DSA Preparation

Part-6

Topic- Graph

All detailed revision of Graph which consists of topic wise revisions and solving problems recently asked in interviews-

Just simplified my experience here...

Hope it goona help you all...

Save this pdf and thanks me later

@himanshu_shekhar16

@himanshushekar

Reference taken form GFG, Leetcode, and InterviewBit

What is Graph in Data Structure and Algorithms?

A Graph is a non-linear data structure consisting of vertices and edges. The vertices are sometimes also referred to as nodes and the edges are lines or arcs that connect any two nodes in the graph. More formally a Graph is composed of a set of vertices(V) and a set of edges(E). The graph is denoted by G(E, V).

Components of a Graph

- **Vertices:** Vertices are the fundamental units of the graph. Sometimes, vertices are also known as vertex or nodes. Every node/vertex can be labeled or unlabelled.
- **Edges:** Edges are drawn or used to connect two nodes of the graph. It can be ordered pair of nodes in a directed graph. Edges can connect any two nodes in any possible way. There are no rules. Sometimes, edges are also known as arcs. Every edge can be labeled/unlabelled.

Graphs are used to solve many real-life problems. Graphs are used to represent networks. The networks may include paths in a city or telephone network or circuit network. Graphs are also used in social networks like linkedIn, Facebook. For example, in Facebook, each person is represented with a vertex(or node). Each node is a structure and contains information like person id, name, gender, locale etc.

DFS and BFS

```
#include <bits/stdc++.h>
using namespace std;
vector<bool> visited;


void dfs(int node, vector<vector<int>> &v)
{
    // preorder
    visited[node] = true;
    cout<<node<<" ";
    vector<int> :: iterator it;
    for(it=v[node].begin(); it!= v[node].end(); it++)
    {
     if(visited[*it] == false)
     {
         dfs(*it, v);
     }
    }
    // postorder
    // cout<<node<<" ";
}


void bfs(int node, vector<vector<int>> &v)
{
    queue<int> pq;
    pq.push(node);
    visited[node] = true;
    while(!pq.empty())
    {
        int t= pq.front();
        pq.pop();
        cout<<t<<" ";
```

```
        vector<int> :: iterator it;
        for( it = v[t].begin(); it!=v[t].end(); it++)
        {
            if(visited[*it]);
            else{
                visited[*it] = true;
                pq.push(*it);
            }
        }
    }
}

int main()
{
    int n, m;
    cin >> n >> m;
    vector<vector<int>> v(n);
    visited = vector<bool> (n, 0);
    for(int i=0; i<m; i++)
    {
        int x, y;
        cin>>x>>y;
        v[x].push_back(y);
        v[y].push_back(x);
    }

    dfs(0, v);
    cout<<endl;
    visited = vector<bool>(n, 0);
    bfs(0, v);

    return 0;
```

```
}
```

Topology sort (Kahn's Algorithm)

```
#include<bits/stdc++.h>
using namespace std;
int main()
{
    int n, m;
    cin>>n>>m;
    vector<vector<int>> ad(n);
    vector<int> indeg(n, 0);
    for(int i=0; i<m; i++)
    {
        int x, y;
        cin>>x>>y;
        ad[x].push_back(y);
        indeg[y]++;
    }

    queue<int> pq;
    for(int i=0; i<n; i++)
    {
        if(indeg[i] == 0)
        pq.push(i);
    }

    while(!pq.empty())
    {
        int t = pq.front();
        pq.pop();
        cout<<t<<" ";
        for(auto it : ad[t])
```

```
        {
            indeg[it]--;
            if(indeg[it] == 0)
            pq.push(it);
        }
    }
    return 0;
}
```

