Assignment 4

214101019

- Some Terminologies:
    1. Nodes are numbered from 1 to n
    2. Adjacency List is used for the implementation
    3. Simple and Parallel edges are used as it is
- Input 1 :

11 17

1 2 1

1 4 1

2 3 1

2 5 1

3 1 1

3 7 1

4 3 1

5 6 1

5 7 1

6 7 1

6 8 1

6 9 1

9 10 1

7 5 1

8 10 1

9 10 1

10 9 1

- Input 2:

4 5
1 2 3
2 3 1
3 4 1
4 1 2
4 2 1
1 3 1

---

1. **Depth First Search**:

- Node: contains value => to store its key value

  Start => to store the discovery time

  Finish => to store the finish time

  Pred => to store the predecessor

  Adjacent => vector to store the nodes adjacent to the node

- Color Array : used to store the color of each node

  White (w) => denotes that the node is not discovered yet

  Black(b) => denotes that node is finished exploring

  Gray (g) => denotes that the node is being explored

- **Algorithm DFS():**
  1. It is used to select a node 1 by default
  2. And after a ends dfs_visit(u)
  3. This algorithm helps in selecting that node which is not discovered yet
  4. That is it select the node who has color white.

- **Algorithm DFS_visit(int u):**
  1. We start with node 1 default
  2. Each time when a node is being discovered
  3. Push the node into the stack; set its color to gray ; Increment time and set start time of the node to time ; also set predecessor to the top node of the stack
  4. Now check its adjacent nodes
  5. If its adjacent node is not discovered yet then call dfs function on this node
  6. Once all its adjacent nodes are explored i.e. all adjacent node's color is black then set color of the node to black ; increment time value and set finish time to this time;pop node from the stack.

- **Edges:**

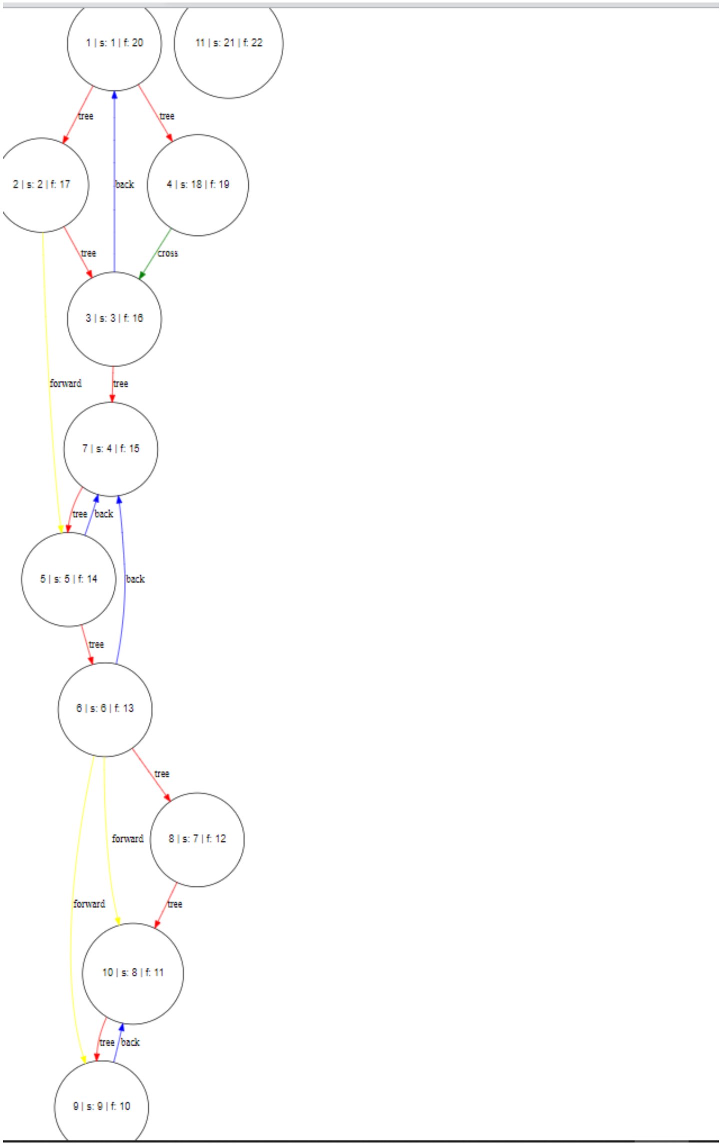
1. Tree Edge:
    a. It is an edge which is present in the tree obtained after applying DFS on the graph.
    b. For edge u,v if start[u] < start[v] and finish[v] < finish[v]
    c. Then we check the predecessor of u
    d. If it is v then mark edge (u,v) as tree edge

