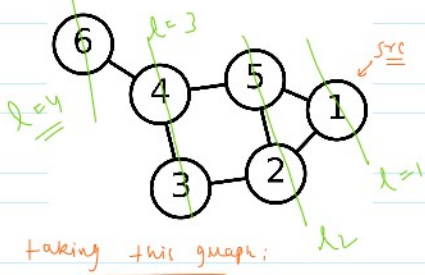
Graph algorithms

18 March 2021 10:24 AM

Bfs is where we traverse level by level

Bfs traversal :

```
class Graph {
    map<ll, list<ll>> l;
public:
    void bfs(int src) {
        int visited[10] = {0}; visited[src] = 1;
        queue<ll> q;
        int temp;
        q.push(src);
        while (!q.empty()) {
            temp = q.front();
            q.pop();
            for(auto n : l[temp]) {
                if(!visited[n])
                    q.push(n);  v[visited[n]++;
                }
            cout << temp << " ";
        }
    };
```



taking this graph;

if (src == 1)

1 → 2 → 5 → 3 → 4 → 6

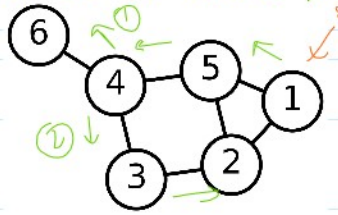
1 2 3 4

Dfs traversal:

for Dfs : → Recursive function.
→ Helper function.

Dfs basically completes chains or it chooses a particular node, then goes till end then backtracks for remaining edges.

Eg:



1 → 5 → 4 → 6 → 3 → 2.

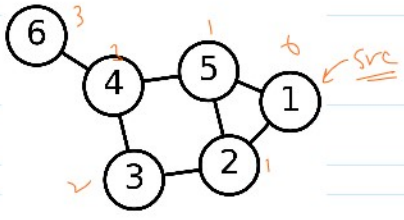
```
class graph {
    map<ll, list<ll>> l;
public:
    void dfs(int src) {
        map<ll,ll> visited;
        dfsT(src, visited);
    }

    void dfsT(int src, map<ll,ll> &v) {
        cout << src << " ";  v[src] = 1;
        for (auto nbr : l[src]) {
            if (!v[nbr]) {
                dfsT(nbr, v);
            }
        }
    }
};
```

Bfs single source shortest path algorithm :

→ This is based on Bfs traversal
→ We find the shortest distance of all the nodes from the source.
→ We use Bfs as it is the LEVEL ORDER TRAVERSAL.

```
void sssp(ll src) {
    queue<ll> q;
    q.push(src);
    ll temp;
    ll dist[10] = {0};

    for (auto x : l) {
        dist[x.first] = INT_MAX;
    }
    dist[src] = 0;

    while (!q.empty()) {
        temp = q.front();
        q.pop();
        for (auto nbr : l[temp]) {
            if (dist[nbr] == INT_MAX) {
                q.push(nbr);
                dist[nbr] = dist[temp] + 1;
            }
        }
    }

    for (auto x : l) {
        cout << "Distance of " << x.first << " from source is " << dist[x.first] << endl;
    }
```

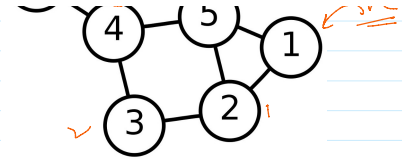
Eg:



OUTPUT

Dist of 1 from src is 0
2 1
3 2
4 2
5 1

```cpp
void sssp(ll src) {
    queue<ll> q;
    q.push(src);
    ll temp;
    ll dist[10] = {0};

    for (auto x : l) {
        dist[x.first] = INT_MAX;
    }
    dist[src] = 0;

    while (!q.empty()) {
        temp = q.front();
        q.pop();
        for (auto nbr : l[temp]) {
            if (dist[nbr] == INT_MAX) {
                q.push(nbr);
                dist[nbr] = dist[temp] + 1;
            }
        }
    }

    for (auto x : l) {
        cout << "Distance of " << x.first << " from source is " << dist[x.first] << endl;
    }
}
```

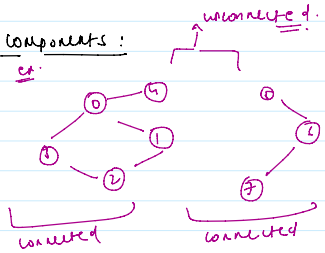


OUTPUT

Dist of 1 from src is 0
3 1
2 2
4 2
5 1
6 3

Connected components :

→ for eg.



