Graph Theory

Part - 01

By Adnan Zawad Toky

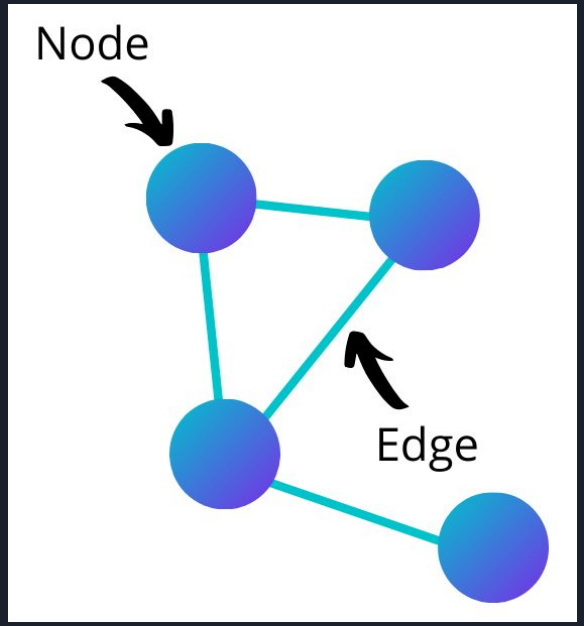
Contents

What is graph?

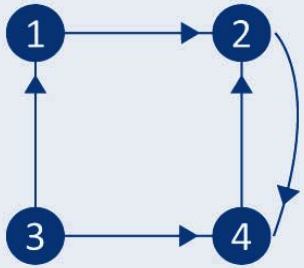
Graph is a data structure consisting of two types of components:

1. Node or vertex
2. Edge (connection between pair of nodes)

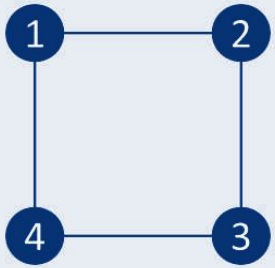
A graph is a pair of sets (V, E), where V is the set of vertices and E is the set of edges, connecting the pairs of vertices.



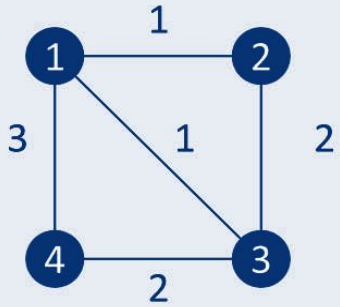
Different Types of Graph



Directed Graph



Undirected Graph



Weighted Graph

Graph Visualization

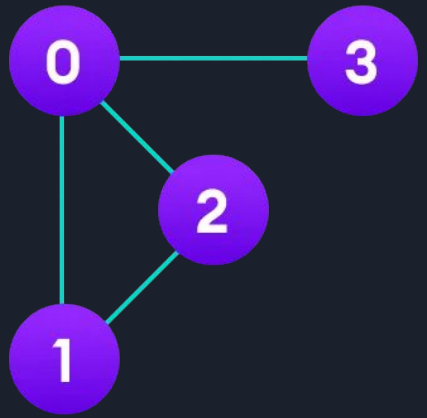
There are lots of tools to visualize graphs. For example:

- https://csacademy.com/app/graph_editor/
- http://mxwell.github.io/draw-graph
- http://pastegraph.herokuapp.com/

Basic Terminologies

Adjacent nodes:

A vertex is said to be adjacent to another vertex if there is an edge connecting them.

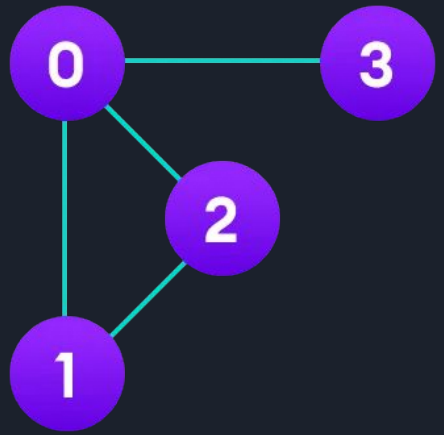


Here, node 0 and node 1 are adjacent to node 2 but node 3 is not.

Basic Terminologies

Degree of a node:

In an undirected graph, degree of a vertex V is the number of vertices adjacent to it.



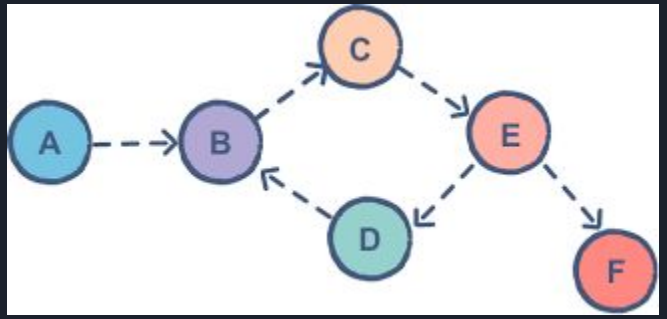
Node 0 has degree 3 and node 2 has degree 2.

Basic Terminologies

In a directed graph:

The **indegree** of a vertex V is the number of edges coming into vertex V.

The **outdegree** of a vertex V is the number of edges going out from vertex V.