2. Forward Edge:
    a. It is an edge (u, v) such that v is descendant but not part of the DFS tree.
    b. If start[u] < start[v] and finish[v]<finish[u]
    c. Then we check the predecessor of v if it is not equal to u
    d. then the edge is a forward edge.
3. Back edge:
    a. It is an edge (u, v) such that v is ancestor of node u but not part of DFS tree.
    b. Presence of back edge indicates a cycle in directed graph.
    c. For edge(u,v) if start[v]<start[u] and finish[u]<finish[v] then the edge is a back edge

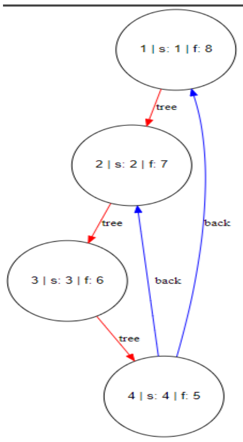
4. Cross Edge:
    a. It is a edge which connects two nodes such that they do not have any ancestor and a descendant relationship between them.
    b. For an edge(u,v) if none of the above conditions are satisfies then the edge is a cross edge

- **Time Complexity:**
    1. As for each node its adjacent node is traversed or checked
    2. Thus total time required is **O(V + E)** as adjacency list is used.

Output :



Output 2:



----

2. Tarjan

- Tarjan algorithm helps in finding out the Strongly Connected Component in the graph using just one DFS traversal that it needs only O(V+E) time.
- Node: contains value => to store its key value

  Start => to store the discovery time

  Low => to store the minimum id of the node reachable from it.

  id => to store the id of the node assigned during the dfs traversal

  Adjacent => vector to store the nodes adjacent to the node

- Stack invariant:

  Tarjan's algorithm maintains a set (often a stack) of valid nodes from which to update low-link (low value) .

  Nodes are added to the stack [set] of valid nodes as they're explored for the first time.

  Nodes are removed from the stack [set] each time a complete SCC is found.

- New low-link update condition

    If u and v are nodes in a graph and we are currently exploring u then our new low-link update condition is

    To update node u’s low-link value to node v’s low-link value there has to be path of edges from u to v and node v must be on the stack.

- Array in_seen

    It is used denote whether a particular node i at index i of this array node is present in the stack or not

- Array processed

    It is used denote whether a particular node i at index i of this array node is processed or not

- **Function SCC_Compute:**

  1. It checks for a node who is not processed yet
  2. If node is not processed yet then it applies SCC_Dfs on it

- **Function SCC_Dfs:**

  1. Here as we visit a node for the first time increment id and assign this incremented id to it .
  2. Mark current node as visited and add them to the stack st .
  3. Now call SCC_dfs on its adjacent nodes if not processed yet.
  4. On SCC_ dfs call back, if the previous node is on the stack then min the current node’s low-link value with the last nodes’s low-link value.

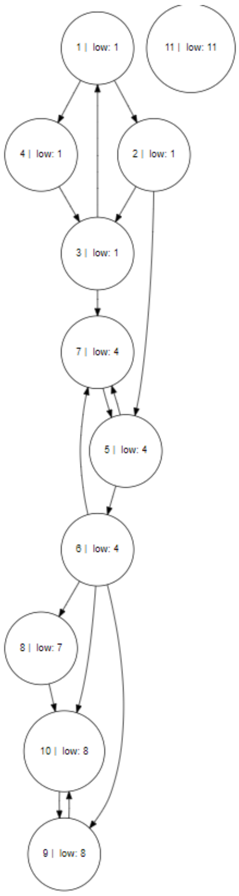
5. After visiting all neighbours, if the current node started a connect component that is if the low-link value of the current node is same as its id
6. Then pop nodes off the stack st until current node is reached and all the nodes popped off will have low-link value equal to that of the current node.
7. This allows low-link values to propagate throughout the cycles.
8. So all the nodes which have same low-link value belongs to same SCC.