→ Traverse the graph to find the components which are not connected we use dfs Traversal.

→ What's New ?

Here we apply dfs() call till all the vertices are not visited.

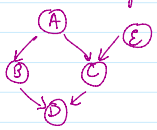
```cpp
void dfsT(ll src , map<ll, ll> &v) {
    cout << src << ' ';
    v[src] = 1;
    for (auto x : l[src]) {
        if (!v[x])
            dfsT(x, v);
    }
}

void dfs(ll src) {
    map<ll, ll> v;
    int cnt = 1;
    for (auto x : l) {
        if (!v[x.first]) {
            cout << "In connected component number : " << cnt << endl;
            dfsT(x.first , v);
            cout << endl;
            cnt++;
        }
    }
}
```

Here we are calling the dfs function multiple times.

Topological Sorting :

→ Sorting based on directed graph.

for eg.

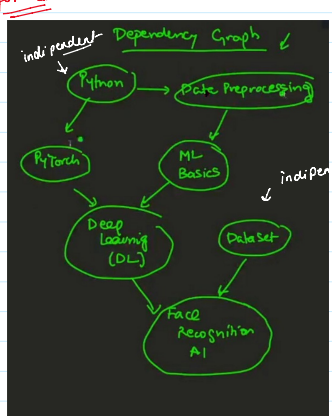


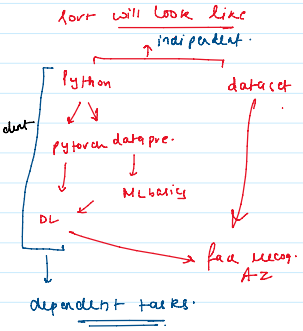
→ Here A & E are independent but B, C, D are dependent or have a parent.

→ In such cases where we sort based on parents or dependencies is called Topological Sorting.

```cpp
void dfsT(ll src , map<ll, ll> &v , list<ll> &vc) {
    //cout << src << ' ';
    v[src] = 1;
    for (auto x : l[src]) {
        if (!v[x])
            dfsT(x, v , vc);
    }
    vc.push_front(src);
}

void dfs() {
    map<ll, ll> v;
    list<ll> li;
    for (auto x : l)
        if (!v[x.first])
            dfsT(x.first, v , li);

    for (auto x : li)
        cout << x << ' ';

}
```

for eg.

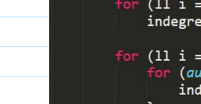




Topological sorting using Bfs :

1) → we make an indegree array.

What is indegree ? → Eg.



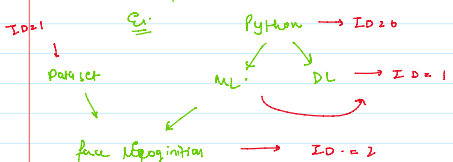
no. of edges coming inside of the vertex.

2) → we calculate indegree of every vertex.

→ we traverse the graph using Bfs traversal and reduce indegree of every vertex found. If indegree of any vertex is '0', then we push it into the queue.

Reasoning ?

Eg.



When we pop Python.

then the ID of ML & DL = 0 & only then we can output them.

```cpp
void topological_sort() {
    ll *indegree = new ll[v];
    for (ll i = 0; i < v; i++)
        indegree[i] = 0;

    for (ll i = 0; i < v; i++) {
        for (auto x : l[i]) {
            indegree[x]++;
        }
    }

    queue<ll> q;

    for (ll i = 0; i < v; i++) {
        if (!indegree[i])
            q.push(i);
    }

    while (!q.empty()) {
        ll temp = q.front();
        q.pop();
        cout << temp << "--> ";
        for (auto x : l[temp]) {
            indegree[x]--;
            if (!indegree[x])
                q.push(x);
        }
    }
}
```

→ Calculating indegree.