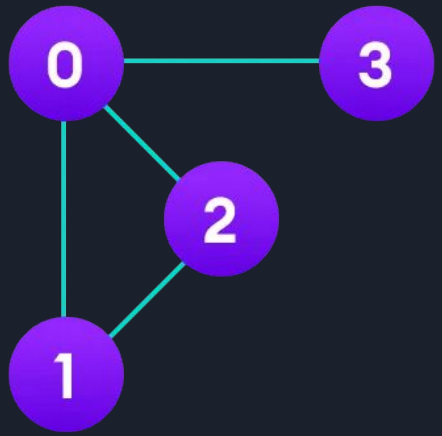


Here, the indegree of node B is 2 and outdegree of node B is 1.

Basic Terminologies

Path:

Path is a sequence of alternating vertices and edges such that the edge connects each successive vertex.



Some paths of the graph:

$0 \rightarrow 3$

$0 \rightarrow 2 \rightarrow 1$

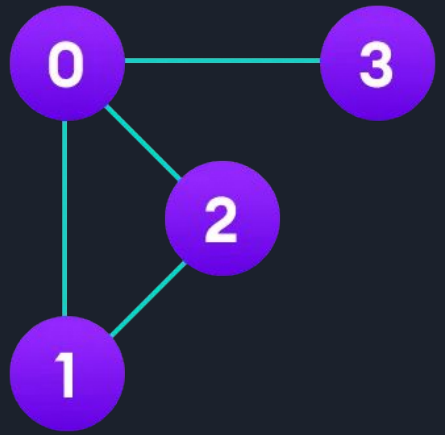
$2 \rightarrow 0 \rightarrow 3$

$2 \rightarrow 1 \rightarrow 0 \rightarrow 3$

Basic Terminologies

Cycle:

Cycle is a path that starts and ends at the same vertex.

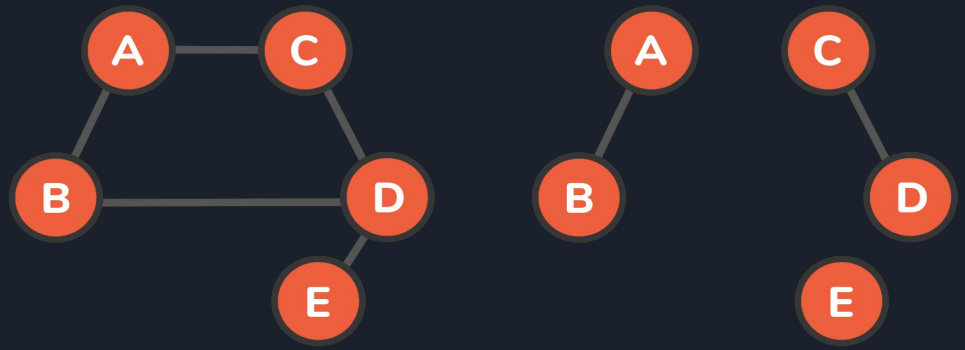


Here, $(2 \rightarrow 0 \rightarrow 1 \rightarrow 2)$ is a cycle

Basic Terminologies

Connected graph:

A graph is connected if there is a path between every pair of vertices.



Connected Disconnected

Graph Representation

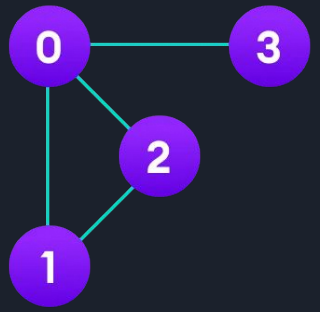
There are two common ways of graph representation

1. Adjacency Matrix Representation
2. Adjacency List Representation

Graph Representation

Adjacency Matrix Representation:

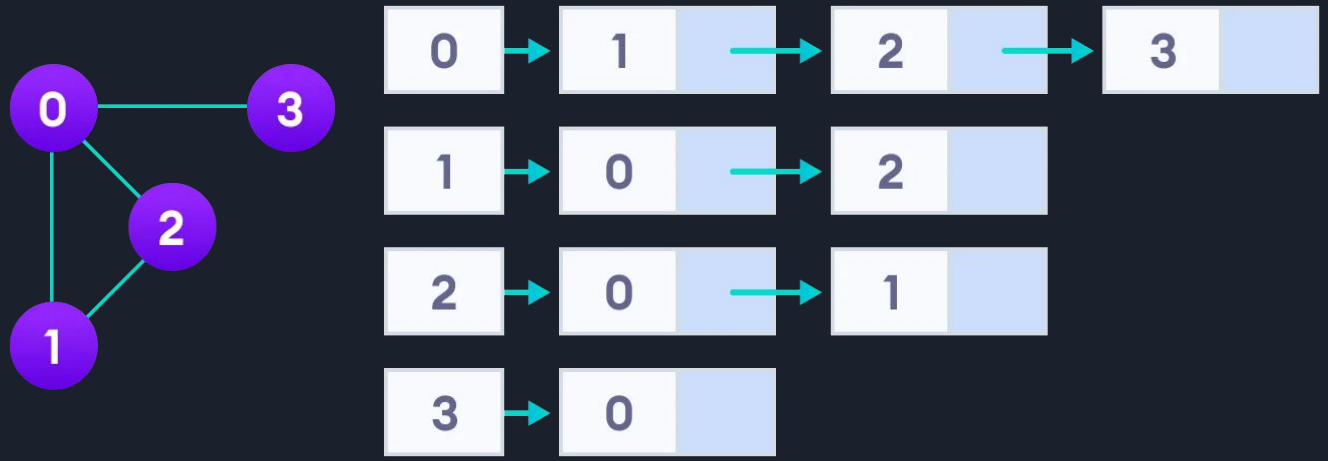
An adjacency matrix is a 2D array of size N x N where N is the number of vertices. If the array is a, then `a[i][j]` is 1 if there is an edge connecting vertex i and vertex j.



