CONTENTS

Commonly Used Terms

- **Vertex/Node**
- **Edge**: A connection between two nodes
- **Undirected Graph**: Edges don't have a particular direction. An edge (u,v) is same as edge(v,u).
- **Directed Graph**: Edges do have a direction. An edge (u,v) doesn't guarantee that edge (v,u) exists. Even if there is edge(v,u) present in the graph, edge(u,v) and edge(v,u) are different.
- **Cyclic**: A graph is called cyclic if we can reach the same node from where we started by following some number of edges.
- **DAG or Directed Acyclic Graph**: A directed graph with no cycle.
- **Degree**: Degree of a node is number of edges attached to that node in a undirected graph. `Total degree of graph = 2 x (number of edges)`
- **Indegree**: The number of edges pointing towards a particular node in a directed graph is indegree of that node
- **Outdegree**: The number of edges pointing outwards a particular node in a directed graph is outdegree of that node
- **Edge weight**: A weight associated with the edge

Graph Representation

1. Adjacency Matrix

```
'''

Assume that nodes are integers and equivalent to range(1, len(nodes))

Declare a Adjacency matrix with values 0  of size (n+1) x (n+1) where n = number of v
ertices.

To mark an edge between two nodes u and v, mark matrix[u][v] and matrix[v][u] as 1

-> Matrix is symmetrical for undirected graph

-> For weighted edges, store the weight of edge instead of 1

-> Faster for dense graph

-> Simpler to implement weighted graph

-> Uses more space

'''

class Graph:

    def __init__(self, nodes, edges):

            self.nodes = nodes

                self.edges = edges

                self.adj_mat = [[0 for _ in range(len(self.nodes)+1)] for __ in rang
e(len(self.nodes)+1)]

                for u, v in self.edges:

                        self.adj_mat[u][v] = 1    # weight instead of 1 for weighted
graphs

                        self.adj_mat[v][u] = 1
```

2. Adjacency List

```
'''

Declare a list of empty lists of size (n+1) where n= number of vertices.

Index represents the vertex and the list stored at that index represents the adjacent
vertices of that vertex.

For every vertex, append the adjacent vertices in the list at that index.

'''

class Graph:

        def __init__(self, nodes, edges):

                self.nodes = nodes

                self.edges = edges

                self.adj = [[] for _ in range(len(self.nodes)+1)]

                for u, v in self.edges:

                        self.adj[u].append(v)

                        self.adj[v].append(u)
```

3. Dictionary

```
'''

Store every node as key and its adjacent nodes as values in list.

'''

class Graph:

    def __init__(self, nodes, edges):

            self.edges = edges

                self.nodes = nodes

                self.graph_dict = {node:[] for node in self.nodes}

                for u, v in self.edges:

                    self.graph_dict[u].append(v)

                        self.graph_dict[v].append(u)
```

Graph Traversal

1. Breadth First Search or BFS

```
'''
TC : O(N + E)
SC : O(N + N)
'''
def bfs(q, visited, res, graph_dict):
        while q:
            size = len(q)
                for i in range(size):
                        node = q.popleft()
                        res.append(node)
                        for adj in graph_dict[node]:
                                if adj not in visited:
                                        q.append(adj)
                                        visited.add(adj)

q = deque([start_node])
visited = set([start_node])
res = []
bfs(q, visited, res, graph_dict)
print(res)
```

2. Depth First Search or DFS

```
def dfs(visited, node, res,  graph_dict):
        res.append(node)
        visited.add(node)
```

```
        for adj in graph_dict[node]:
                if adj not in visited:
                        dfs(visited, adj, res, graph_dict)


res = []
visited = set()
dfs(visited, start_node, res, graph_dict)
print(res)
```

Problems
Flood Fill
Number of Islands
Max Area of Island
Rotting Oranges
Number of Closed Islands
Shortest Path in Binary Matrix
0-1 Matrix
Longest Increasing Path in a Matrix
Evaluate Division
Accounts Merge

Topological Sort

■ Topological sorting for *DAG* is a linear ordering of vertices such that for every directed edge `u-->v`, vertex u comes before v in the ordering.
■ Topological Sorting for a graph is not possible if the graph is not a DAG.

1. Using DFS

```
'''
TC: O(N + E)
SC: O(N)
'''
def findTopo(node):
        visited.add(node)
```

```
        for n in graph_dict[node]:
                if n not in visited:
                        findTopo(n)
        stack.append(node)


stack = []
visited = set()
for node in nodes:
        if node not in visited:
                findTopo(node)
return stack[::-1]
```

2. BFS - Kahn's Algo

```
'''
Store indegree of every node in an array.
Topological sort always start with a node having indegree = 0, add that node(s) in qu
eue.
Traverse (BFS), At any point, if indegree of a node becomes 0, add that to queue
TC: O(N+E)
SC: O(N)
'''


result = []
q = deque([])
indegree = defaultdict(int)
#Find indegree of every node
for node in nodes:
        for n in graph_dict[node]:
```

```
                indegree[n]+=1


#Add 0-indegree-node in queue
for node in nodes:
        if indegree[node]==0:
                q.append(node)


while q:
        node = q.popleft()
        result.append(node)
        for n in graph_dict[node]:
                indegree[n]-=1
                if indegree[n]==0:
                        q.append(n)


return result
```

Problems
Course Schedule
Course Schedule II
Loud and Rich

👉 Shortest Path Algorithms

- Shortest path algorithms are a family of algorithms designed to solve the shortest path problem.
- There are two main types of shortest path algorithms, *single-source* and *all-pairs*.