→ Pushing vertex's with indegree = 0 in queue

When we traverse the array using Bfs when ID of any vertex is >0, we decrement it & when ID=0, we push into the queue.

Checking if a graph is a tree using BFS!

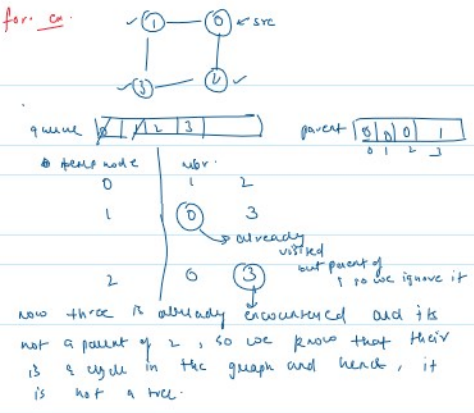
→ we do a bfs traversal in a graph.

→ the difference is that we maintain a parent array. In this parent array we store the parent of every visited node.

→ when that node is encountered agin, if its a parent of the current node's, then we ignore but if its not, it means the given graph has a cycle and thus its not a tree.

```
bool is_tree(ll src) {
    ll parent[v];
    for (ll i = 0; i < v; i++) {
        parent[i] = i;
    }
    ll visited[v] = {0};
    queue<ll> q;
    q.push(src);
    visited[src] = 1;
    ll temp;
    while (!q.empty()) {
        temp = q.front();
        q.pop();
        for (auto x : l[temp]) {
            if (!visited[x]) {
                q.push(x);
                visited[x] = 1;
                parent[x] = temp;
            }
            else if (visited[x])
                if (x != parent[temp])
                    return false;
        }
    }
    return true;
}
```

for. ex:



Checking cycle in a directed graph using DFS:

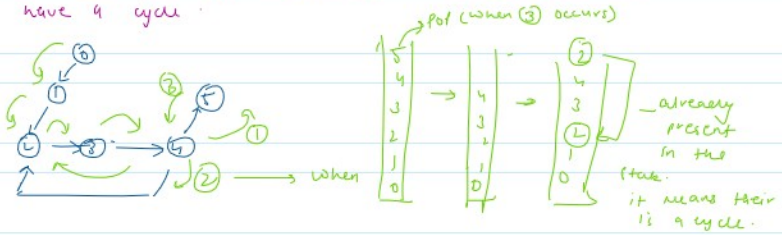
→ here we maintain a path using a stack (not actual stack, array with the name stack) and push every new neighbour in the stack.

→ we store the path we travel in a stack and check if the next found neighbour is present in the path stack.

→ If we find the neighbour, its obvious that we have a cycle.

```
bool cycle_helper(ll node, bool *visited, bool *stack) {
    visited[node] = true;
    stack[node] = true;

    for (auto nbr : l[node]) {
        if (stack[nbr] == true)
            return true;
        else if (!visited[nbr]) {
            bool finder = cycle_helper(nbr, visited, stack);
            if (finder == true)
                return true;
        }
    }

    stack[node] = false;
    return false;
}

bool check_cycle(ll src) {
    bool visited[v] = {0};
    bool stack[v] = {0};
    return cycle_helper(src, visited, stack);
}
```

Ex:

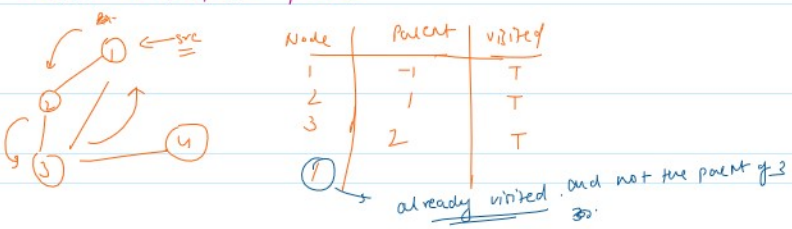


Check for cycle in an undirected graph using dfs traversal

→ unlike bfs, here we dont need to maintain a parent array because every node is the nbr of other. hence we can just take a single variable to represent parent.

```
bool cycle_helper(ll node, ll *visited, ll parent) {
    visited[node] = 1;
    for (auto nbr : l[node]) {
        if (visited[nbr] and parent != nbr)
            return true;
        else if (!visited[nbr]) {
            bool finder = cycle_helper_dfs(nbr, visited, node);
            if (finder)
                return true;
        }
    }
    return false;
}

bool check_cycle(ll src) {
    ll visited[v] = {0};
    return cycle_helper_dfs(src, visited, -1);
}
```

Ex:



Hence we conclude that we have a cycle.

