08 03 2024
Friday

* **Bellman Ford Algorithm –**

This algorithm is used for finding shortest path and to detect negative weight cycles in graph.

There is just one step –

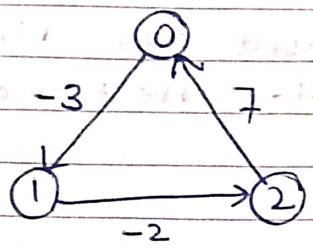
- **Relaxation** – Relax all the edges in graph N-1 times where N is no. of nodes in graph.

Relaxation step – `if (dist[u] + cot < dist[v]){`
`//update shortest path`
`}`

Why N-1 times ?

We can iterate all the edges in any order and relax them. for the same graph, if we consider all the edges in different order, in the worst case, we will get the shortest path after N-1 iterations.

let the graph be –



Edges order →

0 → 1
1 → 2
2 → 0

initially, distance array =

0	∞	∞
0	1	2

So, here are 3 nodes. No. of iterations or relaxations will be N-1 i.e. 2.

Relaxing Ist time → (0→1) dist[0] + (-3) < dist[1]
-3 < ∞ ✓

(1→2) dist[1] + (-2) < dist[2]
-3-2 < ∞ = -5 < ∞ ✓

(2→0) dist[2] + (7) < dist[0]
-5 + 7 < 0
2 < 0 ✗

0	∞ -3	∞ -5
0	1	2

Relaxing 2nd time → Nothing will change in distance array after 2nd relaxation.

Let for the same graph, edges order is -

Edges order →
2 → 0
1 → 2
0 → 1

initially, distance array = | 0 | ∞ | ∞ |
0 1 2

Relaxing Ist time -

(2 → 0) dist[2] + (7) < 0
∞ + 7 < 0
∞ < 0 ✗

(1 → 2) dist[1] + (-2) < ∞
∞ < ∞ ✗

(0 → 1) dist[0] + (-3) < ∞
0 - 3 < ∞ ✓

distance = | 0 | ∞ -3 | ∞ |
0 1 2

Relaxing 2nd time -

(2 → 0) dist[2] + 7 < 0
∞ < 0 ✗

(1 → 2) dist[1] + (-2) < ∞
-3 - 2 < ∞ ✓

(0 → 1) dist[0] + (-3) < -3
0-3 < -3 ✗

distance = | 0 | -3 | ∞ -5 |
0 1 2

So, when edges order were → (0→1), (1→2), (2→0)
It took only 1 relaxation to find shortest path.
When edges order were → (2→0), (1→2), (0→1)
It took 2 relaxations to find shortest path.

So, we can say that, it's the edges order which decide the number of relaxations required but in worst case, no. of relaxations would be N-1.
Conclusion is, after N-1 relaxations, no matter what the edges order is, we will get the shortest path.

= Detecting negative weight cycles in graph-

After N-1 relaxations, relax the edges one more time. If during Nth relaxation, the distance array gets updated, it means negative weight cycle present otherwise not.

Code-

```
void bellManford (int n, T src){
//create distance vector for storing shortest distance
vector<int> dist (n, INT_MAX);
//initialise src distance with 0
 dist [src] = 0;

//relax edge list n-1 times
for (int i=1; i<n; i++) {

      //traverse on entire edge list
      for (auto it: adjList){
       for (auto nbr : it.second){
          T u = it.first;
          T v = nbr.first;
          int wt = nbr.second;
          //relaxation step
          if (dist[u] != INT_MAX &&
             dist[u]+ wt < dist[v]){
              dist[v]= dist[u] + wt;
          }
       }
    }
}
//now, distance array is ready, ek baar or
//relaxation krke saari edges ko check kr lenge
// if negative weight cycle present. If distance array
// update hoti he, it means cycle present.

bool anyUpdate = 0;
for (auto it: adjlist){
  for (auto nbr : it.second){
```

```
            T u = it.first;
            T v = nbr.first;
            int wt = nbr.second;

            if(dist[u] + wt < dist[v]){
                anyUpdate = true;
                break;
            }
        }
    }
}

if(anyUpdate == 1){
    cout << "negative cycle present" << endl;
}
else{
    cout << "negative cycle absent" << endl;

    for(auto i: dist){
        cout << i << " ";
    }
}
}
```