- **Time Complexity:**

1. It performs dfs only once and for the rest of the updations and popping and pushing into the stack it need only O(V)
2. Time Required = **O(V + E)**

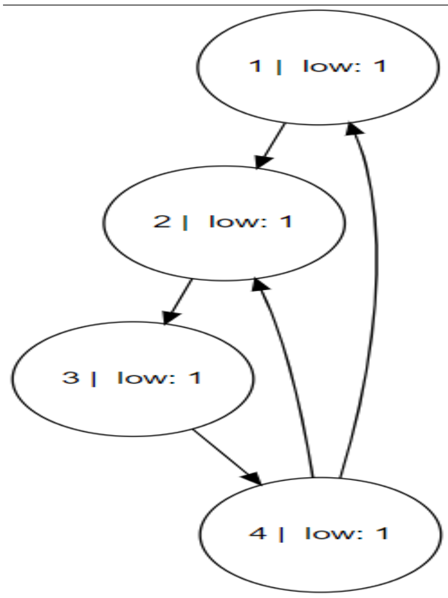
Output 1:

```
Implementing Tarjan

Node(s) in SCC 1 are : 9 10

Node(s) in SCC 2 are : 8

Node(s) in SCC 3 are : 6 5 7

Node(s) in SCC 4 are : 4 3 2 1

Node(s) in SCC 5 are : 11

Total number of SCC = 5
```



Output 2:

```
Node(s) in SCC 1 are : 4 3 2 1

Total number of SCC = 1
```



----

3. **Component and Minimised Graph:**

- Consider a directed graph G = (V, E), we need to construct another graph G'= (V, E'), where E' is a subset of E, such that
- (a) G' has the same strongly connected components as G,
- (b) G' has the same component graph as G, and
- (c) E' is as small as possible.
- Inorder to achieve the above goal we first obtain the Strongly Connected Component using tarjan algorithm.
- We maintain a vector<vector>> named scc_nodes to store the SCCs
- In scc_nodes each row 'r' represents an SCC and each row contains the nodes in the SCC 'r'.
- Now we get two types of edges :
  1. Edges Lying entirely inside an SCC 2. Edges Between 2 SCCs.
- We first reduce the edges going from one component to another
- For that if there are multiple edges going between the nodes belonging to different SCCs a and b then we consider only one edge between component a and b.
- For the edges lying inside a single component we check that the edge under consideration is a strong bridge or not that is removal of this edge increases the number of strongly connected components.
- If the edge under consideration is a strong bridge then we donot remove it
- However, if it is not a strong bridge then we remove it.

To check if edge is a **strong bridge** or not using **reachable(u,v) function**:

1. Here inorder to check if the edge (u,v) is a strong bridge or not we perform the following steps:
2. Remove edge (u,v)

3. Perform call to reachable (u,v) and reachable(v,u)
4. **Reachable(u,v)** performs **bfs** starting from **u** and **checks if after removal of edge(u,v) v is reachable from u or not** if **reachable** then it **returns true** else it **returns false.**
5. Reachable function uses a queue
6. For that it starts from u and pushes u into the queue
7. Now while q is not empty
8. It dequeues the front node from the queue; sets it as processed and then checks for all its adjacent nodes:
9. if the adjacent node's value is not equal to v and that node is not traversed/processed yet then push it into the queue
10. else if the value is equal to v then it indicates that there is another path to reach from u to v hence it returns true.

revise() function:

1. we maintain an array belongs_to of size V ; in which ith index stores the SCC in which node i is present.
2. We maintain is_edge matrix which of the size #scc x #scc.
3. Initially all entries is 0 in this.
4. To add edges between components
5. For each node u we check its adjacent node v
6. If node u belongs to component a and node v belongs to component b then we check if an edge exists between component a and b i.e. is_edge[a][b] = 1
7. If it is 1 then no need to consider this edge as already an edge has been added between them
8. Else the edge (u,v) is considered

Incomponent() function:

