2019

# Assignment 4

DESIGN ANALYSIS OF ALGORITHM

ANIMESH PATNI A20403240

## Question 1:→

In this problem, I am going to use the DFS call by just adding one part to it which haults the algorithm as soon as there is a cycle present. To achieve this I am going to add a stack that would contain the detail of the edges that are visited in the DFS call. We can even achieve the same using the discovery time & hault the code as soon as we find a cycle. The pseudo-code for this problem is as follows:→

```
DFS(G):
1. FOR EACH  u in G.V:
2.           u.COLOR = WHITE.
3.           u.π = NIL.
4. TIME = 0;
5. FOR EACH  u in G.V:
```

```
6.    IF U.COLOR == WHITE:
7.        DFS-VISIT(G, U, -1):

DFS-VISIT(G, U, p)
1.  TIME = TIME + 1
2.  U.d = TIME

3.  U.COLOR = GRAY
4.  FOREACH V ∈ G.Adj(U):
5.      IF V.COLOR == WHITE
6.          V.π = U
7.          DFS_VISIT(G, V, V.π)
8.      ELSE IF p != V and V.COLOR = GRAY.
9.          PRINT("CYCLE EXISTS")
10.         BREAK.
11.     END-IF
12. U.COLOR = BLACK.
13. TIME = TIME + 1
14. U.f = TIME
```

Running time Analysis:→

So, In the above pseudo-code, I

have taken DFS algorithm & modified it to find the cycle. As the problem says to write an algorithm that takes O(V) times, independent of the edges. So, to achieve this, I am terminating the code at line 10 as soon as we find the cycle. So, basically, we go through all the vertices & stop, which makes us independent of the edges. Thus, if there is a cyclic graph, we are terminating the code & it would take O(V) independent of E. If suppose we have an acyclic, the DFS algo will work as normal taking O(V+E).

## Describing the above algo :→

We would go through the graph similarly as the in the case of DFS traversal. The only change is that we are passing parent p for the vertex inside DFS-VISIT, which helps us line 8, in which we check if the adjacent vertex is visited & not the parent of the current vertex then we have a cycle & we terminate the code. Everything else is as same as DFS algo.

## Proof of Correctness :→

So, I am going prove this by bringing in the start-time & finish

time. If there is a back-edge that goes from a vertex of lower finish time to a vertex of a higher finish time, this shows we have a cycle. In my algorithm, we are updating the parent of every vertex and if cycle exists then the vertex next to the current vertex is not the parent but it is visited, this means we have a back edge. Hence Proved. The algorithm stands. The correctness has been referred from theorem 22.10 from the book.