Single source shortest path algorithm — weighted graph (Dijkstra's algorithm)

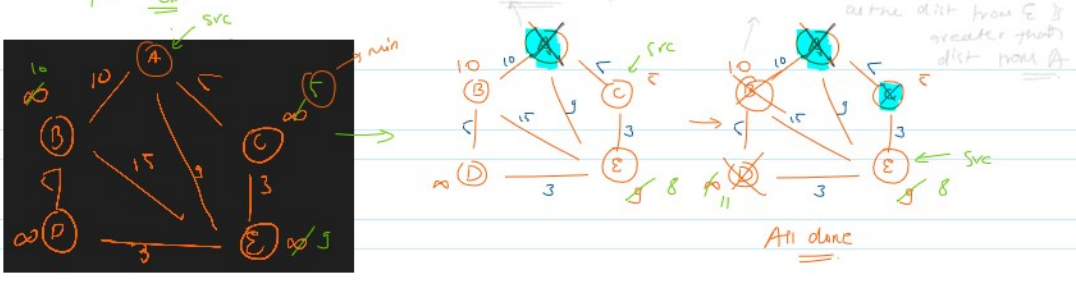
→ dijkstra's algorithm is a single source shortest path algorithm

→ Initially we mark distance of every node = ∞, src − dist = 0

→ then we traverse the graph and find shortest dist.

```
void dijkstra(char src) {
    unordered_map <char , ll> dist; /* location , distance */
    set<pair<ll , char>> s;
    for (auto x : m) {
        dist[x.first] = INT_MAX;
    }
    dist[src] = 0;
    s.insert(make_pair(0, src));

    while (!s.empty()) {
        auto i = *(s.begin());
        ll d = i.first;
        char v = i.second;

        for (auto nbr : m[v]) {
            if (nbr.second + d < dist[nbr.first]) {
                char c = nbr.first;
                auto f = s.find(make_pair(dist[c], c));
                if (f != s.end()) {
                    s.erase(f);
                }
                dist[nbr.first] = nbr.second + d;
                s.insert(make_pair(dist[c], c));
            }
        }

        s.erase(s.begin());
    }
```

for ex.



Disjoint set union (DSU)

→ DSU is a simple set which performs 2 main operations.

find()
- → It is a recursive function.
- → finds the super parent of the set & returns it.
- → It is called by the union function.

union()
- → It calls on 2 vertices and find their super parent.
- → It merges the two nodes.

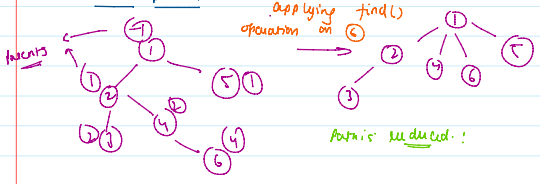
```
int findSet(ll x, ll *parent) {
    if (parent[x] == -1)
        return x;

    return findSet(parent[x], parent);
}

void unionSet(ll x, ll y, ll *parent) {
    ll s1 = findSet(x, parent);
    ll s2 = findSet(y, parent);

    if (s1 != s2)
        parent[s2] = s1;
}
```

→ O(n)

→ O(n²)

We can optimize this code further using 2 optimizations.
- → (1) path compression
- → (2) union by rank.