1. Dijktra's Algorithm
- Single - source
- Finds the shortest paths from the source to all vertices in the given graph.
- Detailed explanation: https://leetcode.com/discuss/study-guide/1059477/A-guide-to-Dijkstra's-Algorithm

```
'''

In graph representation, use tuple (node, weight).

e.g.  {1:[(2,3), (3,3)], 2: [(1,3)], 3: [(1,3)]}

TC: O((N+E)LogN)

SC: O(N)

'''

result =[0]+ [float('inf')]*n    #n = no. of nodes

result[k]=0

heap = [(0, k)]


while heap:

        ncost, node = heapq.heappop(heap)

        if result[node] < ncost: continue

        for nxt, cost in graph_dict[node]:

                if result[nxt] > result[node] + cost:

                        result[nxt] = result[node] + cost

                        heapq.heappush(heap, (result[nxt], nxt))


return result    # Distance from source to every node
```

Problems
Network delay Time
Cheapest Flight Within K stops
Path with Maximum Probability
Swim in Rising Water

2. Bellman Ford

- Dijkstra's algorithm doesn't work for graphs with negative weight edges. For graphs with negative weight edges, Bellman–Ford algorithm can be used.
- Single Source - Finds the shortest path from the source to all vertices in the given graph.

■ Bellman-Ford does not work with undirected graph with negative edges as it will declared as negative cycle.

```
'''

Representation: graph = [[u,v,w]] represents that there is edge between node u to nod
e v with weight w

TC: O(NE)

SC: O(N)

'''

result = [0] + [float('inf')]*n

result[k] = 0



# This step guarantees shortest path if graph doesn't contain negative weight cycle

for node in range(1,n):

        for u,v,w in graph:

                if result[u] + w < result[v]:

                        result[v] = result[u] + w



# Check if there is negative weight cycle.

# After relaxing n-1 times, if we try to relax 1 more time and the distance reduces,
that means a negative weight cycle exists

for u, v, w in graph:

        if result[u] != float("Inf") and result[u] + w < result[v]:

                print("Graph contains negative weight cycle")

                return



return result
```

Problems
Network delay Time

3. Floyd Warshall

■ All Pairs - finds shortest distances between every pair of vertices in a given edge weighted directed Graph

```
'''

Representation: Adjacency Matrix, graph = [[0,3], [3,0]] => There is edge between 0-->1 with weight 3

TC: O(N^3)

SC: O(N^2)

'''

result = [[wt for wt in row] for row in graph]    # Initialize same as graph

for i in range(n):

    result[i][i] = 0    # Distance from node itself is zero


for k in range(n):

   for i in range(n):        # Source node

         for j in range(n):    # Destination node

                 result[i][j] = min(result[i][j], result[i][k] + result[k][j]


return result
```

👉 Union-Find / Disjoint Set

■ Detailed Video Explanations: https://leetcode.com/explore/learn/card/graph/618/disjoint-set/3880/

```
'''

Taken from Graph Explore Card

Union By Rank

Path Compression Optimization
```

```
'''



class UnionFind:

        def __init__(self, size):

                self.root = [i for i in range(size)]

                # Use a rank array to record the height of each vertex, i.e., the "r
ank" of each vertex.

                # The initial "rank" of each vertex is 1, because each of them is

                # a standalone vertex with no connection to other vertices.

                self.rank = [1] * size



    # The find function here is the same as that in the disjoint set with path compre
ssion.

        def find(self, x):

                if x == self.root[x]:

                        return x

                self.root[x] = self.find(self.root[x])

                return self.root[x]



    # The union function with union by rank

    def union(self, x, y):

        rootX = self.find(x)

        rootY = self.find(y)

        if rootX != rootY:

            if self.rank[rootX] > self.rank[rootY]:

                self.root[rootY] = rootX

            elif self.rank[rootX] < self.rank[rootY]:

                self.root[rootX] = rootY
```

```
            else:
                self.root[rootY] = rootX
                self.rank[rootX] += 1

    def connected(self, x, y):
        return self.find(x) == self.find(y)
```

Problems
Number of Provinces
Number of Connected Components in Undirected Graph
Accounts Merge
Reduntant Connection
Most Stones Removed with Same Row or Column
Largest Component Size by Common Factor

👉 Minimum Spanning Tree

- A Spanning tree is a subset to a connected graph G, where all the edges are connected, i.e, we can traverse to any edge from a particular edge with or without intermediates.
- A minimum spanning tree (MST) for a weighted, connected, undirected graph is a spanning tree with a weight less than or equal to the weight of every other spanning tree.
- Cycle shouldn't be present
- If there are `n` nodes, there will be `n-1` edges

1. Prim's Algo
- Detailed Explanation: https://leetcode.com/discuss/study-guide/1131969/Minimum-Spanning-Tree-beginner's-guide

```
'''

Graph Representation - Same as Dijktra's

TC: O((N+E)logN)

SC: O(N)

'''


minCost = 0
```

```
visited = set()

heap = [(0,k)]     #(cost, node), k = start_node


while heap and len(visited)<n:    # n = number of nodes

        cost, node = heapq.heappop(heap)

        if node not in visited:

                minCost += cost

                visited.add(node)

                for nxt, c in graph[node]:

                        if nxt not in visited:

                                heapq.heappush(heap, (c, nxt))


return minCost
```