1. it performs the steps described previously
2. for each scc for each node u in it and its adjacent node v
3. edge (u,v) is removed
4. reachable (u,v) and reachable(v,u) is called
5. if either of them returns false then it means no other path is possible from u to v or v to u and because we need to maintain the scc property hence we cannot remove this edge

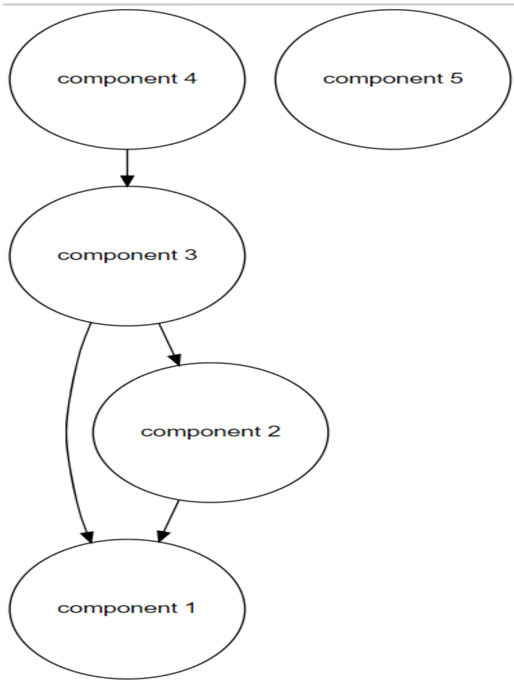
6. So, we need to compulsorily consider edge(u,v)

Time Complexity:

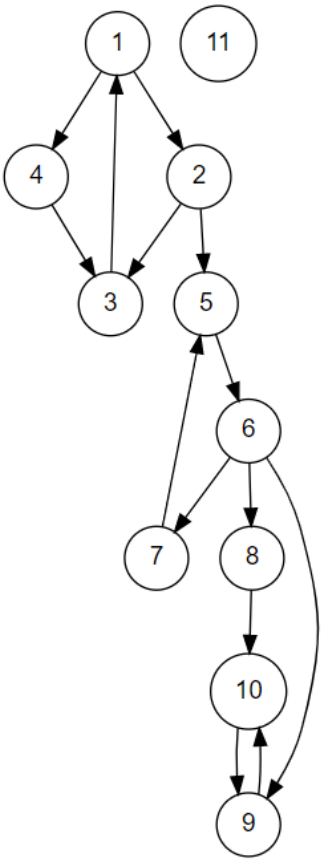
1. To perform tarjan algorithm we need O(V+E) time
2. For construction of belongs_to we traverse the scc_nodes thus we need need O(V + V) = O(V) time.
3. To add edges between two components we construct is_edge matrix =O(V*V)
4. To identify edges between two components we traverse the adjacency list = O(V+E)
5. To identify edges inside a scc we perform bfs on each edge so in total for all scc = O(E*(V+E))
6. Total time = **O(E*(V+E))**

Output :

Component Graph

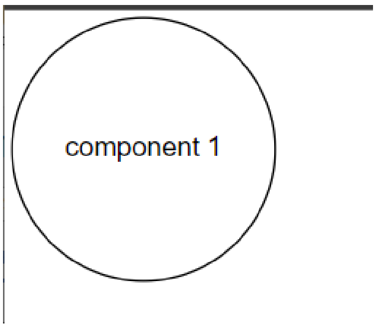


Minimised Graph

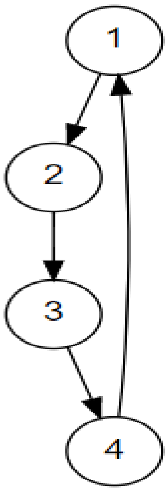


Output 2:

Component graph



Minimised graph



---

4. **Semi-Connected:**

- A directed graph G = (V, E) is Semi-connected if, for all pairs of vertices u, v in V, we have either a path u ---> v or a path v ---> u.
- To check if graph is semi-connected or not
- We first obtain the Strongly Connected Component using tarjan algorithm.
- Now, we construct the component graph where each node is represents a SCC and each edge corresponds to the edge(s) going from one component to the another in the original graph.
- Now, all the nodes inside a particular SCC will be connected so we just need to show that if the nodes in the component graph satisfies the condition
- That no more than 1 node has indegree equal to 0 then we can state that the graph is Semi-connected

- This is because the component graph will surely not form a cycle.
- So, if more one node in the component graph has indegree 0 then it simply implies the fact that the there is no path exists between the SCCs with indegree 0.