	0	1	2	3
0	0	1	1	1
1	1	0	1	0
2	1	1	0	0
3	1	0	0	0

Graph Representation

Adjacency List Representation:



Graph Traversal

Graph traversal is the process of visiting or checking each node of the graph.

Two commonly used algorithms for graph traversing:

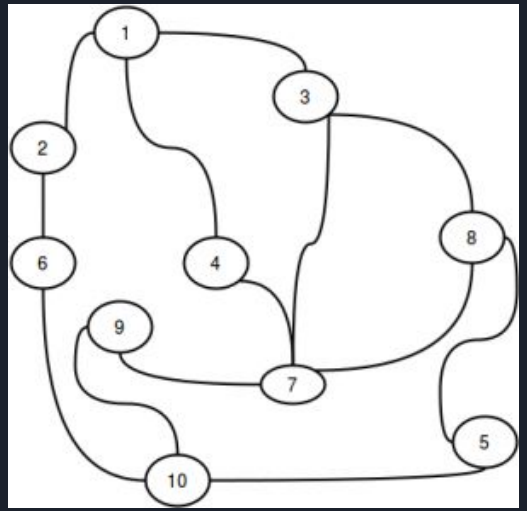
1. Breadth First Search (BFS)
2. Depth First Search (DFS)

Breadth First Search (BFS)

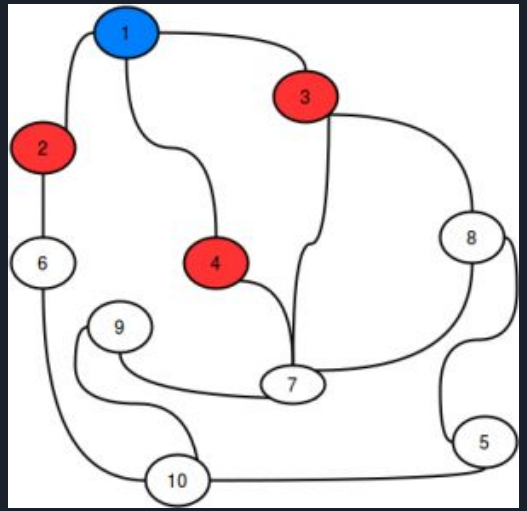
Breadth First Search (BFS)

- The algorithm starts traversing the graph from a specific node called the source node. The level of source node is 0.
- Each node is visited exactly once.
- Adjacent nodes of level 0 node is marked as level 1 node.
- Adjacent (unmarked) nodes of any level 1 node is marked as level 2 node.
- Traversal starts from level 0 (source) node. Then level 1 node is visited. Then level 2 and so on.

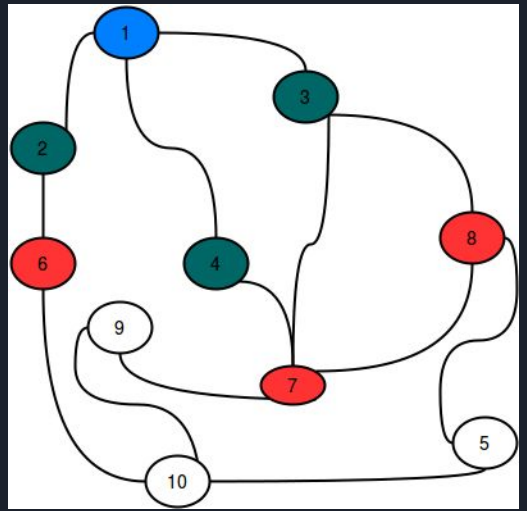
Breadth First Search (BFS)



