CS 345A Assignment 4

Yasharth Bajpai (170822), Nidhi Hegde (180472)

October 21, 2020

Difficult

Overview:

We are given a DAG, so we calculate a topological ordering on the graph. Starting from s in the topologically sorted ordering, we sweep through the ordering, assigning weights to the paths in the following manner:

- Let's say we are at a vertex v in the ordering. We define $In(v) = \{e \ : \ (e, v) \in E\}$
- At the current step, we assign weights to all the edges from vertices in the set $In(v)$ to the vertex v and we also compute $numPaths[v]$ which represents the number of unique paths between s and v. For that, we do the following:
- We initialize a variable $currCount$ to 0.
- Starting at any vertex x in the set $In(v)$,we check if the vertex lies in any path from s to t, if it does, we assign a weight $currCount$ to the edge (x, v) and increment $currCount$ by $numPaths[x]$. We repeat this step for all other vertices in the set.
- At the end of the process, the variable $currCount$ stores $\left(\sum_{\forall u \in In[v]/\{i_n\}} numPath[u]\right) - 1$. We then update $numPaths[v] = currCount + 1$ to store the number of unique paths from s to v ID'd from 0 to $numPaths[v] - 1$.

  On repeating the above procedure for every element in the topologically sorted array in order, we have the required graph, weighted for Unique PathIDs from s to t.

Pseudo Code:

Algorithm 1 Algorithm to Unique PathIDs to a DAG

```
function UniquePath(G(V, E))
    G_tp ⟵ TopologicalSort(G)
    numPaths[V] ⟵ 0
    for each v in order from G_tp do
        if (In(v) == 0) then
            continue
        end if
        currCount ⟵ 0
        for each u in In[v] do
            if (!(u ≠ s and numPaths[u] == 0))) then
                w(u, v) = currCount
                currCount = currCount + numPaths[u]
            end if
        end for
        numPaths[v] = currCount + 1
    end for
end function
```

Proof of Correctness:

We can be certain that there exists a topological ordering for the graph, it being a DAG. Once Topologically sorted, vertices in the graph are considered in the order of their occurrence. We'll prove the correctness of this algorithm using Induction as follows.

Let's consider the vertex v is picked at a certain point in the algorithm. $In[v]$ is defined as the set of vertices with an incoming edge to v.

Induction Hypothesis: At end of each step of the iteration, $numPaths[v]$ stores the number of unique paths from s to v and every such path is assigned a unique *pathid* from 0 to $numPaths[v] - 1$.

Base Case: Consider a two vertex graph from an edge from s to t. A weight of 0 is provided to the edge (s, t). This is the only unique path in the graph. $numEdges[t]$ is set to 1.

Induction Step: We assume the claim holds true for all the vertices before v in the array.
Since it is a topological order, any vertex with a directed edge to v should lie before v in the array, and hence we can say that the claim holds true for all the vertices in the set $In[v]$.
Now, we do the following steps:

1. Let's take a vertex i_1 from this set. As the claim is true for i_1, we know that all paths from s to i_1 are uniquely ID'd from 0 to $[i_1] - 1$.We assign a weight 0 to the edge (i_1, v), thus all the paths from s to v via i_1 will also be assigned a *pathid* from 0 to $numPaths[i_1] - 1$.

2. Next, consider the next vertex i_2 in the set $In[v]$, the paths from s to i_2 being uniquely ID'd from 0 to $numPaths[i_2] - 1$ If we add a weight $numPaths[i_1]$ to the edge (i_2, v), then all the paths from s to v via i_2 will be assigned a $PathID$ from $numPaths[i_1]$ to $numPaths[i_1] + numPaths[i_2] - 1$.

We proceed in this way until all vertices in $In[v]$ have been handled. Let's say there are n vertices in the set.
Step 1 assigns $pathID$'s from 0 to $numPaths[i_1] - 1$.
Step 2 assigns $pathID$'s from $numPaths[i_1]$ to $numPaths[i_1] + numPaths[i_2] - 1$, and so on.
Step N assigns weights from $\sum_{\forall u \in In[v]/\{i_n\}} numPaths[u]$ to $\left(\sum_{\forall u \in In[v]/\{i_n\}} numPaths[u]\right) + numPath[i_n] - 1$.
On reaching the end, we can ensure the following things:

- **All the paths from s to v have been considered**. This is because we consider all the vertices in $In(v)$, and any path from s to v should be via one of the vertices in this set.

- **The allocation of pathId is continuous and non-overlapping**. This fact is clear by the procedure of the algorithm described through Step 1 through Step N. After any step, the list of allocated ID's becomes 0 to $currCount - 1$. At the current step, the allocation starts at $currCount$. Since there is no overlap, there will be no two paths between s and v with the same $pathID$.

- **PathID's are assigned in the range from 0 to $numPaths[v] - 1$**. Clearly, the allocation starts at the $PathID$ value 0. If we prove that Step n ends at the $PathID$ value $numPaths[v] - 1$. we are done, as we know that the allocation is continuous. Since the number of paths from s to v can also be

expressed as the sum of number of paths from s to v via any of the vertices in set $In(v)$, we can say that $numPaths[v] = \sum_{\forall u \in In[v]/\{i_n\}} numPaths[u]$, which is exactly where we end in step N.

Hence, we prove that the algorithm works for vertex v if it works for all the preceeding vertices in the array. So, our Induction Hypothesis is proved.

Note: Even if there are multiple source vertices, we need a minor change in our algorithm. Instead of considering all the vertices in the set $In(v)$, we only consider those vertices with a path originating from s. So, if there is a vertex in the set $In(v)$ with no possible path from s, we simply ignore it, as done in our pseudo-code.
This step does not incur any extra time since we already maintain the value $numPaths$ for every vertex. So, if a vertex is not s and has $numPaths = 0$, then we simply ignore it.
We also skip any vertices before s in the topological ordering, during the outer loop of the procedure.

Time Complexity:

The algorithm utilizes Topological Sorting as a pre-processing before going on to assign the weights. Topological Sorting costs us $O(V + E)$.
Now, the outer loop works once for each vertex in the graph. The inner loop works for all incoming edges to a vertex. The number of times the statements inside the inner block are executed is bounded by E which is the total number of edges in the graph. The statements inside the inner loop are constant time($O(1)$) operations. So the complexity of the nested loops comes out to be $O(V) + O(E)O(1) \implies O(V + E)$.

Therefore, the complexity of the entire procedure is $O(V + E)$ i.e. Linear in the size of the graph.