Path compression:

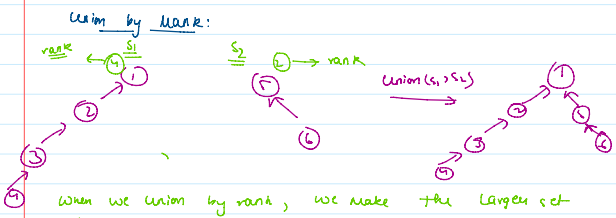


Making super parent the direct parent of child.

```
int findSet(ll x, ll *parent) {
    if (parent[x] == -1)
        return x;

    return parent[x] = findSet(parent[x], parent); // --> PATH COMPRESSION
}

void unionSet(ll x, ll y, ll *parent, ll *rank) {
    ll s1 = findSet(x, parent);
    ll s2 = findSet(y, parent);

    // UNION BY RANK
    if (s1 != s2) {
        if (rank[s1] > rank[s2]) {
            rank[s1] += rank[s2];
            parent[s2] = s1;
        } else {
            rank[s2] += rank[s1];
            parent[s1] = s2;
        }
    }
}
```

→ the set with higher rank becomes the parent.

Union by rank:

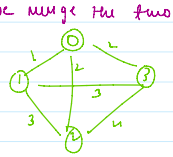


When we union by rank, we make the larger set the parent of smaller set.

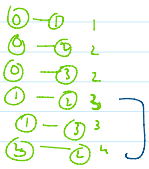
Graph Kruskal's algorithm:

→ We take a graph as input.
→ We sort all the edges according to their weight.
→ Then we merge the two vertices together using a DSU.

for eg.



step-1 sort acc. to weight



set
{0,1}
{0,1,2}
{0,1,2,3}

already in the set so the weights are not added to the answer.

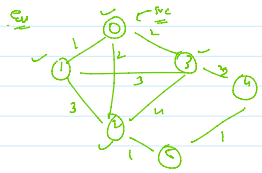
ans = 0
+1 = 1
+2 = 3
+2 = 5
5
5
5

weight of MST → 5

```
ll kruskals() {
    DSU s(V);
    ll ans = 0;
    sort(v.begin(), v.end());
    for (auto vc : v) {
        ll w = vc[0];
        ll x = vc[1];
        ll y = vc[2];


        ll s1 = s.find(x);
        ll s2 = s.find(y);

        if (s1 != s2) {
            s.unite(s1, s2);
            ans += w;
        }

    }

    return ans;
}
```

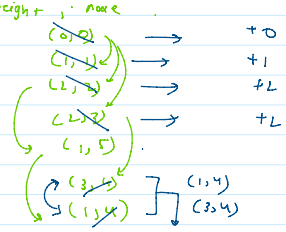
Graph Prim's algorithm:

→ We take a graph and insert all its edges in a priority queue. (Implemented as Min-heap).
→ Then we traverse the graph in a BFS-fashion and push the neighbours of the top of P.Q. into the queue.



P.Q. (min-heap)
weight, node

ans = 0



Since 4 is reachable by both but since it's a min heap, the weight '1' will come on top

```
int prim_mst() {
    priority_queue <pair<ll, ll>, vector<pair<ll, ll>>, greater<pair<ll, ll>>> q;

    ll ans = 0;
    ll *visited = new ll[V] {0};
    q.push({0, 0}); // weight , vertex

    while (!q.empty()) {
        auto best = q.top();
        q.pop();

        auto to = best.first;
        auto w = best.second;

        if (visited[to])
            continue;

        ans += w;
        visited[to] = 1;

        for (auto nbr : l[to]) {
            if (!visited[nbr.first])
                q.push({nbr.first , nbr.second});
        }
    }

    return ans;
}
```

Bellman ford algorithm :

→ This is a single source shortest path algorithm like dijkstra but unlike dijkstra we can have a negative weight edge here.

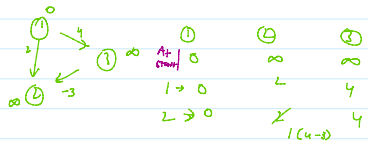
→ We take an edge list implement either of graph

→ We traverse the graph (V-1) times & update the edges accordingly.

→ Lastly we check that is their a negative weight cycle.