Revise_Adjacency() function:

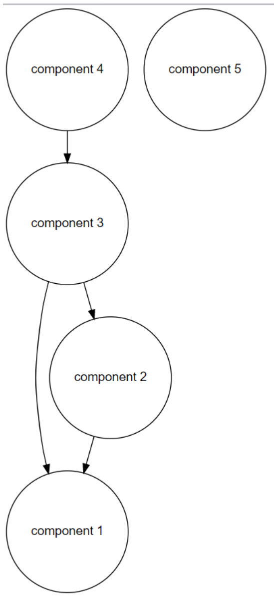
1. We obtain the SCCs using tarjan algorithm and store it in vector<vector>> name scc_nodes.
2. Each row ‘r’ of scc_nodes represents a SCC and it contains the nodes present in this SCC
3. Now, we maintain an array belongs_to of size V ; in which ith index stores the SCC in which node i is present.
4. We also maintain an adjacency list called adjList_rev which stores the adjacency list for the component graph.
5. Now, for each edge(u,v) we check that
6. if the belongs_to[u] == belongs_to[v] then they belong to same component and no need to add any edge.
7. If belongs_to[u] != belongs_to[v] then they belong to different component
8. We maintain a set and add belongs_to[v] of all such adjacent nodes in the set
9. Later, for a particular scc ‘r’ when for all the nodes in the scc the above steps are performed then we for SCC the elements of the set as adjacent nodes for node r in the component graph.
10. Note that set ensures no repeated edges are added between component.

Time Complexity:

7. To perform tarjan algorithm we need O(V+E) time
8. For construction of belongs_to we traverse the scc_nodes thus we need need O(V + V) = O(V) time.
9. To construct the adjacency list for the component graph we traverse all the edges and check the belongs_to value so we need O(V+E) time.
10. Total time = **O(V+E)**

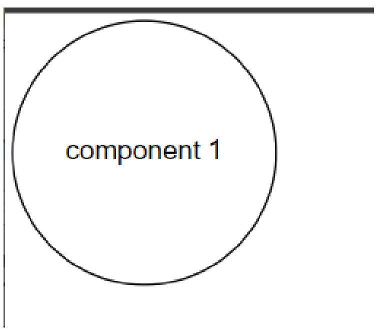
Output:

```
Checking if graph is semi connected or not

Edges in condensation graph are:

edge : 2 -> 1
edge : 3 -> 1
edge : 3 -> 2
edge : 4 -> 3
--------------------------------------------------------------------------------

THE INPUTTED GRAPH IS NOT SEMI CONNECTED
```



Output 2:

```
THE INPUTTED GRAPH IS SEMI CONNECTED
```



----

5. **Dijkstra :**

- Dijkstra is a Greedy algorithm which finds the shortest path from the source node provided as input.
- It works only on Graph with positive edge weights.
- Here binary_heap is used for min _priority queue implementation
- Node:
    - contains key => to store the node value
    - pred => to store the predecessor of the node
    - dis => to store the distance of the node from the source node
  - Distance of the node
    - Initially distance is set to a larger value
    - Predecessor is 0 by default for each node

  - Queue in_queue

    It is used denote whether a particular node i at index i of this array node is present in the queue or not

  - Array processed

    It is used denote whether a particular node i at index i of this array node is processed or not

Relaxation Function:

1. Here for an edge (u,v)
2. We check dis[v] and (dis[u] + w[u][v])
3. That is we check that is it possible that the distance of node v from the source reduces if we consider the path via u
4. If yes then we set predecessor of node v to node u and update its distance value.
5. Else nothing is updated

Priority_Queue using Binary Heap:

1. Here min priority_queue is implemented using binary heap

2. For binary heap insertion and deletion and decrease and heapify requires O(log n) time.
3. In Dijkstra we maintain binary heap of size = #Vertices
4. And here for each edge reachable from the source node relaxation is called which results in decrease key operation which can also be calculated in O(logV) time
5. Hence for E edge we need O(E*logV) time