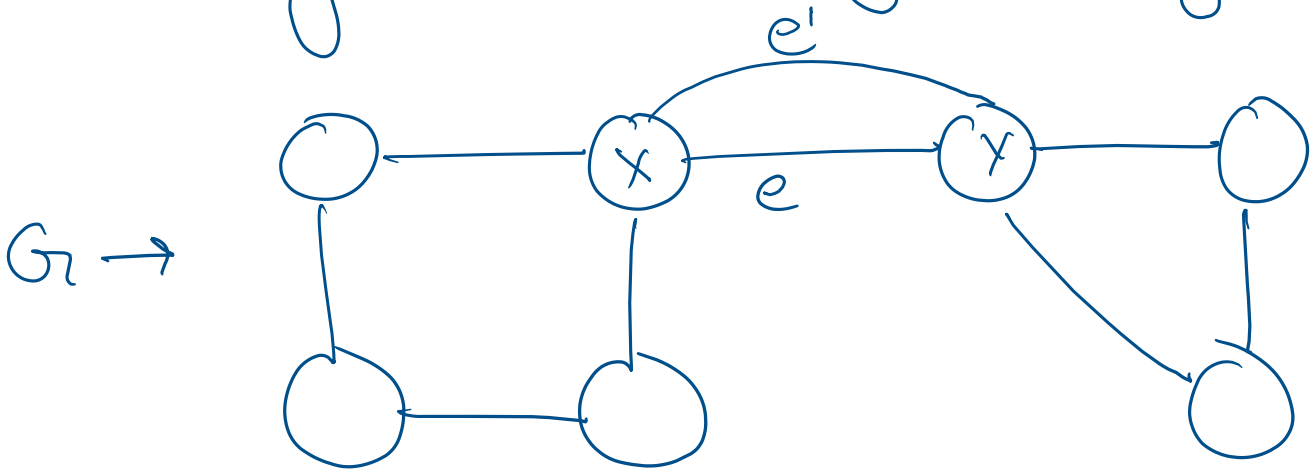
P.T.O→

## Question 2:-

### PART A:→

The first part of this problem I will prove it by contradiction. As in the question it is clearly mentioned that a bridge is an edge, whose removal disconnects the multi-graph G. Suppose we have a graph G, which contains a cycle C, which contains an edge e. Assuming e is a bridge edge. So, now if we remove e from G, the graph G would still be connected by the second edge that connects the vertices u and v, which contradicts the definition of bridge in the question proving that edge is a bridge if

there is no simple cycle C of G which contains e. So, summing up the above conclusion, if there are two vertices x & y, which lies on a simple cycle if and only if there exists one path between x & y that doesnot include the direct path from x to y. So, removing the direct path from x to y doesnot disconnect the graph so x and y is not a bridge. edge.




So, if 4 remove from the above graph G, it is still connected

via the edge e', this $x \rightarrow y$ is not a bridge. Hence proved.

PART-B :→

This part of the problem, I am going to solve it by modifying the DFS algorithm.

```
DFS(G):
1. FOREACH VERTEX U ∈ G.V
2.         PARENT[U] = NIL,
3.         VISIT[U] = FALSE
4. TIME=0 ; DISC[] ; ANCES[]
5. FOREACH VERTEX U ∈ G.V.
6.     IF(VISIT[U] == FALSE):
7.         DFS-VISIT(G,U, VISIT, DISC, ANCES, TIME)

DFS-VISIT(G,U, VISIT, DISC, ANCES, TIME):
1. TIME = TIME + 1
2. VISIT[U] = TRUE.
3. DISC[U] = TIME
```

```
4.  ANCES[U] = TIME
5.  FOR EACH V ∈ G.Adj(U):
6.      IF (VISIT[V] == FALSE):
7.          PARENT[V] = U
8.          DFS-VISIT(G, V, VIST, DISC, ANCES, TIME).
9.          ANCES[U] = MIN(ANCES[U], ANCES[V])
10.         IF (ANCES[V] > DISC[U]):
11              HIDE-EDGE (U → V)
12.             IF (U IS NOT IN G.Adj(V)):
13.                 BRIDGE++
14.             END-IF → END-FOR.
15.         END-IF.
16.     ELSE-IF ( V ≠ PARENT[U]):
17          ANCES[U] = MIN(DISC[V], ANCES[U])
18.     END-IF.
19. END-FOR
20. PRINT(BRIDGE).
```

Describing the above algorithm, I am going to use the normal DFS traversal with a slight modification to calculate

the bridges. So, in the DFS part of the algorithm I am storing the visit to each node, the discover time & the ancestor if there is any based on the discover time in a simple 1-dimensional array. Parent of the node is also stored in an array. Now at the beginning I am initializing parent as well as the visit for that node to null. Then for every node, calling the DFS-VISIT, which is the main part of the algorithm. It recursively calls DFS-VISIT till all the nodes are visited, this works exactly like DFS-traversal, the change comes in at step a in DFS-VISIT. At this point, we check whether the tree rooted at v, has connection