Cycle detection for directed graph
By BFS

```
#include <bits/stdc++.h>
using namespace std;
int main()
{
    int n, m;
    cin >> n >> m;
    vector<vector<int>> v(n);
    // visited = vector<bool>(n, 0);
    vector<int> indeg(n, 0);
    for (int i = 0; i < m; i++)
    {
        int x, y;
        cin >> x >> y;
        v[x].push_back(y);
        indeg[y]++;
    }
    queue<int> pq;
    for (int i = 0; i < n; i++)
    {
        if (indeg[i] == 0)
            pq.push(i);
    }
```

```
    int count = 0;
    while (!pq.empty())
    {
        int t = pq.front();
        count++;
        pq.pop();
        for (auto it : v[t])
        {
            indeg[it]--;
            if (indeg[it] == 0)
                pq.push(it);
        }
    }
    if (count == n)
        return false;

    return true;
    return 0;
}
```

By DFS

```
#include<bits/stdc++.h>
using namespace std;

bool iscycle(int i, vector<vector<int>> &ad, vector<bool> &vis, vector<bool> &st)
{
    st[i] = true;
    if(!vis[i])
    {
        vis[i] = true;
        for(auto it : ad[i])
        {
```

```
            if(!vis[it] && iscycle(it, ad, vis, st))
            return true;

            if(st[it])
            return true;
        }
    }
    st[i] = false;
    return false;
}

int main()
{
    int n, m;
    cin>>n>>m;
    vector<vector<int>> ad(n);
    for(int i=0; i<m; i++)
    {
        int x, y;
        cin>>x>>y;
        ad[x].push_back(y);
    }
    bool cycle = false;
    vector<bool> vis(n, false);
    vector<bool> st(n, false);
    for(int i=0; i<n; i++)
    {
        if(!vis[i] && iscycle(i, ad, vis, st))
        cycle = true;
    }
    if(cycle)
    cout<<"cycle is present"<<endl;
```

```
    else
    cout<<"cycle is not present"<<endl;
    return 0;
}
```

Cycle detection in undirected graphs

```
//dfs*************************


#include<bits/stdc++.h>
using namespace std;


bool iscycle(int i, vector<vector<int>> &ad, vector<bool> &visited, int parent)
{
    visited[i] = true;
    for(auto it : ad[i])
    {
        if(it != parent)
        {
            if(visited[it])
            return true;
            if(!visited[it] && iscycle(it, ad, visited, i))
            return true;
        }
    }
    return false;
}


//bfs*****************************
bool iscycle(vector<vector<int>> &adj, vector<bool> &visited, int i, int parent)
{
    queue<pair<int, int>> pq;
    visited[i] = true;
```

```
    pq.push({i, -1});
    while(!pq.empty())
    {
        pair<int, int> p = pq.front();
        int a = p.first;
        int b = p.second;
        pq.pop();
        for(auto it : adj[a])
        {
            if(it != b)
            {
                if(visited[it])
                return true;
                else
                {
                    visited[it] = true;
                    pq.push({it, a});
                }
            }
        }
    }
    return false;
}

int main()
{
    int n, m;
    cin>>n>>m;
    vector<vector<int>> ad(n);
    for(int i=0; i<m; i++)
    {
        int x,y;
```

```
        cin>>x>>y;
        ad[x].push_back(y);
        ad[y].push_back(x);
    }
    bool cycle = false;
    vector<bool> visited(n, false);
    for(int i=0; i<n; i++)
    {
        if(!visited[i] && iscycle(i, ad, visited, -1))
        cycle = true;
    }
    if(cycle)
    cout<<"cycle is present"<<endl;
    else
    cout<<"cycle is not present"<<endl;
    return 0;
}
```