Node 1 is source

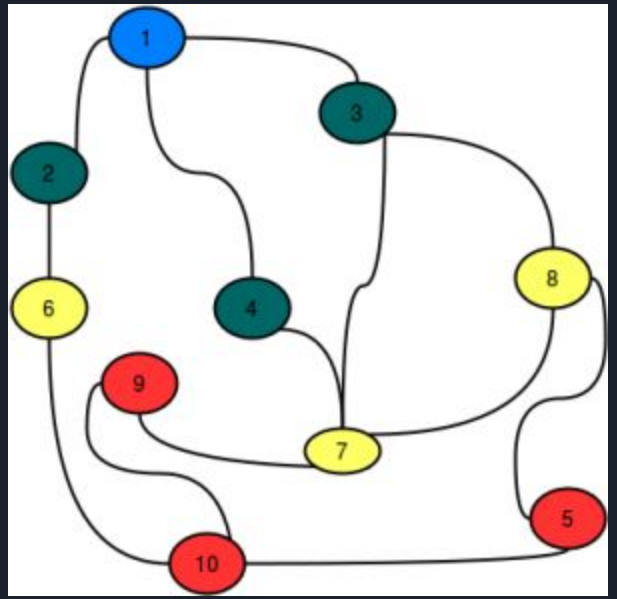


Node 2, 3 & 4 are level 1 nodes

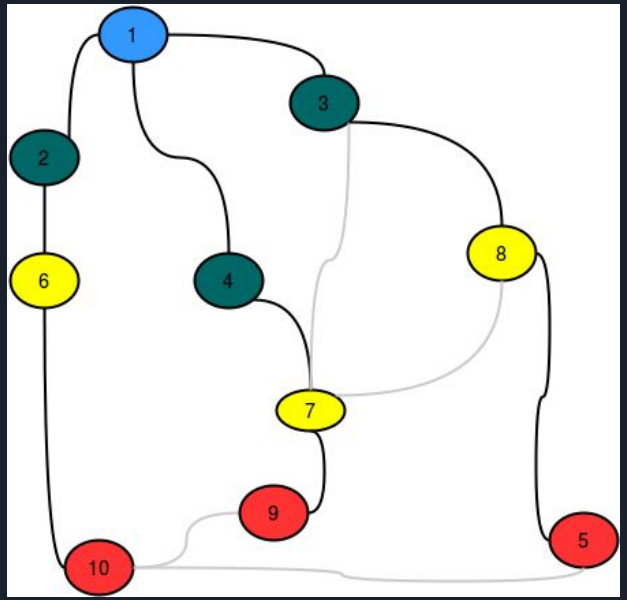


Node 6, 7 & 8 are level 2 nodes

Breadth First Search (BFS)



Node 5, 9 & 10 are level 3 nodes



BFS Tree

Breadth First Search (BFS)

```cpp
int n, m;
cin >> n >> m;
// n = number of nodes
// m = number of edges

vector<int> adj[n+1]; // adjacency list

for(int i = 0; i < m; i++){
    int u, v;
    cin >> u >> v;
    // there is an edge between node u and v
    adj[u].push_back(v); // v is an adjacent node of u
    adj[v].push_back(u); // u is an adjacent node of v
}
```

Breadth First Search (BFS)

```
vector<int> vis(n+1, 0);
vector<int> level(n+1, 0);
queue<int> q;
int src = 1; // starting bfs from node 1
vis[src] = 1; // source node is visited by default
level[src] = 0; // level of source node is 0
q.push(u);

while(!q.empty()){
    int u = q.front();
    q.pop();

    for(int v: adj[u]){ // iterating over the list of adjacent nodes
        if(vis[v]) continue; // node v is already visited
        level[v] = level[u]+1;
        q.push(v);
    }
}
```

Breadth First Search (BFS)

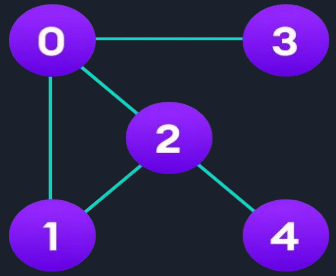
For an unweighted graph, the level of each node is the length of shortest path (minimum distance) from source node.

The time complexity of BFS algorithm is O(V+E) where V is the number of nodes and E is the number of edges. It is because, each node and edge of the graph is visited at most once.

Depth First Search (DFS)

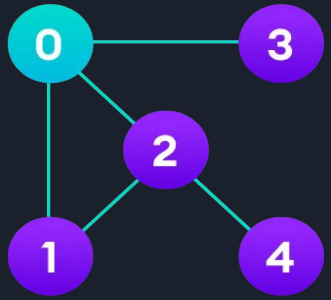
DFS is a recursive algorithm. The basic idea is to start from the root or any arbitrary node and mark the node as visited and move to the adjacent unmarked node and continue this loop until there is no unmarked adjacent node. Then backtrack and check for other unmarked nodes and traverse them.

Depth First Search (DFS)



Visited

Stack



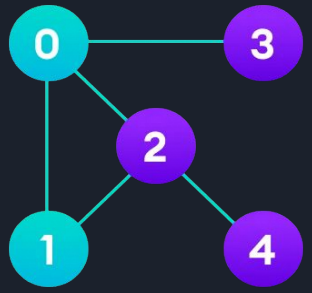
0				

Visited

1	2	3		

Stack

Depth First Search (DFS)

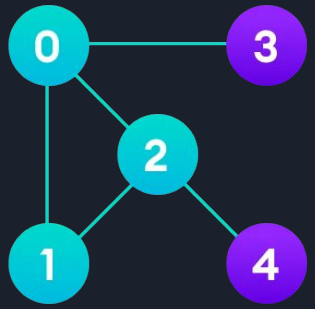


0	1			

Visited

2	3			

Stack



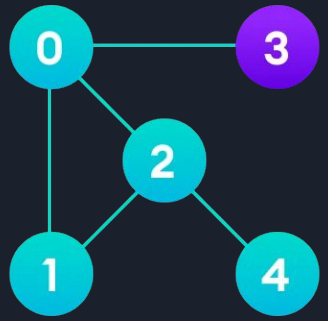
0	1	2		

Visited

4	3			

Stack

Depth First Search (DFS)

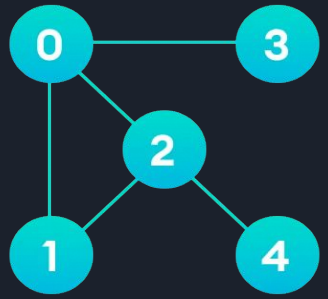


0	1	2	4	

Visited

3				

Stack



0	1	2	4	3

Visited

Stack

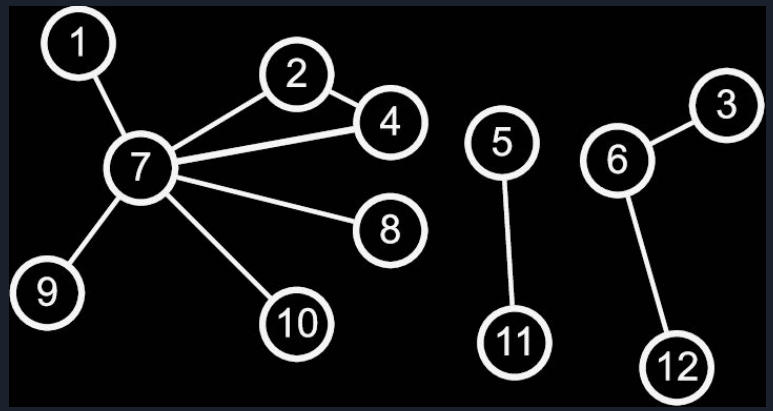
Depth First Search (DFS)

```
int vis[N];

void dfs(int u){
    if(vis[u]) return;

    vis[u] = 1;
    // node u is visited
    for(int v: adj[u]){
        dfs(v);
    }
}
```

Time Complexity: O(V+E)

Connected Components

A connected component or simply component of an undirected graph is a subgraph in which each pair of nodes is connected with each other via a path.