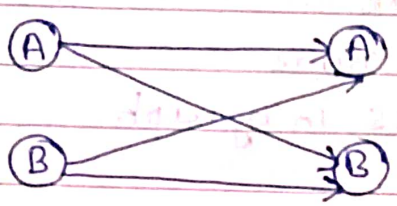
Time complexity → O(n * m)

n = no. of nodes.
m = no. of edges

* **Floyd- Warshall Algorithm →**

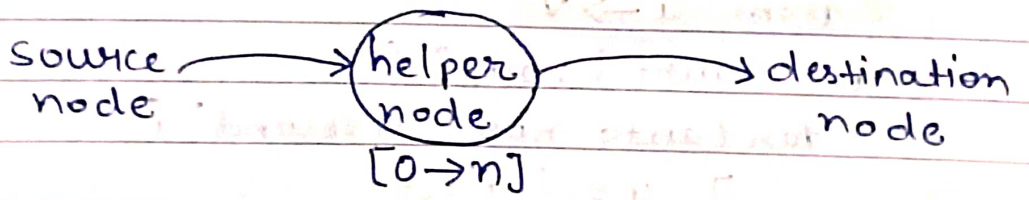
This algorithm is used to find multi-source shortest path (mssp). Will work for both directed and undirected graphs.

Let we have two nodes -



Considering A as source, find shortest path to A and B. Considering B as source, find shortest path to A and B.

So, what we have to do is to consider all nodes as source node and all nodes as destination node and then find shortest path from src to destination node.



helper node → Is the node used to reach destination node via source node. Will be considering all nodes in graph as helper node.

```
for helper 0 → n
    for src  0 → n
        for dest 0 → n
            dist[src][dest] = min(dist[src][dest],
                                  dist[src][helper] + dist[helper][dest]
```

helper = helper node
src = source node
dest = destination node
n = total nodes in graph

Code-

```
void floydWarshall(int n) {

vector<vector<int>>dist(n, vector<int>(n, leg));

//src → src ka distance 0 hoga
for(int i=0; i<n; i++) {
    dist[i][i]=0;
}

// some weights are given in adjacency list
// from u → v
for(auto i : adj) {
    for(auto nbr : i.second) {
        T u = i.first;
        T v = nbr.first;
        int wt= nbr.second;
        dist[u][v]= wt;
    }
}
```

```
//main logic of floyd warshall

for(int helper=0; helper<n; helper++){
 for(int src=0; src<n; src++){
  for(int dest=0; dest<n; dest++){
    dist[src][dest] = min(dist[src][dest],
                          dist[src][helper] + dist[helper][dest]);
  }
 }
}

// distance array is ready
for(auto i: dist){
  for(auto j: i){
    cout<<j<<" ";
  }
  cout<<endl;
}
```

Time complexity - $O(n^3)$

Note - distance (dist) vector ko initialise le9 se kiya bcoz agr INT_MAX se krte to integer overflow ho jata. Agr src se helper ya helper se destination ka path nhi hoga to vo INT_MAX hoga or INT_MAX me kuch add krdenge to vo overflow kr jayega. That's why le9 de liya so that extra condition na lgani pde.

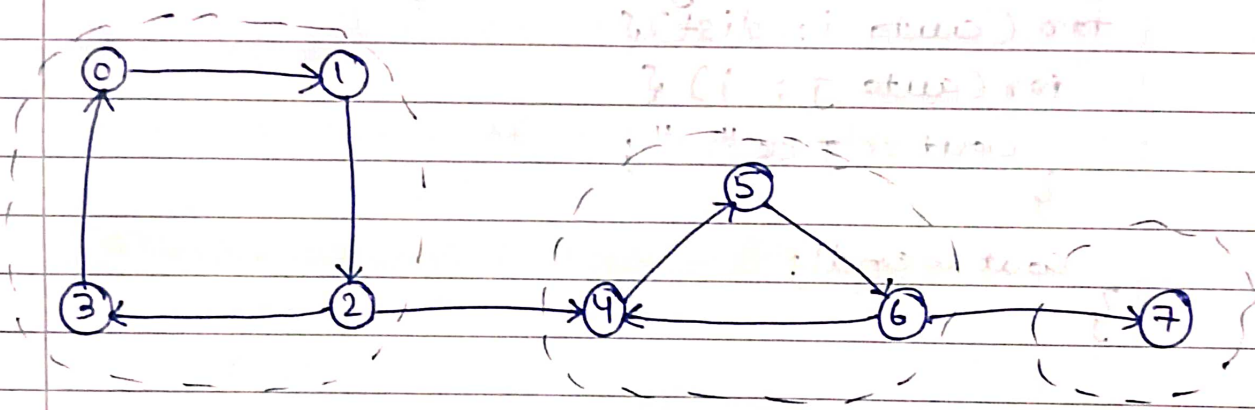
* Strongly Connected Components (Kosaraju's algo) -

This algorithm is used to find the number of strongly connected components in graph. This works only on directed graphs.

Strongly connected component -

If a node x is reachable from node y, then node y must be reachable from node node x for them to be strongly connected.