2. Kruskal's Algo

```
'''

Representation: graph = [[u,v,w]] represents that there is edge between node u to nod
e v with weight w

Union Find declaration in above section

TC: O(ElogE)

SC: O(N)

'''

uf = UnionFind(n+1)    #n = number of nodes

minCost,e,result = 0,0,[]

graph = sorted(connections,key = lambda x:x[2])    # Sort the graph w.r.t edge weight

for u,v,w in graph:

        if uf.find(u) != uf.find(v):

                result.append((u,v))
```

```
                minCost += w
                uf.union(u,v)


#Print mst
for u,v in result:
        print(f'{u}-->{v}')


return minCost
```

Problems
Min Cost to Connect All Points

👉 More...

1. Cycle detection in Undirected Graph

```
''' --------------- Using BFS ------------'''
def checkCycle(node, visited, graph_dict):
        visited.add(node)
        q = deque([])
        q.append((node, -1))

        while q:
                node, parent = q.popleft()
                for n in graph_dict[node]:
                        if n not in visited:
                                visited.add(n)
                                q.append((n, node))
                        elif n!=parent:
```

```
                                return True
        return False


visited = set()
for node in nodes:
        if node not in visited:
                if checkCycle(node, visited, graph_dict):
                        return True
return False
'''------------------ Using DFS ------------------'''
def checkCycle(node, parent):
        visited.add(node)
        for n in graph_dict[node]:
                if n not in visited:
                        if checkCycle(n, node):
                                return True
                elif n != parent:
                        return True
        return False


visited = set()
for node in nodes:
        if node not in visited:
                if checkCycle(node, -1):
                        return True
return False
```

2. Cycle detection in Directed Graph

```
'''--------------- DFS ----------------'''

def checkCycle(node):

        visited.add(node)

        rec[node] = 1

        for n in graph_dict[node]:

                if n not in visited:

                        if checkCycle(n):

                                return True

                elif rec[n]:

                        return True

        rec[node] = 0

        return False


visited = set()

rec = [0]*len(nodes)

for node in nodes:

        if node not in visited:

                if checkCycle(node):

                        return True

return False
```

3. Bipartite Graph

■ A bipartite graph, also called a bigraph, is a set of graph vertices decomposed into two disjoint sets such that no two graph vertices within the same set are adjacent.

■ In other words, a graph which can be colored using exactly two colors such that no adjacent nodes have same color.

■ If a graph contains "ODD" cycle then it's not bipartite, else the graph is bipartite

Solution of Is Graph Bipartite

```
'''----------- BFS ------------'''
def bfsCheck(node):
        q = deque([])
        q.append(node)
        color[node] = 1
        while q:
                n = q.popleft()
                for it in graph[n]:
                        if color[it] == -1: #graph is not colored
                                color[it] = 1-color[n]
                                q.append(it)
                        elif color[it] == color[n]:  #same color of adjacent nodes
                                return False
        return True



n = len(graph)
color = [-1]*n
for i in range(n):
        if color[i] == -1:
                if not bfsCheck(i):
                        return False
return True
'''------------- DFS --------------'''
def dfsCheck(node):
```

```
        if color[node]==[-1]:
                color[node]=1
        for it in graph[node]:
                if color[it] == -1:
                        color[it] = 1 - color[node]
                        if not dfsCheck(it):
                                return False
                elif color[it] == color[node]:
                        return False
        return True


n = len(graph)
color = [-1]*n
for i in range(n):
        if color[i]==-1:
                if not dfsCheck(i):
                        return False
return True
```

1. **Union Find:**

   Identify if problems talks about finding groups or components.

   https://leetcode.com/problems/friend-circles/
   https://leetcode.com/problems/redundant-connection/
   https://leetcode.com/problems/most-stones-removed-with-same-row-or-column/
   https://leetcode.com/problems/number-of-operations-to-make-network-connected/
   https://leetcode.com/problems/satisfiability-of-equality-equations/
   https://leetcode.com/problems/accounts-merge/

   All the above problems can be solved by Union Find algorithm with minor tweaks. Below is a standard template for union find problems.

```
class Solution {

 vector<int>parent;

 int find(int x) {

         return parent[x] == x ? x : find(parent[x]);

 }

public:

 vector<int> findRedundantConnection(vector<vector<int>>& edges) {


         int n = edges.size();


         parent.resize(n+1, 0);

         for (int i = 0; i <= n; i++)

                 parent[i] = i;


         vector<int>res(2, 0);

         for (int i = 0; i < n; i++) {

                 int x = find(edges[i][0]);
```

```
                int y = find(edges[i][1]);
                if (x != y)
                        parent[y] = x;
                else {
                        res[0] = edges[i][0];
                        res[1] = edges[i][1];
                }
        }

        return res;
 }
};
```