There are 3 connected components in the graph.
{1, 2, 4, 7, 8, 9, 10}
{5, 11}
{3, 6, 12}

Dijkstra’s Algorithm

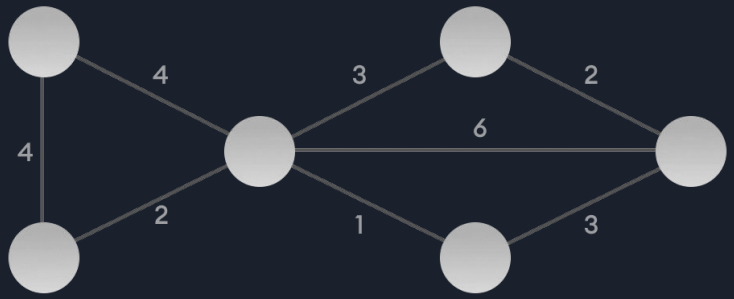
- Dijkstra's algorithm allows us to find the shortest path between any two vertices of a graph.
- The weights of the graph must be nonnegative for this algorithm to work.
- This algorithm finds the minimum distances to every node from a single source node.

Dijkstra’s Algorithm

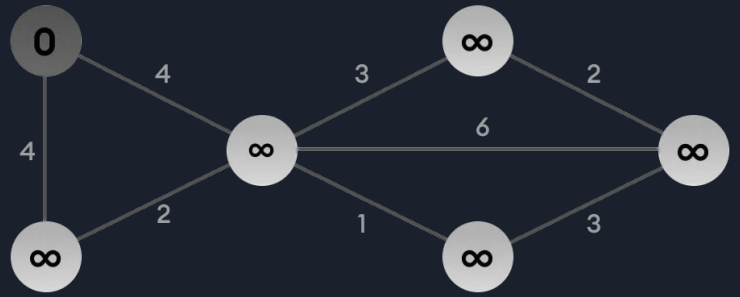
The algorithm works as follows:

- We maintain a list of nodes and an array dis where dis[i] is the minimum distance of node i from the source node.
- Initially distance of every node is infinity and distance of the source node is 0.
- Now, for each iteration, a node u is selected from the list which has the minimum distance from source and node u is removed from the list.
- For each node v from the adjacency list of node u, if the current distance of node v is greater than dis[u]+w(u, v), we update the distance of v and insert node v to the list. Here, w(u, v) denotes the weight of edge between the node u and node v.
- The algorithm stops when the list is empty.
- A priority queue is used to find the node with the minimum distance efficiently.