Let the graph -

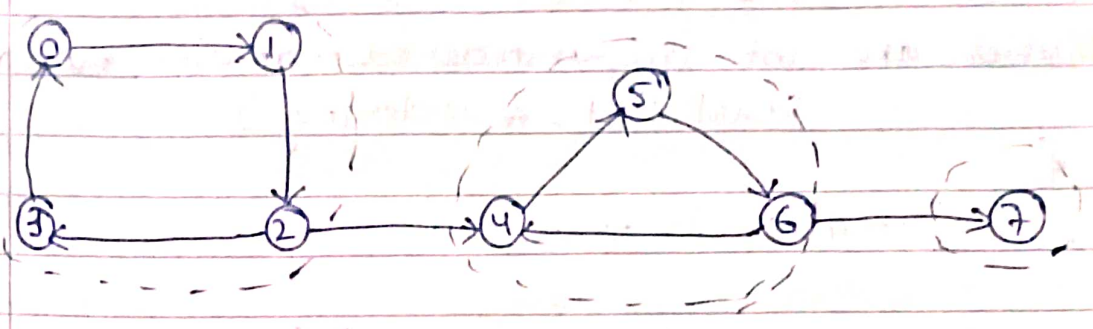


There are 3 SCC's in this graph.

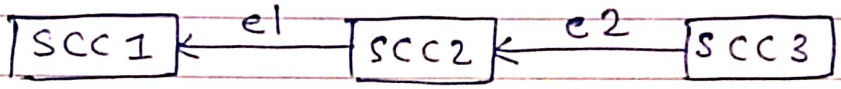
Algorithm -

(1) Find ordering using DFS,
(2) Reverse all the edges of graph.
(3) Count number of strongly connected components.

Why reversing edges? To disconnect SCC's



So, if we start from node 0, SCC2 is reachable from SCC1 via e1 and SCC3 is reachable from SCC2 via e2.



Now, SCC2 is not reachable from SCC1 and SCC3 is not reachable from SCC2.

This is done by reversing the edges. Pr hume nhi pta by which edges, these SCC's are connected. So, hum saare edges reverse kr denge.

Why finding order?

→ Now, edges reverse krne ke baad bhi agr SCC3 se start kre to ek hi flow me SCC2 & SCC1 traverse ho jayenge via e2 and e1 respectively so, SCC's ka count 1 aa jayega jbki there are 3 SCC's. To prevent that, we will find an ordering using DFS (kind of topo sort but not exactly). This ordering makes sure ya to hum current SCC me traverse kr rhe he ya fir usme jo already visited he & that will cause no harm.

// for finding order

Code –

```
void dfs (int src, unordered_map <int, bool>&visited,
          stack <int > & ordering )
{
   visited[src]= true;

   for (auto nbr : adj[src]){
      if (!visited [nbr]){
         dfs (nbr, visited, ordering);
      }
   }
   ordering. push_back (src);
}
```

// To count SCC's

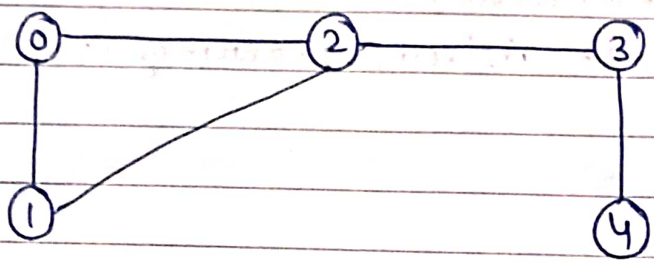
```
void dfs2 (int src, unordered_map<int, bool>&visited2,
           unordered-map<int, list<int>>& adjNew)
{
   visited2 [src]= true;

   for(auto nbr : adjNew[src]){
      if (!visited2 [nbr]){
         dfs2(nbr, visited2, adjNew);
      }
   }
}
```

Time complexity - O(V+E)

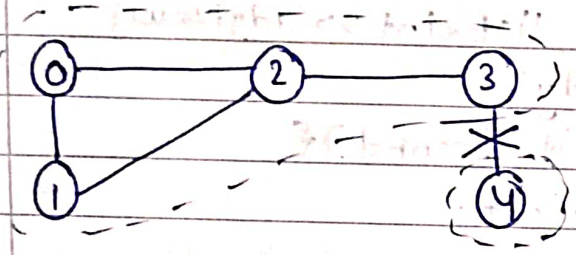
```
int kosaRajuAlgoSCC (int n) {

  //step 1: find ordering
  unordered_map<int, bool> visited;
  stack<int> ordering; // for storing order

  for (int node = 0; node < n; node++) {
    if (!visited[node]) {
      dfs (node, visited, ordering);
    }
  }

  // step 2 - reverse edges
  unordered_map <int, list<int>> adjNew;
  for (auto it : adj) {
    for (auto nbr : it.second) {
        int u = it.first;
        int v = nbr;
        adjNew[v].push_back (u);
    }
  }

  // traverse stack ordering & count SCC's
  unordered_map <int, bool> visited2;
  int count = 0;

  while (!ordering.empty()) {
    int topNode = ordering.top();
    ordering.pop();
    if (!visited[topNode]) {
        dfs2(topNode, visited2, adjNew);
        count++;
    }} return count; }
```