2. **Depth First Search**

    1. **Start DFS from nodes at boundary:**
    https://leetcode.com/problems/surrounded-regions/
    https://leetcode.com/problems/number-of-enclaves/

```
2.  class Solution {
3.     int rows, cols;
4.     void dfs(vector<vector<int>>& A, int i, int j) {
5.             if (i < 0 || j < 0 || i >= rows || j >= cols)
6.                     return;
7.
8.             if (A[i][j] != 1)
9.                     return;
10.
11.            A[i][j] = -1;
12.            dfs(A, i+1, j);
13.            dfs(A, i-1, j);
```

```
14.             dfs(A, i, j+1);
15.             dfs(A, i, j-1);
16.     }
17. public:
18.     int numEnclaves(vector<vector<int>>& A) {
19.
20.             if (A.empty()) return 0;
21.
22.             rows = A.size();
23.             cols = A[0].size();
24.             for (int i = 0; i < rows; i++) {
25.                     for (int j = 0; j < cols; j++) {
26.                             if (i == 0 || j == 0 || i == rows-1 || j ==
    cols-1)
27.                                     dfs(A, i, j);
28.                     }
29.             }
30.
31.             int ans = 0;
32.             for (int i = 0; i < rows; i++) {
33.                     for (int j = 0; j < cols; j++) {
34.                             if (A[i][j] == 1)
35.                                     ans++;
36.                     }
37.             }
38.
39.             return ans;
40.     }
```

```
};
```

Time taken to reach all nodes or share information to all graph nodes:
https://leetcode.com/problems/time-needed-to-inform-all-employees/

```
class Solution {
    void dfs(unordered_map<int, vector<int>>&hm, int i, vector<int>& informTime, int &res, int curr) {

            curr += informTime[i];
            res = max(res, curr);

            for (auto it = hm[i].begin(); it != hm[i].end(); it++)
                    dfs(hm, *it, informTime, res, curr);
    }
public:
    int numOfMinutes(int n, int headID, vector<int>& manager, vector<int>& informTime) {

            unordered_map<int, vector<int>>hm;
            for (int i = 0; i < n; i++)
                    if (manager[i] != -1) hm[manager[i]].push_back(i);

            int res = 0, curr = 0;
            dfs(hm, headID, informTime, res, curr);
            return res;
    }
};
```

DFS from each unvisited node/Island problems
https://leetcode.com/problems/number-of-closed-islands/
https://leetcode.com/problems/number-of-islands/
https://leetcode.com/problems/keys-and-rooms/

https://leetcode.com/problems/max-area-of-island/
https://leetcode.com/problems/flood-fill/

```
65. class Solution {
66.     void dfs(vector<vector<char>>& grid, vector<vector<bool>>& visited, int i, int j, int m, int n) {
67.             if (i < 0 || i >= m || j < 0 || j >= n) return;
68.             if (grid[i][j] == '0' || visited[i][j]) return;
69.             visited[i][j] = true;
70.             dfs(grid, visited, i+1, j, m, n);
71.             dfs(grid, visited, i, j+1, m, n);
72.             dfs(grid, visited, i-1, j, m, n);
73.             dfs(grid, visited, i, j-1, m, n);
74.     }
75.     public:
76.     int numIslands(vector<vector<char>>& grid) {
77.             if (grid.empty()) return 0;
78.
79.             int m = grid.size();
80.             int n = grid[0].size();
81.             vector<vector<bool>>visited(m, vector<bool>(n, false));
82.
83.             int res = 0;
84.             for (int i = 0; i < m; i++) {
85.                     for (int j = 0; j < n; j++) {
86.                             if (grid[i][j] == '1' && !visited[i][j]) {
87.                                     dfs(grid, visited, i, j, m, n);
88.                                     res++;
89.                             }
```

```
                    }
            }

            return res;
    }
    };
```

96. **Cycle Find:**
    https://leetcode.com/problems/find-eventual-safe-states/

```
class Solution {
    bool dfs(vector<vector<int>>& graph, int v, vector<int>& dp) {

                    if (dp[v])
                            return dp[v] == 1;

                    dp[v] = -1;

                    for (auto it = graph[v].begin(); it != graph[v].end(); it++)
                            if (!dfs(graph, *it, dp))
                                    return false;

                    dp[v] = 1;

                    return true;
            }
    public:
            vector<int> eventualSafeNodes(vector<vector<int>>& graph) {

```

```
                int V = graph.size();

                vector<int>res;
                vector<int>dp(V, 0);

                for (int i = 0; i < V; i++) {
                        if (dfs(graph, i, dp))
                                res.push_back(i);
                }

                return res;
        }
};
```

3. **Breadth First Search**

    1. **Shortest Path:**
    https://leetcode.com/problems/01-matrix/
    https://leetcode.com/problems/as-far-from-land-as-possible/
    https://leetcode.com/problems/rotting-oranges/
    https://leetcode.com/problems/shortest-path-in-binary-matrix/