Component of graphs

```
#include<bits/stdc++.h>
using namespace std;
vector<vector<int>> ad;
vector<bool> visited;
vector<int> component;

int get_comp(int i)
{
    if(visited[i])
    return 0;

    visited[i] = true;
    int ans = 1;
```

```
    for(auto it : ad[i])
    {
        if(!visited[it])
        ans += get_comp(it);
        visited[it] = true;
    }
    return ans;
}
int main()
{
    int n, m;
    cin>>n>>m;
    ad = vector<vector<int>> (n);
    visited = vector<bool> (n, false);
    for(int i=0; i<m; i++)
    {
        int x, y;
        cin>>x>>y;
        ad[x].push_back(y);
        ad[y].push_back(x);
    }

    for(int i=0; i<n; i++)
    {
        if(!visited[i])
        component.push_back(get_comp(i));
    }
    for(auto it : component)
    {
        cout<<it<<" ";
    }
```

```
    cout<<endl;
    return 0;
}
```

Bipartite

```
#include<bits/stdc++.h>
using namespace std;
vector<vector<int>> ad;
vector<bool> visited;
vector<int> col;
bool bipart ;

void color(int i, int curr)
{
    if(col[i] != -1 && col[i] != curr)
    {
        bipart = false;
        return;
    }

    col[i] = curr;
    if(visited[i])
    return;

    visited[i] = true;
    for(auto it : ad[i])
    {
    color(it, curr^1);
    }
}
int main()
{
```

```
    int n, m;
    cin>>n>>m;
    ad = vector<vector<int>> (n);
    visited = vector<bool> (n, false);
    col = vector<int> (n, -1);
    bipart = true;
    for(int i=0; i<m; i++)
    {
        int x, y;
        cin>>x>>y;
        ad[x].push_back(y);
        ad[y].push_back(x);
    }

    for(int i=0; i<n; i++)
    {
        if(!visited[i])
        color(i, 0);
    }

    if(bipart)
    cout<<"graph is bipartite"<<endl;
    else
    cout<<"graph is not bipartite"<<endl;
    return 0;
}
```

Cycle detection by disjoint and union

```
#include<bits/stdc++.h>
using namespace std;
const int N = 1e5 +2;
vector<int> parent(N), sz(N);
```

```
int find_set(int v)
{
    if(parent[v] == v)
    return v;
    return parent[v] = find_set(parent[v]);
}

void union_set(int x, int y)
{
    int a = find_set(x);
    int b = find_set(y);
    if(a != b)
    {
        if(sz[a] < sz[b])
        swap(a, b);

        parent[b] = a;
        sz[a] += sz[b];
    }
}

int main()
{
    for(int i=0; i<N; i++)
    {
      parent[i] = i;
      sz[i] = 1;
    }
    int n, m;
    cin>>n>>m;
    vector<vector<int>> edges;
```

```
    for(int i=0; i<m; i++)
    {
        int x, y;
        cin>>x>>y;
        edges.push_back({x, y});
    }
    bool cycle = false;
    for(auto it : edges)
    {
        int x = it[0];
        int y = it[1];
        int a = find_set(x);
        int b = find_set(y);
        if(a == b)
        cycle = true;
        else
        union_set(a, b);
    }

    if(cycle)
    cout<<"cycle is present"<<endl;
    else
    cout<<"cycle is not present"<<endl;

    return 0;
}
```

Kruskal algorithm (Minimum spanning tree)

```
#include <bits/stdc++.h>
using namespace std;
const int N = 1e5 + 2;
vector<int> parent(N), sz(N);
```

```
int find_set(int v)
{
    if (parent[v] == v)
        return v;
    return parent[v] = find_set(parent[v]);
}

void union_set(int x, int y)
{
    int a = find_set(x);
    int b = find_set(y);
    if (a != b)
    {
        if (sz[a] < sz[b])
            swap(a, b);

        parent[b] = a;
        sz[a] += sz[b];
    }
}

int main()
{
    for (int i = 0; i < N; i++)
    {
        parent[i] = i;
        sz[i] = 1;
    }
    int n, m;
    cin >> n >> m;
    vector<vector<int>> edges;
```

```
    for (int i = 0; i < m; i++)
    {
        int x, y, w;
        cin >> x >> y>> w;
        edges.push_back({w, x, y});
    }

    sort(edges.begin(), edges.end());
    int cost = 0;
    for(auto it : edges)
    {
        int w = it[0];
        int x = it[1];
        int y = it[2];
        int a = find_set(x);
        int b = find_set(y);
        if(a == b)
        continue;
        else
        {
            union_set(x, y);
            cout<<x<<"-->"<<y<<endl;
            cost += w;
        }
    }
    cout << cost << endl;
    return 0;
}
```

