 to $n_2 - 1$ by induction hypothesis. Since we already have paths from s to u with Ids from 0 to $n_1 - 1$, assign this edge a weight n_1 so that we get paths with Ids between n_1 and $n_1 + n_2 - 1$
- Assign a weight of $n_1 + n_2$ to (x_3, u) to get pathIDs in the range $n_1 + n_2$ to $n_1 + n_2 + n_3 - 1$.
- Similarly, for any vertex $x_i (i > 1)$, assign the corresponding edge a weight of $\sum_{j=1}^{j=i-1} n_i$

Thus at the end of the iteration, we get weights so that pathIds from s to u are unique and lie between 0 to n - 1. Thus by induction, correctness follows.

1.3 Time Complexity Analysis:

Topological Sorting of the graph takes $\mathbf{O}(m + n)$ time.
Number of paths can from s to u can be written as $p(u) = \sum_{x:(x,u)\in E} p(x)$. By following the bottom up approach, this can be calculated in $\mathbf{O}(deg(u))$ time for each vertex. So this takes $\mathbf{O}(m + n)$ time.
The edge weights can be assigned in the same traversal as for calculating number of paths which again takes $\mathbf{O}(deg(u))$ time for each vertex. Thus, this will also take $\mathbf{O}(m+n)$ time which leads to an overall complexity of $\mathbf{O}(m + n)$.

2 Question 2:

To find if the graph is a unique path graph given a graph G with a vertex s such that each vertex is reachable from s in $O(m+n)$ time.

2.1 Algorithm:

Algorithm 3 UNIQUE PATH GRAPH

```
1: UniquePathGraph ← false
2: Visited[] ← false
3: Finished[] ← false
4: D[] ← 0;
5: crossingEdges[] ← 0
6: procedure AUGMENTED-DFS(u)
7:     Visited[u] ← true
8:     count++
9:     D[u] ← count
10:    topMostPoint ← D[u]
11:    for each edge (u, v) do
12:        if Visited[v] = false then
13:            highPoint = Augmented-DFS(v)          ▷ Parent gets the high point of its subtrees
14:            topMostPoint = min(topMostPoint, highPoint)
15:            if highPoint < D[u] then              ▷ Back edge is crossing the parent node
16:                crossingEdges[u] ++
17:                if crossingEdges[u] > 1 then
18:                    UniquePathGraph ← false
19:                    return
20:                end if
21:            end if
22:        else
23:            if Finished[v] then                   ▷ Forward or Cross Edge
24:                UniquePathGraph ← false
25:                return
26:            else
27:                crossingEdges[u]++                ▷ Since u has a backedge
28:                                                  ▷ increase the count for subtree rooted at u
29:                if crossingEdges[u] > 1 then
30:                    UniquePathGraph ← false
31:                    return
32:                end if
33:                topMostPoint = min(topMostPoint, D[v])
34:            end if
35:        end if
36:    end for
37:    Finished[u] ← true;
38:    return topMostPoint
39: end procedure
```

2.2 Proof of Correctness:

As discussed in class, in a DFS tree presence of any cross edge or front edge guarantees that the graph is not unique path. Similarly, 2 back edges from the same vertex also guarantee that graph is not unique path.

The proof that front edges, cross edges do NOT permit the graph to be unique path is as discussed in class. Here, we give the proof for the other constraint.

Lemma 1. *Consider any subtree of the DFS tree made by staring from the source vertex. If there are 2 or more backedges leaving this subtree, then the path is not unique path graph.*

Proof. Let there be 2 such backedges (x,y) and (u,v). Root of the subtree is w. Then there are 2 paths possible from w to y(Assuming v is above y), one is $P(w,x)+(x,y)$ and the other is $P(w,u)+(u,v)+P(v,y)$ □

Therefore, we check for every vertex u, the number of backedges from subtrees of u with end vertex (strictly)higher than u(i.e. $D[endpt] < D[u]$). DFS at any node returns the highest point(among backedges) of its subtree to its parent. The parent then increments its number of crossing edges(number of back edges with end points higher than itself) if the end point is higher than it. The graph ceases to have unique path property if at any node, number of back edges going higher than it from its subtree is greater than 1. The parent compares the return values of all its descendants, if no edge crosses it, it returns its own DFS number.

2.3 Time Complexity:

Since only one DFS is performed, time complexity is $\mathbf{O}(m+n)$.