    Start BFS from nodes from which shortest path is asked for.
    Below is the sample BFS approach to find the path.

```
class Solution {
    public:
    vector<vector<int>> updateMatrix(vector<vector<int>>& matrix) {


        if (matrix.empty()) return matrix;
        int rows = matrix.size();
        int cols = matrix[0].size();
        queue<pair<int, int>>pq;
```

```
            for (int i = 0; i < rows; i++) {
                    for (int j = 0; j < cols; j++) {
                            if (matrix[i][j] == 0) {
                                    pq.push({i-1, j}), pq.push({i+1,
j}), pq.push({i, j-1}), pq.push({i, j+1});
                            }
                    }
            }


            vector<vector<bool>>visited(rows, vector<bool>(cols,
false));
            int steps = 0;
            while (!pq.empty()) {
                    steps++;
                    int size = pq.size();
                    for (int i = 0; i < size; i++) {
                            auto front = pq.front();
                            int l = front.first;
                            int r = front.second;
                            pq.pop();
                            if (l >= 0 && r >= 0 && l < rows && r <
cols && !visited[l][r] && matrix[l][r] == 1) {
                                    visited[l][r] = true;
                                    matrix[l][r] = steps;
                                    pq.push({l-1, r}), pq.push({l+1,
r}), pq.push({l, r-1}), pq.push({l, r+1});
                            }
                    }
            }
```

```
            return matrix;
        }
};
```

4. **Graph coloring/Bipartition**
   https://leetcode.com/problems/possible-bipartition/
   https://leetcode.com/problems/is-graph-bipartite/

   Problems asks to check if its possible to divide the graph nodes into 2 groups
   Apply BFS for same. Below is a sample graph coloring approach.

```
class Solution {
 public:
        bool isBipartite(vector<vector<int>>& graph) {
                int n = graph.size();
                vector<int>color(n, -1);

                for (int i = 0; i < n; i++) {
                        if (color[i] != -1) continue;

                        color[i] = 1;
                        queue<int>q;
                        q.push(i);

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

                                for (int j = 0; j < graph[t].size(); j++) {
                                        if (color[graph[t][j]] == -1) {
```

```
                                                color[graph[t][j]] = 1-
color[t];

                                                q.push(graph[t][j]);

                                            } else if (color[graph[t][j]] ==
color[t]) {

                                                return false;

                                            }

                                        }

                                }

                        }


                        return true;

                }

        };
```

5. **Topological Sort:**
   Check if its directed acyclic graph and we have to arrange the elements in an order in which we need to select the most independent node at first. Number of in-node 0

   https://leetcode.com/problems/course-schedule/
   https://leetcode.com/problems/course-schedule-ii/

   Below is sample approach. Find if cycle is present, if not apply topological sort.

```
class Solution {

 int V;

 list<int>*adj;


 bool isCyclicUtil(int v, vector<bool>&visited, vector<bool>&recStack) {


         visited[v] = true;

         recStack[v] = true;
```

```
        for (auto it = adj[v].begin(); it != adj[v].end(); it++) {
                if (!visited[*it] && isCyclicUtil(*it, visited, recStack))
                        return true;
                else if (recStack[*it])
                        return true;
        }

        recStack[v] = false;
        return false;
}

bool isCyclic() {
        vector<bool>visited(V, false);
        vector<bool>recStack(V, false);

        for (int i = 0; i < V; i++) {
                if (isCyclicUtil(i, visited, recStack))
                        return true;
        }

        return false;
}

void topologicalSortUtil(int v, vector<bool>&visited, vector<int>& res) {
        visited[v] = true;

        for (auto it = adj[v].begin(); it != adj[v].end(); it++)
```

```
                if (!visited[*it])
                        topologicalSortUtil(*it, visited, res);

        res.push_back(v);
}

vector<int>topologicalSort(int v) {
        vector<int>res;

        vector<bool>visited(V, false);
        topologicalSortUtil(v, visited, res);

        for (int i = 0; i < V; i++) {
                if (!visited[i])
                        topologicalSortUtil(i, visited, res);
        }

        return res;
}

public:
vector<int> findOrder(int numCourses, vector<vector<int>>& prerequisites) {
        V = numCourses;
        adj = new list<int>[V];

        unordered_map<int, vector<int>>hm;
```

```
        for (int i = 0; i < prerequisites.size(); i++) {
                adj[prerequisites[i][0]].push_back(prerequisites[i][1]);
                hm[prerequisites[i][1]].push_back(prerequisites[i][0]);
        }

        if (isCyclic()) return vector<int>();

        int i = 0;
        for (i = 0; i < V; i++) {
                if (hm.find(i) == hm.end())
                        break;
        }

        return topologicalSort(i);
  }
};
```

6. **Find Shortest Path (Dijkstra's/Bellman Ford)**
   https://leetcode.com/problems/network-delay-time/

   Dijkstras and Bellman Ford:

```
class Solution {
  public:
        int networkDelayTime(vector<vector<int>>& times, int N, int K) {

                priority_queue<pair<int, int>, vector<pair<int, int>>,
greater<pair<int, int>>>pq;
                vector<int>dist(N+1, INT_MAX);

                pq.push(make_pair(0, K));
```

```
                dist[K] = 0;

                unordered_map<int, vector<pair<int, int>>>hm;
                for (int i = 0; i < times.size(); i++)
                        hm[times[i][0]].push_back(make_pair(times[i][1],
times[i][2]));

                while (!pq.empty()) {
                        pair<int, int>p = pq.top();
                        pq.pop();

                        int u = p.second;
                        for (auto it = hm[u].begin(); it != hm[u].end();
it++) {
                                int v = it->first;
                                int w = it->second;

                                if (dist[v] > dist[u] + w) {
                                        dist[v] = dist[u] + w;
                                        pq.push(make_pair(dist[v], v));
                                }
                        }
                }

                int res = 0;
                for (int i = 1; i <= N; i++)
                        res = max(res, dist[i]);
```

```
                return res == INT_MAX ? -1 : res;
        }
};



class Solution {
public:
        int networkDelayTime(vector<vector<int>>& times, int N, int K) {

                int n = times.size();
                if (!n) return 0;

                vector<int>dist(N+1, INT_MAX);
                int res = 0;

                dist[K] = 0;
                for (int i = 0; i < N; i++) {
                        for (int j = 0; j < n; j++) {
                                int u = times[j][0];
                                int v = times[j][1];
                                int w = times[j][2];
                                if (dist[u] != INT_MAX && dist[u] + w <
dist[v])
                                        dist[v] = w + dist[u];
                        }
                }

                for (int i = 1; i <= N; i++)
```

```
                            res = max(res, dist[i]);


                    return res == INT_MAX ? -1 : res;
            }
    }
```