* **Bridge in a graph (Tarjan's Algorithm)**
Critical connections in a network (Leetcode - 1192)

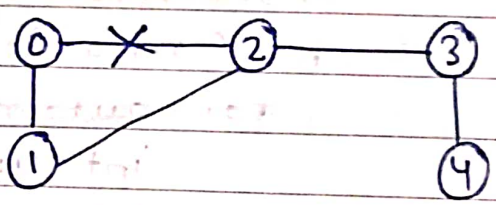
Bridge - Those edges in graph whose removal increases number of disconnected components in graph is a bridge.



no. of components = 1



Removal of edge b/w node 3 and 4 increment no. of components to 2. So, this is a bridge i.e. [3 → 4]



Removal of edge b/w node 0 and 2 doesn't affected no. of components as no node gets disconnected. So, this is not a bridge i.e [0 → 2]

Brute force approach - Remove every edge one by one from graph and count no. of disconnected components everytime. Time complexity using this approach is $O(E) \times O(V+E)$ i.e. quadratic. That's why using tarzan's algorithm.

Tarjan's algorithm -

- Time of insertion (tin) - During DFS calls, tin is the time when a node is visited for the first time.

- lowest time of insertion (low) - It tracks if any other path exists for a node other than parent node. Agr low update ho jata he, it means particular node tk pahochne ka or koi path bhi exist krta he. It means bridge nhi he.

⇒ tin ek baar set ho gya to update nhi hoga. low update ho skta he. We will make an array for tin and for low. These arrays contains tin and low for the corresponding nodes.

* low updation -

```
low[src] = min(low[src], low[nbr])
```

* Bridge testing condition -

```
if (low[nbr] > tin[src]) {
    // edge is bridge
}
```

Algorithm -

(1) Declare variable timer which will keep track of time of insertion (tin) and low. Initially, timer can be .0, 1, 2 ... anything. Also, initially, (tin == low). Start DFS.

(2) Go to neighbours of node -

- if (nbr == parent) {
  // skip the iteration
  }

- if (!visited[nbr]) {
  // dfs call
  // low updation
  // bridge testing
  }
- if (visited[nbr]) {
  // low updation // no need of bridge testing
  }

Note → (nbr == parent) wala case isliye skip kiya because hume check krna he ki parent ke alawa nbr tak aane ka koi path exist krta he ya nhi

→ Agr koi node visited he or parent ke alawa koi node use re-visit kr rhi he. It means us node tk aane ke 2 paths he. So, bridge to nhi hoga. Isliye already visited wale case me bridge testing nhi ki.

Code -

```
class Solution{
public:

void findBridges( int src, int parent,
unordered_map<int, list<int>>& adj, int &timer,
vector<vector<int>>& ans, vector<int>& tin,
vector<int>& low, unordered_map<int,bool>& visited)
{
    visited[src]= true;
    low[src]= tin[src] = ++timer;

    for (auto nbr: adj[src]){
      if (nbr== parent){
          //skip iteration
          continue;
      }
      if (!visited[nbr]){
          //dfs call
          findBridges(nbr, src, adj, timer, ans, tin, low,
                      visited);
          //low updation
          low[src]= min(low[nbr], low[src]);
          // bridge testing
          if (low[nbr]> tin[src]){
            //push src & nbr in 2-D vector
            ans.push_back({src,nbr});
          }
      }
      else{
          //visited but not parent
          //low updation
          low[src]= min( low[src], low[nbr]);
      } } }
```

```
vector<vector<int>> criticalConnections( int n,
 vector<vector<int>>& connections)
{
    unordered_map<int, list<int>> adj;

    for (auto it : connections) {
        int u= it[0];
        int v = it[1];
        adj[u].push_back(v);
        adj[v].push_back(u);
    }

    int timer = 0;
    vector<vector<int>>ans;
    vector<int> tin(n, 0);
    vector<int> low(n, 0);
    unordered_map<int, bool>visited;

    int src=0;
    int parent= -1;

    findBridges(src, parent, adj, timer, ans, tin, low,
                visited);
    return ans;
  }
};
```

Time complexity - $O(V+E)$