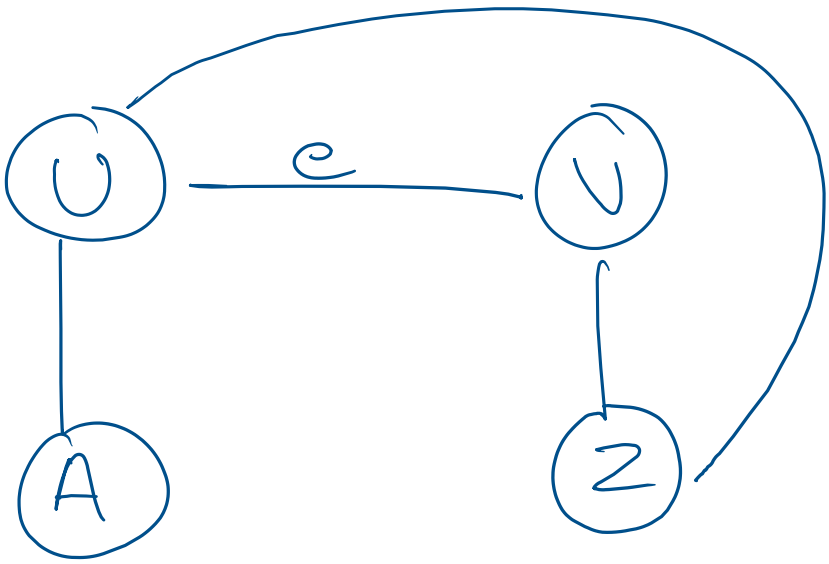
to any ancestor of u. If yes, then we update the ances[u] to the discover time of the ancestor, else it remains the same. Then at step-10, we check for the bridge, that the ances value of v is greater than the discovery value of u, ie v is reachable only after u, then we go to line 11 & hide that edge. After hiding the edge, we check whether there is any adjacent edge which connects $u \rightarrow v$, if no then we increase the counter by 1. At the end we print the counter. This algo, prints all the bridges present in a multi-graph. Hide-edge basically hides that edge so it won't come up as an adj. edge to that vertex.

## Running Time Analysis :→

So, the running time of the above algo, would be similar to that of the DFS traversal. The loop in line 1-3 & 5-7 inside DFS would take exactly $O(V)$, excluding the time used for calling DFS-VISIT. Now taking DFS-VISIT into our analysis, we use the aggregate method to determine the running time. DFS-VISIT would be called for every vertex in G but we also look at the adjacent vertex of v so, it would take $O(E)$. Therefore, the total running time is $O(V+E)$.

## Proof of Correctness :→

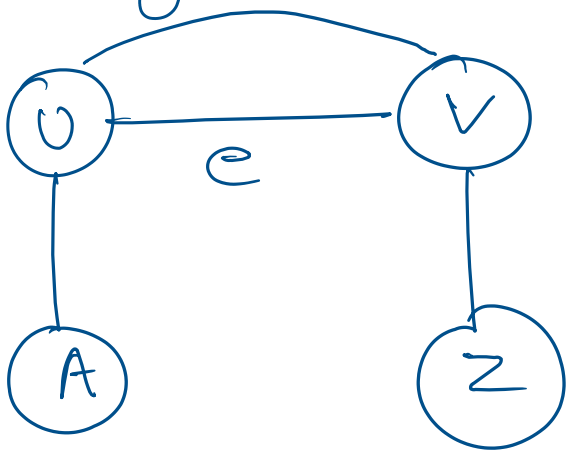
The proof of correctness would be similar to that of DFS. We can prove that the algo is correct by contradiction. Suppose we say an edge e is a bridge edge in the graph below.




So, the DFS would visit all the edges & keep on updating the disc & ances for all the vertices.

It checks for the condition ANCES[v] > disc [v] which would never be true & prints 0 as the output. So, hence proved.

Case 2:- As we can a have parallel edges in multigraph.




In this case we again assume e is a bridge edge. DFS will traverse through the whole graph & it would even enter the condition ANCES[v] > disc[v], but in that condition

we hide the edge ie $u \rightarrow v$ & check whether $v$ is still in the adj. list of $v$. If yes, then e is not a bridge edge and count remains same. Hence Proved.