Complete List: Below are mostly list of problems (mostly medium level and may 1 or 2 easy) which are better to start practice with:
(Updated on 14th June '20)

Union Find:

1. https://leetcode.com/problems/friend-circles/
2. https://leetcode.com/problems/redundant-connection/
3. https://leetcode.com/problems/most-stones-removed-with-same-row-or-column/
4. https://leetcode.com/problems/number-of-operations-to-make-network-connected/
5. https://leetcode.com/problems/satisfiability-of-equality-equations/
6. https://leetcode.com/problems/accounts-merge/
7. https://leetcode.com/problems/connecting-cities-with-minimum-cost/

DFS:
DFS from boundary:

1. https://leetcode.com/problems/surrounded-regions/
2. https://leetcode.com/problems/number-of-enclaves/

Shortest time:

1. https://leetcode.com/problems/time-needed-to-inform-all-employees/

Islands Variants

1. https://leetcode.com/problems/number-of-closed-islands/
2. https://leetcode.com/problems/number-of-islands/
3. https://leetcode.com/problems/keys-and-rooms/
4. https://leetcode.com/problems/max-area-of-island/
5. https://leetcode.com/problems/flood-fill/
6. https://leetcode.com/problems/coloring-a-border/

Hash/DFS:

1. https://leetcode.com/problems/employee-importance/
2. https://leetcode.com/problems/find-the-town-judge/

Cycle Find:

1. https://leetcode.com/problems/find-eventual-safe-states/

BFS:
BFS for shortest path:

1. https://leetcode.com/problems/01-matrix/
2. https://leetcode.com/problems/as-far-from-land-as-possible/
3. https://leetcode.com/problems/rotting-oranges/
4. https://leetcode.com/problems/shortest-path-in-binary-matrix/

Graph coloring:

1. https://leetcode.com/problems/possible-bipartition/
2. https://leetcode.com/problems/is-graph-bipartite/

Topological Sort:

1. https://leetcode.com/problems/course-schedule-ii/

Shortest Path:

1. https://leetcode.com/problems/network-delay-time/
2. https://leetcode.com/problems/find-the-city-with-the-smallest-number-of-neighbors-at-a-threshold-distance/
3. https://leetcode.com/problems/cheapest-flights-within-k-stops/

Please correct the approach/solution if you find anything wrong.

Similar POST
DP: https://leetcode.com/discuss/general-discussion/662866/Dynamic-Programming-for-Practice-Problems-Patterns-and-Sample-Solutions
Sliding Window: https://leetcode.com/discuss/general-discussion/657507/Sliding-Window-for-Beginners-Problems-or-Template-or-Sample-Solutions

I created a small list of graph problems that can be useful to memorize/practice to solve more graph problems (from my point of view).
We should not memorize the algorithm itself, but rather the idea and some implementation notes.
So under each algorithm I put notes which can help to remember the algorithm.

P.S.: Please put comments how do you remember algorithms or ideas, and I will update the this post with interesting notes.

Update: updated with Eulearian Path, visualization of BFS, UnionFind, Dijkstra, Topological Sort and Bellman Ford.