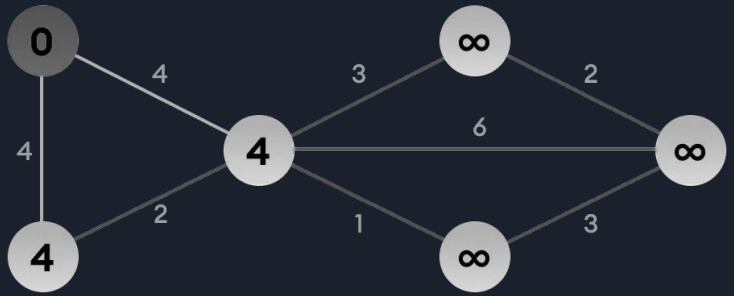
Dijkstra’s Algorithm



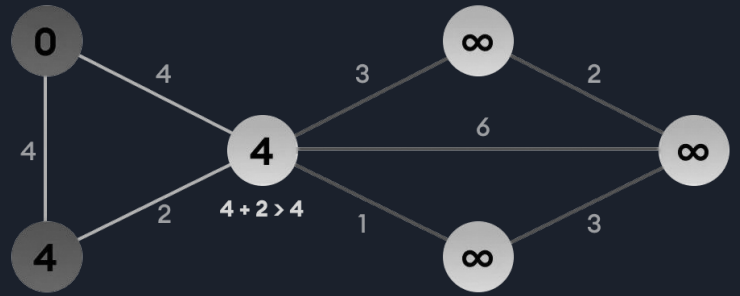
Step: 1



Step: 2

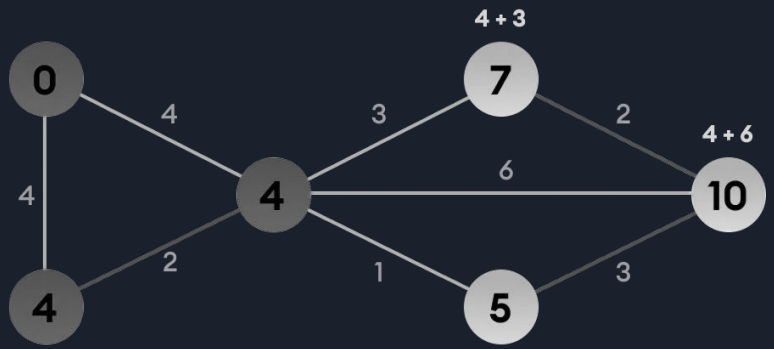


Step: 3

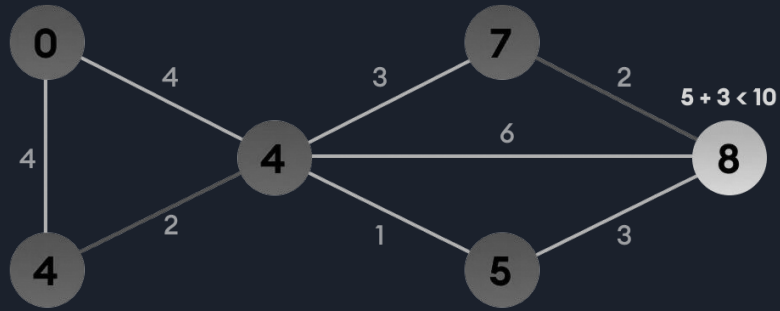


Step: 4

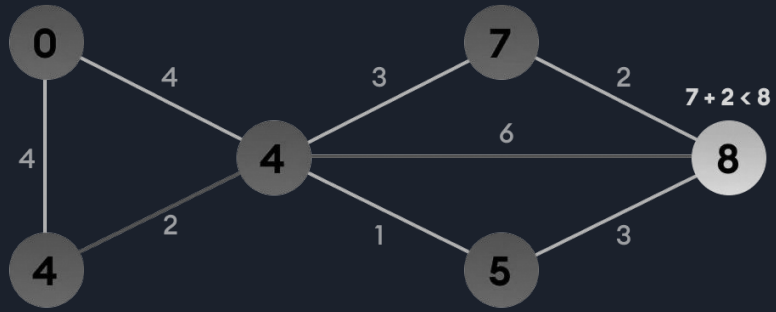
Dijkstra’s Algorithm



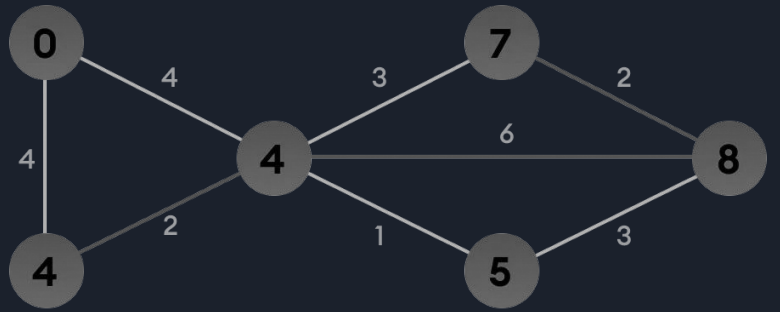
Step: 5



Step: 6



Step: 7



Step: 8

Dijkstra's Algorithm

```cpp
int n, m;
cin >> n >> m;
// n = number of nodes
// m = number of edges

vector<pair<int, int>> adj[n+1]; // adjacency list

for(int i = 0; i < m; i++){
    int u, v, w;
    cin >> u >> v >> w;
    // there is an edge between node u and v of weight w
    adj[u].push_back({v, w}); // v is an adjacent node of u
    adj[v].push_back({u, w}); // u is an adjacent node of v
}
```

Dijkstra’s Algorithm

```cpp
vector<int> dis(n+1, INF);
priority_queue<pair<int, int>> pq;
int src = 1; // node 1 is the source
dis[src] = 0; // distance of source node is 0
pq.push({0, src});

while(!pq.empty()){
   int u = pq.top().second;
   int d = -pq.top().first;
   pq.pop();

   if(dis[u]<d) continue; // an useful optimization

   for(auto v: adj[u]){ // iterating over the list of adjacent nodes
       if(dis[v.first]<=dis[u]+v.second) continue; // distance is already minimum
       dis[v.first] = dis[u]+v.second; // distance of adjacent node has updated
       pq.push({-dis[v.first], v.first}); // keeping the negative of distance
       // because default priority of priority queue is maximum to minimum
       // but we need minimum element first.
   }
}

// dis[i] is the minimum distance of node i from source
```

Bellman Ford & Floyd Warshall

- Bellman Ford
  - Finds single source shortest path with negative weight edges
  - Reference: https://cp-algorithms.com/graph/bellman_ford.html

- Floyd Warshall
  - Finds all pair shortest paths
  - Reference: https://cp-algorithms.com/graph/all-pair-shortest-path-floyd-warshall.html

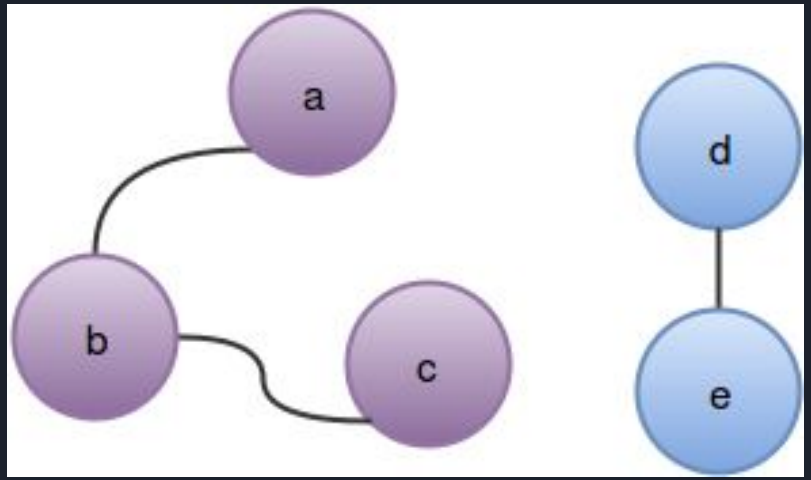
Challenge Problem

Given a N x N grid. Each cell contains a positive integer value which represents the amount of money you have to pay to visit that cell. You are at cell (1, 1) and you want to visit the cell (N, N). What is the minimum amount of money to travel from (1, 1) to (N, N) if you are only allowed to move left, right, up or down inside the grid?