Question 3:→

The idea is to reduce the 'E to E' that only connects the different strongly connected components in the graph.

Pseudo code:→

1. Call DFS to compute finishing time for each vertex $v$.
2. Compute $G^T$
3. Call DFS $(G^T)$ but in the DFS algo we consider the vertices in order of decreasing finish time as

computed in step 1.

④ We get the vertices of each tree in dfs forest formed in line 3 as a separate strongly connected components.

⑤ Now using the property of the component graph, $G^{scc} = (V^{scc}, E^{scc})$, which is defined as follows in CLRS. Suppose that $G$ has strongly components $C_1, C_2, \ldots C_k$. The vertex $V^{scc}$ is $\{v_1, v_2, \ldots v_k\}$ and it contains a vertex $v_i$ for each strongly connected components. $C_i$ of $G$. There is a edge $(v_i, v_j) \in E^{scc}$ if $G$ contains a directed edge $(x, y)$ for some $x \in C_i$ and some

$y \in C_j$. By contracting all edges or reducing all edges that are in the same strongly connected components of G, the resulting graph is $G^{scc}$.

(6) Now as, we have $G^{scc}$. We start with $E' = NULL$

(7) For each scc in graph G, take the vertices in the scc be $v_1, v_2 \cdots v_m$ and add to $E'$. The directed edges $\{v_1, v_2\}, \{v_2, v_4\}$ etc. forms a directed cycle that includes all the vertices of the scc.

(8). Now for each edge $(s, t)$ in $G^{scc}$ select any vertex $u$ in $s \rightarrow scc$ and similarly any vertex $v$ in $t \rightarrow scc$ and add that edge (with direction) to $E'$.

The above algorithm can be also achieved by using a topological sort approach as the $G^{SCC}$ the component graph is a DAG proved in lemma 22.13 in the book. So, in this approach we have a DAG and after implementing Topological sort just after line 5 would give us $G' = (V, E')$, which gives the linked list of vertices with the edge and eventually $E'$ is minimized.

Running Time Analysis:→

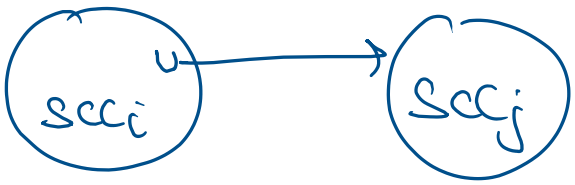
Line 1 to 4 would take $O(V+E)$, line 5 would take also take $O(V+E)$, as we are going through all the vertices and all the adjacent edges to the current vertex, line 6

would just take $O(1)$ time, as we are just setting $E' = \phi$. line 7 would take $O(V)$ as we are going through all the vertices, line 8 would take $O(E)$ time as now we are taking only the edge that matters. If we going for the topological sort approach then the running time would be $O(V+E)$, that is for DFS that is present in topological sort algorithm. So, the total running time would be $O(V+E)$.

Proof of Correctness :→

There are two connected components in a graph G, which are adjacent to each other, they are SCC; &

$scc_j$, they have edge $(v_i, v_j)$ of $G^T$ with $v_i$ is present in $scc_i$ & $v_j$ is present in $scc_j$, these will also be there in G. Now, suppose we have one vertex u in $scc_i$ which connects to $scc_j$.




So, if we go through the topological sort approach the discovery time of u would be smaller than the discovery time of the first vertex in $scc_j$, thus being the only edge connecting two components & this

will only be present in E' & G'. We get $SCC_i$ by contracting all the incident vertices in that component. Now suppose, we union $SCC_i$ & $SCC_j$ and then v would have the largest finish time. So, taking the approach in which we create $G^{SCC}$ & then add edges that connect two different components would work. Hence Proved.

## Question 4:→

So, in the problem after going through lot of approaches (Floyd-warshall, Bellman Ford), I am going to solve this using the bellman ford algorithm. As, we have two edges u to v as source and destination.

The pseudo-code is as follows:→

G → Graph
S → source
w → weight
t → destination
d → distance
π → parent
COUNT → Array to hold the count.