Graph problems

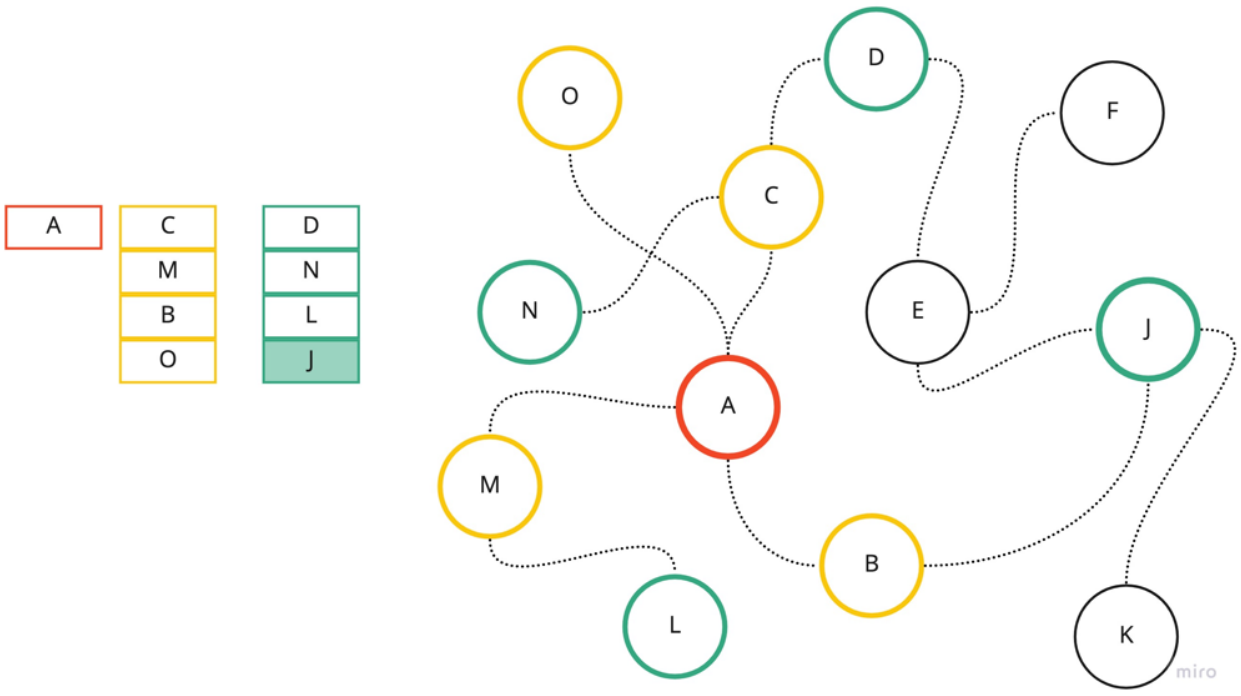
Description:
In computer science, graphs are used to represent networks of communication, data organization, computational devices, the flow of computation, etc. For instance, the link structure of a website can be represented by a directed graph, in which the vertices represent web pages and directed edges represent links from one page to another.

Common algorithms

BFS:
Breadth-first search (BFS) is an algorithm for searching a tree data structure for a node that satisfies a given property. It starts at the tree root and explores all nodes at the present depth prior to moving on to the nodes at the next depth level. Extra memory, usually a queue, is needed to keep track of the child nodes that were encountered but not yet explored.
Part of a solution for the problem "Evaluate Division": https://leetcode.com/problems/evaluate-division/

```
double bfs(unordered_map<string, vector<ip>>& adj, vector<string>& query) {

    unordered_set<string> visited;

    string start = query[0];

    string end = query[1];

    queue<ip> q;

    q.push({start, 1.0});

    visited.insert(start);

    while (!q.empty()) {

        int sz = q.size();

        for (int i = 0; i < sz; i++) {
```

```
            auto [node, cost] = q.front(); q.pop();
            if (!adj.count(node)) continue;
            if (node == end) return cost;
            for (auto& a : adj[node]) {
                if (!visited.count(a.first)) {
                    q.push({a.first, cost * a.second});
                    visited.insert(node);
                }
            }
        }
    }
    return -1.0;
}
```



Visualization of BFS with first 3 layers. A - root node/first lavel, C/M/B/O - second layer, D/N/L/J - third layer.

Notes:

1. Algorithm uses queue for implementation.
2. It checks whether a vertex has been explored before enqueueing the vertex.
3. We can track if the node was already explored by modifying the original matrix.
4. BFS algorithm can be instructed with additional array dist which can help to track the parent node of the next node. This will help to reconstruct the path by looping backward from end node.

DFS:
Is an algorithm for traversing or searching tree or graph data structures. The algorithm starts at the root node (selecting some arbitrary node as the root node in the case of a graph) and explores as far as possible along each branch before backtracking.

```
void dfs(unordered_map<int, vector<int>>& graph, int node, unordered_set<int>& visite
d) {

    visited.insert(node);

    for (auto& n : graph[node]) {

        if (!visited.count(n)) {

            dfs(graph, n, visited);

        }

    }

}

// Find connected components.

int count = 0;

for (int i = 0; i < n; i++) {

    if (!visited.count(i)) {

                count++;

        dfs(graph, i, visited);

    }

}
```

Notes:

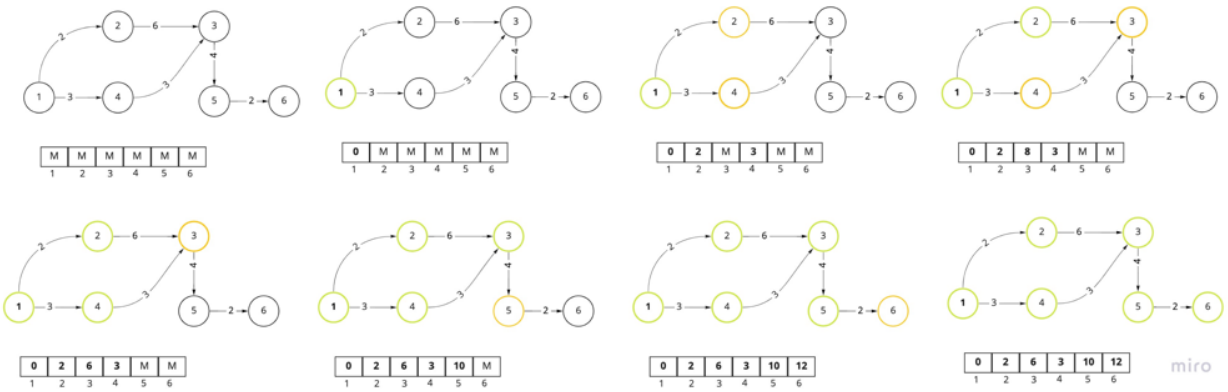
1. Algorithm usually uses recursion implementation.
2. We mark the node as visited and will keep exploring its neighbors if there are not yet explored.