Disjoint Set Union (DSU)

- Suppose A, B, C, D and E are 5 people.
- If A-B are friends and if B-C are friends then we can say A-C are also friends.
- Let’s say A-B, B-C and D-E are friends with each other.
- Query:
  - Are A and C friends?
  - Are B and D friends?

Disjoint Set Union (DSU)



Subset-1: {A, B, C}

Subset-2: {D, E}

A & C are friends as they belong to the same subset.

B & D are not friends as they belong to the different subset.

Disjoint Set Union (DSU)

How to process the queries efficiently?

- Finding connected components by running BFS/DFS
- Using DSU

Disjoint Set Union (DSU)

A disjoint-set is a data structure that operates with a set partitioned in several disjoint subsets. It typically supports two operations:

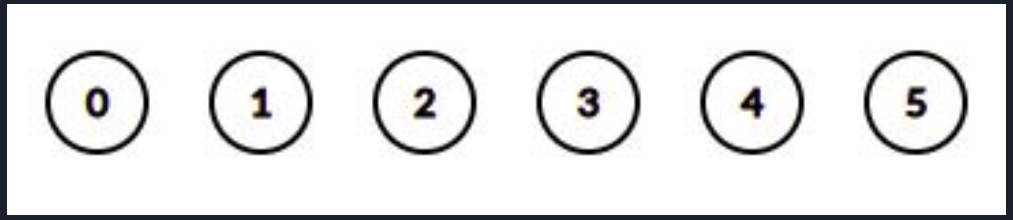
1. Find: Given a particular element of the set. It identifies the subset of the element. This can be used for determining whether two elements are in the same subset or not. (Check if two nodes are connected.)
2. Union: Joins two subsets into a single subset. (Add an edge between two nodes)

Disjoint Set Union (DSU)

- The disjoint-set works by representing each connected component as a rooted tree.
- Each component has a representative. The root of each tree is that component's representative.
- If the representative of two nodes are same, we can say that they are in the same component/subset. Otherwise they are in the different component/subset.

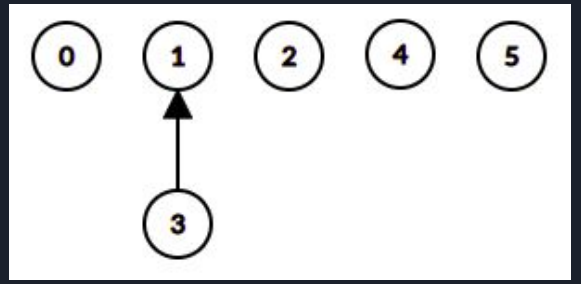
Disjoint Set Union (DSU)

Suppose there are 6 people. Initially, there are no friendship relations among them, so each node is the root of a tree (the people are numbered from 0 to 5):

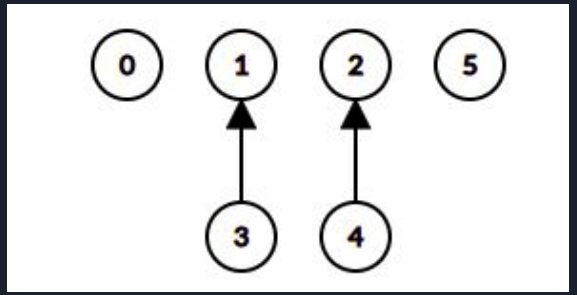


Disjoint Set Union (DSU)

1 and 3 become friends. Let's choose 1 as the representative.

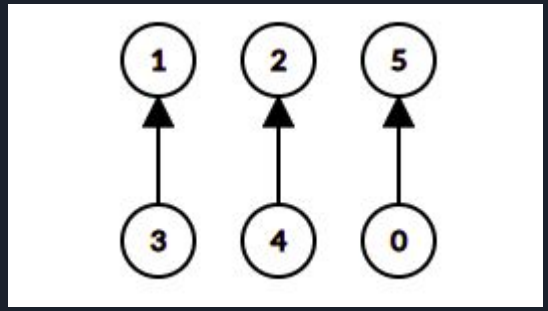


2 and 4 become friends. Let's choose 2 as the representative.

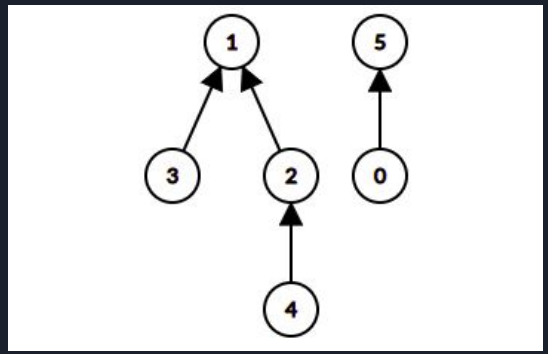


Disjoint Set Union (DSU)

0 and 5 become friends. Let's choose 5 as the representative.