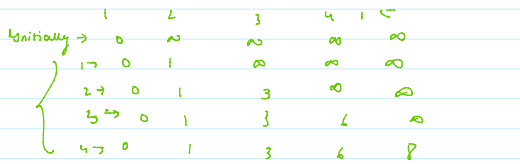
```
vector<ll> bellman_ford(ll src, vector<vector<ll>> &edges, ll n) {

    vector<ll> dist(n + 1, INT_MAX); → initially all the nodes at ∞ except src.
    dist[src] = 0;
    for (ll i = 0; i < n; i++) {

        for (auto edge : edges) {
            ll u = edge[0];
            ll v = edge[1];
            ll w = edge[2];

            if (dist[u] != INT_MAX and dist[u] + w < dist[v]) {
                dist[v] = dist[u] + w;
            }
        }
    }

    for (auto edge : edges) {
        ll u = edge[0];
        ll v = edge[1];
        ll w = edge[2];

        if (dist[u] != INT_MAX and dist[u] + w < dist[v]) {
            cout << "Negative edge cycle!";
            exit(0);
        }
    }

    return dist;
}
```

for ex.



Why do we check (V-1) times ?

for ex.



In the worst case, i.e. the graph is not connected, it took us (V-1) i.e. (5-1) steps to complete the traversal.

Floyd warshall's algorithm :

→ Unlike bfs, dijkstra or bellman-ford, floyd warshall is a all source shortest path algorithm. (APSP)

→ It is a $O(n^3)$ algo.

```
loop (0, V, k) {
    loop (0, V, i) {
        loop (0, V, j) {
        }
    }
}
```

→ Using 'k' or most outer loop, we choose the intermediate node

→ Using 'i' & 'j' we choose the src & dest.

```
void floydWarshall(vector<vector<ll>> &v , ll n) {
    vector<vector<ll>> dist(v);
    for (ll k = 0; k < n; k++) {
        for (ll i = 0; i < n; i++) {
            for (ll j = 0; j < n; j++) {
                if (dist[i][k] + dist[k][j] < dist[i][j])
                    dist[i][j] = dist[i][k] + dist[k][j];
            }
        }
    }
    for (ll i = 0; i < n; i++) {
        for (ll j = 0; j < n; j++) {
            cout << dist[i][j] << ' ';
        }
        cout << endl;
    }
}
```

Kosaraju's algorithm :

→ To isolate or find multiple strongly connected components in a graph we use this algorithm.

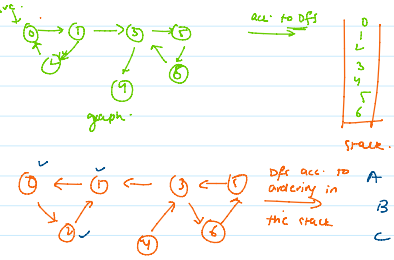
What is a strongly connected component ?

in this ex. {1, 2, 3} form a strongly connected comp. i.e. we can reach any node from any node.

→ In this algorithm → we traverse the graph using Dfs and insert the nodes in the stack when we return from the recursive calls.

→ Then we traverse the reverse graph (all edge are reversed) using Dfs and acc. to the ordering in the stack.

for ex.



```
void dfs(vector<ll> graph[], ll i, ll *visited, vector<ll> &stack) {
    visited[i] = 1;

    for (auto nbr : graph[i]) {
        if (!visited[nbr])
            dfs(graph, nbr, visited, stack);
    }

    stack.push_back(i);
}

void dfs2(vector<ll> rev_graph[], ll i, ll *visited) {
    visited[i] = 1;
    cout << i << ' ';
    for (auto nbr : rev_graph[i]) {
        if (!visited[nbr])
            dfs2(rev_graph, nbr, visited);
    }
}
```

```
void solve(vector<ll> graph[], vector<ll> rev_graph[], ll n) {
    ll visited[n];
    memset(visited, 0, sizeof(visited));
    vector<ll> stack;

    for (ll i = 0; i < n; i++) {
        if (!visited[i])
            dfs(graph, i, visited, stack);
    }

    memset(visited, 0, sizeof(visited));

    ll num = 1;
    for (ll i = stack.size() - 1; i >= 0; i--) {
        if (!visited[stack[i]]) {
            cout << "Connected component " << num++ << " -->  ";
            dfs2(rev_graph, stack[i], visited);
            cout << endl;
        }
    }

}
```


A → 0, 1, 2

B → 3, 6, 5

C → 4