Function Dijkstra():

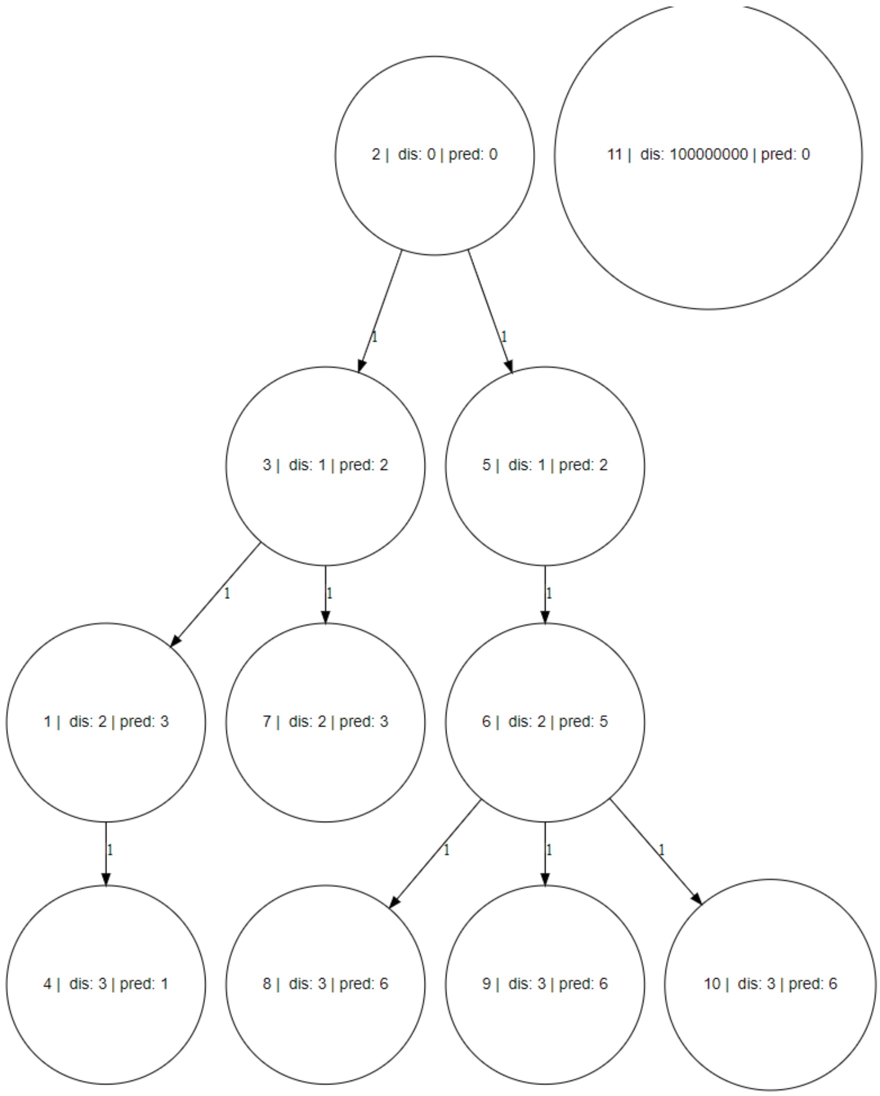
1. Here the user is first prompted to enter the start node that is the source vertex.
2. The source vertex is now pushed into the queue
3. While q is not empty we perform the following:
4. Extract the min node(root node) form the priority queue and set processed to true.
5. We check its adjacent nodes and check if the node is processed or not
6. If it is not processed then we check if relaxation is possible or not.
7. If relaxation is possible then we update the predecessor and distance values
8. Now we push the node into the queue and set in_queue to true.

Time Complexity:

1. Here we perform initialization which need O(V) time
2. Later we perform Extract min V time which needs O(VlogV)
3. And for updating the distance values and pushing them in he queue it needs O(ElogV) time
4. Total time = O(V + ElogV + VlogV) = **O(E*logV)**

Ouptut 1:

Source = 2



Output 2:

Source = 4