2 and 1 become friends. Let's choose 1 as the representative.



Disjoint Set Union (DSU)

To combine two sets containing a and containing b, we first find the representative of the set in which a is located, and the representative of the set in which b is located. If the representatives are identical, then we have nothing to do, the sets are already merged. Otherwise, we can simply specify that one of the representatives is the parent of the other representative - thereby combining the two trees.

Disjoint Set Union (DSU)

```
int parent[N];
// parent[x] = x if x is the representative (root of the tree)

void make_set(int n) {
    for(int i = 1; i <= n; i++){
        parent[i] = i; // initially each node is representative of that node
    }
}

int find_set(int v) {
    if (v == parent[v]) // v is the root (representative)
        return v;
    return find_set(parent[v]); // representative of v is the representative of parent of v
}

void union_sets(int a, int b) {
    a = find_set(a);
    b = find_set(b);
    if (a != b) // a and b are in the different subsets
        parent[b] = a; // a is now the representative of b
}
```

Disjoint Set Union (DSU)

Complexity of find_set() function is O(N).

The overall complexity is O(N^2). We can optimize the algorithm in two ways:

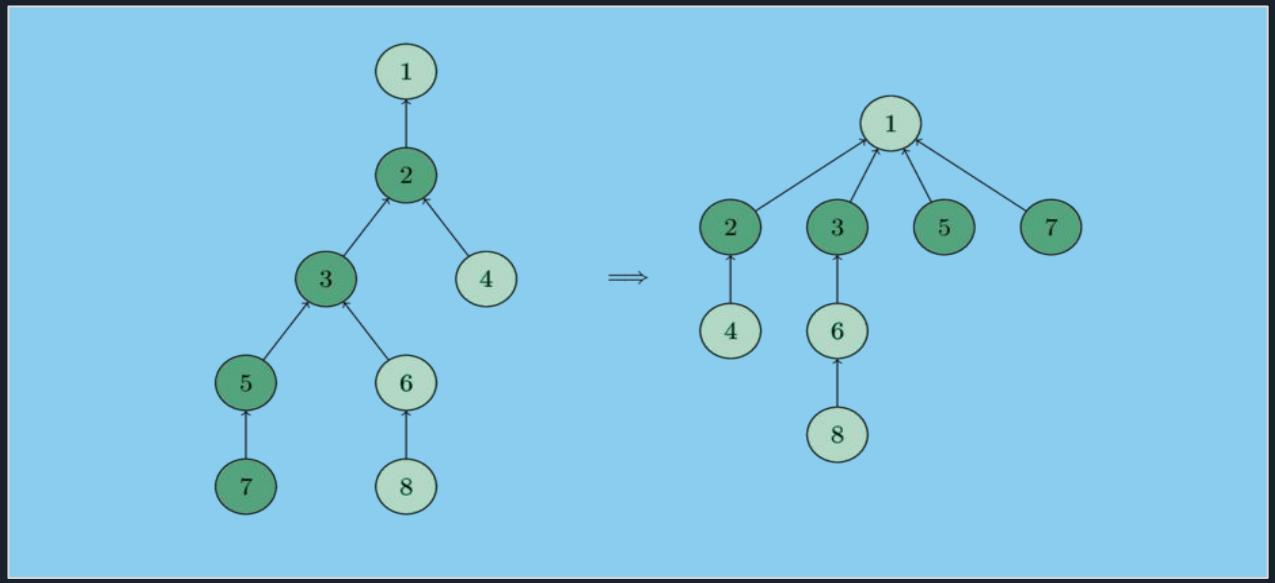
1. Path Compression Optimization
2. Union by Size/Rank

Disjoint Set Union (DSU)

Path Compression Optimization

1. This optimization is designed for speeding up find_set.
2. If we call find_set(v) for some vertex v, we actually find the representative p for all vertices that we visit on the path between v and the actual representative p.
3. The trick is to make the paths for all those nodes shorter, by setting the parent of each visited vertex directly to p.

Disjoint Set Union (DSU)



The tree on the right side is the compressed tree after calling find_set(7) which shortens the paths for the visited nodes 7, 5, 3 and 2.

Disjoint Set Union (DSU)

Implementation of path compression:

```
int find_set(int v) {
    if (v == parent[v])
        return v;
    return parent[v] = find_set(parent[v]);
    // first we find the representative of v and update the parent
    // of v with the representative
}
```

Average Complexity: O(logN) per query.

Disjoint Set Union (DSU)

Union By Size/Rank:

- In the naive implementation the second tree always got attached to the first one.
- In this optimization, we attach the tree with the lower size/rank to the one with the higher size/rank.
- This strategy ensures the maximum depth of the tree at most log(n)
- Here rank is the maximum depth of the tree.

Disjoint Set Union (DSU)

Union by Size:

```
void make_set(int v) {
    parent[v] = v;
    size[v] = 1;
}

void union_sets(int a, int b) {
    a = find_set(a);
    b = find_set(b);
    if (a != b) {
        if (size[a] < size[b])
            swap(a, b);
        parent[b] = a;
        size[a] += size[b];
    }
}
```

Union by rank:

```
void make_set(int v) {
    parent[v] = v;
    rank[v] = 0;
}

void union_sets(int a, int b) {
    a = find_set(a);
    b = find_set(b);
    if (a != b) {
        if (rank[a] < rank[b])
            swap(a, b);
        parent[b] = a;
        if (rank[a] == rank[b])
            rank[a]++;
    }
}
```

Thanks
Happy Coding