Dijkstra algorithm (will not work for negative cycle)

```
#include<bits/stdc++.h>
using namespace std;
```

```
int main()
{
    int n, m;
    cin>>n>>m;
    vector<vector<pair<int, int>>> edges(n+1);      // n+1 size because node starts from 1;
    vector<int> dist(n+1, INT_MAX);
    for(int i=0; i<m; i++)
    {
        int x, y, w;
        cin>>x>>y>>w;
        edges[x].push_back({y, w});
        edges[y].push_back({x, w});
    }
    int source;
    cin>>source;
    set<pair<int, int>> s;
    dist[source] = 0;
    s.insert({0, source});
    while(!s.empty())
    {
        auto x = *(s.begin());
        s.erase(x);
        for(auto it : edges[x.second])
        {
            if(dist[it.first] > dist[x.second] + it.second)
            {
                s.erase({dist[it.first], it.first});
                dist[it.first] = it.second+dist[x.second];
                s.insert({dist[it.first], it.first});
            }
        }
    }
```

```
    for(int i=1; i<=n; i++)
    {
        if(dist[i] != INT_MAX)
        cout<<dist[i]<<" ";
        else
        cout<<"can't find";
    }
    cout<<endl;
    return 0;
}
```

Bellman's ford Algorithm (it can work for negative as well as positive edge weight)
Bellman-Ford algorithm does not work for graphs that contains a negative weight cycle.

```
#include<bits/stdc++.h>
using namespace std;
const int INF = 1e9;
int main()
{
    int n,m;
    cin>>n>>m;
    vector<vector<int>> edges;
    for(int i=0; i<m; i++)
    {
        int x, y, w;
        cin>>x>>y>>w;
        edges.push_back({x, y, w});
    }
    int source;
    cin>>source;
    vector<int> dist(n, INF);
    dist[source] = 0;
    for(int i =0; i<n-1; i++)
    {
```

```
        for(auto x : edges)
        {
            int u = x[0];
            int v = x[1];
            int w = x[2];
            dist[v] = min(dist[v], w + dist[u]);
        }
    }
    for(auto x : dist)
    {
        cout<<x<<" ";
    }
    cout<<endl;
    return 0;
}
```

Negative Cycle Detection

```
#include <bits/stdc++.h>
using namespace std;
int main()
{
    int n, m;
    cin >> n >> m;
    vector<vector<int>> edges;
    for (int i = 0; i < m; i++)
    {
        int x, y, w;
        cin >> x >> y >> w;
        edges.push_back({x, y, w});
    }
    // int source;
    // cin>>source;
```

```
    vector<int> dist(n + 1, 1e8); // nodes start with 1
    dist[1] = 0;
    for (int i = 1; i < n; i++)
    {
        for (auto x : edges)
        {
            int u = x[0];
            int v = x[1];
            int w = x[2];
            dist[v] = min(dist[v], w + dist[u]);
        }
    }
    bool flag = false;
    for (auto x : edges)
    {
        int u = x[0];
        int v = x[1];
        int w = x[2];
        if (dist[v] > min(dist[v], w + dist[u]))
        {
            flag = true;
            break;
        }
    }
    if (flag)
    {
        cout << "Negative cycle is present" << endl;
    }
    else
    {
        cout << "No negative cycle" << endl;
    }
```

```
    return 0;
}
```