BELLMAN-FORD-SHORTEST-COUNT (G, S, w, t):

1. FOR EACH VERTEX V ∈ G.V:
2. $V.d = \infty$
3. $V.\pi = NIL$
4. $S.d = 0$

```
5  COUNT[][] → initialize 0 to G.V x G.V global mat.
6  FOR i = 1 to |G.V| - 1 :
7      FOR each edge (u,v) ∈ G.E :
8          RELAX (u, v, w, s, COUNT)
9  FOR each edge (u,v) ∈ G.E :
10     IF v.d > u.d + w(u,v)
11         RETURN FALSE.
12 PRINT-COUNT (COUNT, s, t)
13 RETURN TRUE.
```

RELAX (u, v, w, s, COUNT):

```
1  IF v.d > u.d + w(u,v) :
2      v.d = u.d + w(u,v)
3      v.π = u
4      COUNT[s][v] = 1
5  IF v.d == u.d + w(u,v) :
6      COUNT[s][v] = COUNT[s][v] + 1
```

PRINT (COUNT, s, t) :-

① PRINT (COUNT[s][t])

So, describing the above algorithm, I am initializing the parent and the distance of all the vertices in graph G. The source distance is set to 0, the source given by the user. Then I am initializing a global matrix count = 0 for all the indexes. Line 6 - 8 loops through vertices in G as well as edges in E. I am then calling the relax method where the main modification is done. I am passing the source as well as the count matrix with u, v & w as the parameter to this function.

In the relax function we check for each vertex from the source, if there is a shortest path from source, if yes then we initialized count from source to that vertex v as 1. The second if condition in line 5 of relax checks if the weight of two paths is same, if yes, then increase the counter for that source & v by 1. Updation happens, till every path from s to all the vertices is checked. Line 9-11 checks for the negative cycle if any. Line 12 calls print which prints the value of count for that source

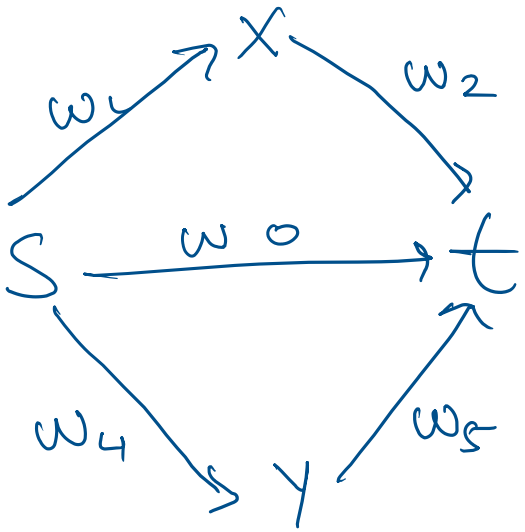
to destination.

Time Complexity:→

The time complexity of the above algorithm will be same as that of Bellman-Ford Algorithm. The main part of the algorithm runs from 6 to 8; traversing all vertices and edges thus taking O(VxE) time.

Proof of Correctness:→

For this proove, suppose we have source s & a destination edge t. The proof of Bellman-Ford will remain the same as in the text-book. The only change in the algorithm will be shown by the graph below.




Suppose $w_1 + w_2 = w_0$.

As the question says graph, therefore, there cannot be parallel edges (as multigraph has parallel edges). So, we have 3 paths to reach from $s \to t$. We traverse to t from direct edge $s \to t$ and update $t.d = w_0$, & $count[s][t] \to 1$. Next we look at x update $x.d \to w_1$ $count[s][x] \to 1$., similarly for $d.y = w_4$ & $count[s][y] \to 1$. Now from $x \to t$. & we have assumed,

$w_1 + w_2 = w_0$, so this would enter the condition 5 in relax function and update the count from $s \to t = 2$.

So, generalizing the above if there are n paths with the same weight $w_0$ (from $s \to t$), then the count would be n.