Floyd's warshall algorithm

```
#include <bits/stdc++.h>
using namespace std;
int const N = 1e9;
int main()
{
    vector<vector<int>> graph = {{0, 5, N, 10}, {N, 0, 3, N}, {N, N, 0, 1}, {N, N, N, 0}};
    int n = graph.size();
    vector<vector<int>> dist = graph;
    for (int i = 0; i < n; i++)
    {
        for (int j = 0; j < n; j++)
        {
            for (int k = 0; k < n; k++)
            {
                dist[i][j] = min(dist[i][j], dist[i][k] + dist[k][j]);
            }
        }
    }
    for (int i = 0; i < n; i++)
    {
        for (int j = 0; j < n; j++)
        {
            if (dist[i][j] == N)
            {
                cout << "N ";
            }
            else
                cout << dist[i][j] << " ";
```

```
        }
        cout << endl;
    }
    return 0;
}
```

Prim's Algorithm

```
#include<bits/stdc++.h>
using namespace std;
void prims(vector<vector<pair<int, int>>> edges, int n){
    priority_queue<pair<int, int>, vector<pair<int, int>>, greater<pair<int, int>>> pq;
    vector<bool> visited(n, false);
    pq.push({0, 0});
    while(!pq.empty()){
        pair<int, int> p = pq.top();
        pq.pop();
        if(!visited[p.second]){
            visited[p.second] = true;
            cout<<p.second<<"->"<<p.first<<endl;
            for(auto it : edges[p.second]){
                pq.push({it.second, it.first});
            }
        }
    }
}
int main()
{
    int n, m;
    cin>>n>>m;
    vector<vector<pair<int, int>>> edges(n);
    for(int i=0; i<m; i++){
        int x, y, w;
```

```
        cin>>x>>y>>w;
        edges[x].push_back({y, w});
        edges[y].push_back({x, w});
    }
    prims(edges, n);
    return 0;
}
```

ADVANCE GRAPH THEORY
Kosaraju Algorithm (Strongly Connected Components)

```
#include <bits/stdc++.h>
using namespace std;

void Strong_dfs(int idx, vector<bool> &visited, vector<vector<int>> &newedges)
{
    visited[idx] = true;
    for (auto it : newedges[idx])
    {
        if (!visited[it])
        {
            Strong_dfs(it, visited, newedges);
        }
    }
}

void dfs(int idx, vector<bool> &visited, vector<vector<int>> &edges, stack<int> &st)
{
    visited[idx] = true;
    for (auto it : edges[idx])
    {
        if (!visited[it])
        {
            dfs(it, visited, edges, st);
```

```
        }
    }
    st.push(idx);
}

int main()
{
    int n, m;
    cin >> n >> m;
    vector<vector<int>> edges(n);
    for (int i = 0; i < m; i++)
    {
        int x, y;
        cin >> x >> y;
        edges[x].push_back(y);
    }
    vector<bool> visited(n, false);
    stack<int> st;
    for (int i = 0; i < n; i++)
    {
        if (!visited[i])
        {
            dfs(i, visited, edges, st);
        }
    }

    vector<vector<int>> newedges(n);
    for (int i = 0; i < n; i++)
    {
        for (auto it : edges[i])
        {
            newedges[it].push_back(i);
```

```
        }
    }
    fill(visited.begin(), visited.end(), false);


    int count = 0;
    while (!st.empty())
    {
        int t = st.top();
        st.pop();
        if (!visited[t])
        {
            Strong_dfs(t, visited, newedges);
            count++;
        }
    }
    cout << count << endl;
    return 0;
}
```

0-1 BFS

```
#include <bits/stdc++.h>
using namespace std;
int main()
{
    int n, m;
    cin >> n >> m;
    vector<vector<pair<int, int>>> edges(n + 1);
    for (int i = 0; i < m; i++)
    {
        int x, y;
        cin >> x >> y;
        if (x == y)
```

```
        continue;
    edges[x].push_back({y, 0});
    edges[y].push_back({x, 1});
}
vector<int> level(n + 1, INT_MAX);
deque<int> pq;
pq.push_back(1);
level[1] = 0;
while (!pq.empty())
{
    int t = pq.front();
    pq.pop_front();
    for (auto it : edges[t])
    {
        int node = it.first;
        int wt = it.second;
        if (level[t] + wt < level[node])
        {
            level[node] = level[t] + wt;
            if (wt == 1)
                pq.push_back(node);
            else
                pq.push_front(node);
        }
    }
}
if (level[n] == INT_MAX)
{
    cout << -1 << endl;
}
else
{
```

```
        cout << level[n] << endl;
    }
    return 0;
}
```

Articulation Points
Time complexity = O(Node+Edges)
Space complexity = O(n);

```
#include <bits/stdc++.h>
using namespace std;
vector<int> parent, disc, low;
vector<bool> visited, check;

void dfs(int idx, int parent, vector<vector<int>> &edges, int &time)
{
    disc[idx] = low[idx] = time;
    time++;
    int count = 0;
    visited[idx] = true;

    for (auto it : edges[idx])
    {
        if (it == parent)
            continue;
        if (visited[it] == true)
        {
            low[idx] = min(low[idx], disc[it]);
        }
        else
        {
            dfs(it, idx, edges, time);
            low[idx] = min(low[idx], low[it]);
            if (parent != -1 && low[it] >= disc[idx])
```

```
            {
                check[idx] = true;
            }
            count++;
        }
    }
    if (parent == -1)
    {
        if (count >= 2)
        {
            check[idx] = true;
        }
    }
}
int main()
{
    int n, m;
    cin >> n >> m;
    vector<vector<int>> edges(n + 1);
    for (int i = 0; i < m; i++)
    {
        int x, y;
        cin >> x >> y;
        edges[x].push_back(y);
        edges[y].push_back(x);
    }
    // n+1 because nodes start from 1.
    disc.resize(n + 1);
    low.resize(n + 1);

    visited.resize(n + 1, false);
    check.resize(n + 1, false);
```

```
    int time = 0;
    for (int i = 1; i < n + 1; i++)
    {
        if (!visited[i])
        {
            dfs(i, -1, edges, time);
        }
    }
    int ans = 0;
    for (int i = 1; i < n + 1; i++)
    {
        if (check[i] == true)
            ans++;
    }
    cout << ans << endl;
    return 0;
}
```

Tarjan Algorithm (Critical Connections in a Network)
A critical connection is a connection that, if we removed the edge, will make unable to reach some other nodes.

```
#include<bits/stdc++.h>
using namespace std;

vector<int> parent, disc, low;
vector<bool> visited;
vector<vector<int>> ans;

void dfs(int idx, int parent, vector<vector<int>> &edges, int &time)
{
    disc[idx] = low[idx] = time;
    time++;
    visited[idx] = true;
```

```
    for (auto it : edges[idx])
    {
        if (it == parent)
            continue;
        if (visited[it] == true)
        {
            low[idx] = min(low[idx], disc[it]);
        }
        else
        {
            dfs(it, idx, edges, time);
            low[idx] = min(low[idx], low[it]);
            if (low[it] > disc[idx])
            {
                ans.push_back({idx, it});
            }
        }
    }
}

int main()
{
    int n, m;
    cin>>n>>m;
    vector<vector<int>> edges(n);
    for(int i = 0; i<m; i++){
        int u,v;
        cin>>u>>v;
        edges[u].push_back(v);
        edges[v].push_back(u);
    }
    disc.resize(n);
```

```
    low.resize(n);
    visited.resize(n, false);
    int time = 0;
    for (int i = 0; i < n; i++)
    {
        if (!visited[i])
        {
            dfs(i, -1, edges, time);
        }
    }
    cout<<"Critical Connections are :"<<endl;
    for(auto it : ans){
        cout<<"["<<it[0]<<","<<it[1]<<"]"<<endl;
    }
    return 0;
}
```

Feel Free to connect with me –

https://www.linkedin.com/in/himanshushekhar16/