3. DFS can be useful to find connected components. We can iterate through the nodes and call dfs() to find all nodes which belongs to component.
4. We can use unordered_map<int, vector> to represent the graph.

Dijkstra's algorithm:
Is an algorithm for finding the shortest paths between nodes in a graph, which may represent, for example, road networks.

```
class Solution {

public:

    int networkDelayTime(vector<vector<int>>& times, int n, int k) {

        using ip = pair<int, int>;

        vector<vector<ip>> adj(n + 1);

        for (auto& t : times) adj[t[0]].push_back({t[1], t[2]});

        priority_queue<ip, vector<ip>, greater<ip>> pq;

        vector<int> dist(n + 1, INT_MAX);

        vector<bool> visited(n + 1, false);

        pq.push({0, k});

        dist[k] = 0;

        while (!pq.empty()) {

            auto [n_cost, node] = pq.top(); pq.pop();

            visited[node] = true;

            if (dist[node] < n_cost) continue;  // Optimization: skip node if we alre
ady find better option.

            for (auto& [next, cost] : adj[node]) {

                if (visited[next] == true) continue; // Optimization: do not re-visit
nodes.

                if (dist[next] > dist[node] + cost) {

                    dist[next] = dist[node] + cost;

                    pq.push({dist[next], next});

                }
```

```
            }
        }
        int res = 0;
        for_each(dist.begin() + 1, dist.end(), [&](int d) {
            res = max(res, d);
        });
        return res == INT_MAX ? -1 : res;
    }
};
```



Visualization of traversing graph using Dijkstra algorithm and distance array. The priority queue itself is not shown there, as well as redundant nodes that we might have in priority queue. Green node - explored/current node, Yellow - node in the priority queue.

Notes:

1. We will have a set to track visited nodes.
2. We will create a distance array to track the distance to each node. Initial node will have 0, others maximum. There is a room for optimization: if the node we got from the pq has larger cost than in our dist[] array, we should not explore it as we already got a better option.
3. We will use min priority_queue to get the node with the minimum distance from the current node.
4. If we are only interested in shortes distance till some END node, we can terminate the search earlier: if (node == dst) return cost;
5. If we already find a better path we shouldn't explore it further: if (dist[node] < stops) continue;

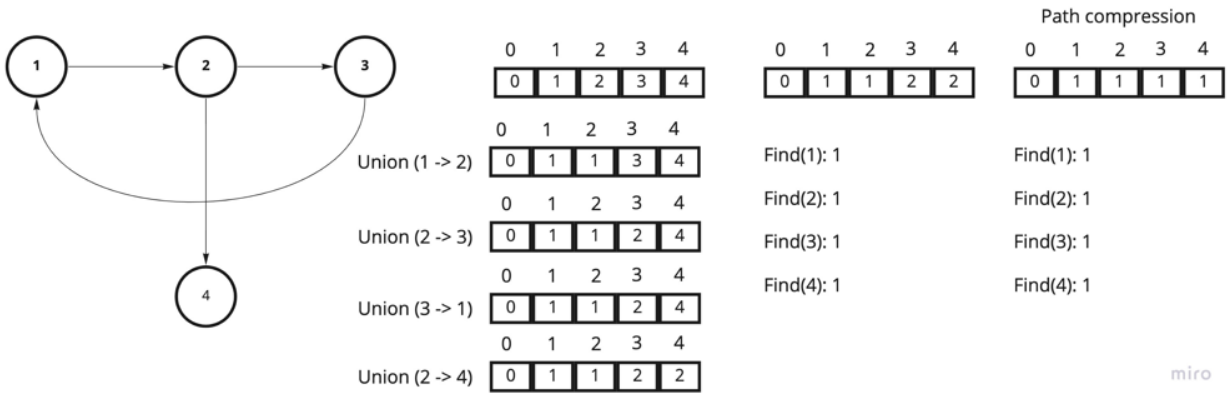
Union-Find:

Union–find data structure or disjoint-set data structure or merge–find set, is a data structure that stores a collection of disjoint (non-overlapping) sets. Equivalently, it stores a partition of a set into

disjoint subsets. It provides operations for adding new sets, merging sets (replacing them by their union), and finding a representative member of a set. Helps to find the number of connected components, and can help to find MST.

```
class UnionFind {

    public:

    UnionFind(int n) : parent(n) {

        iota(parent.begin(), parent.end(), 0);

    }


    int Find(int x) {

        int temp = x;

        while (temp != parent[temp]) {

            temp = parent[temp];

        }

                // Path compression below

        while (x != temp) {

            int next = parent[x];

            parent[x] = temp;

            x = next;

        }

        return x;

    }


    void Union(int x, int y) {

        int xx = Find(x);

        int yy = Find(y);

        if (xx != yy) {

            parent[xx] = yy;
```

```
        }
    }


    private:
            vector<int> parent;
};
```



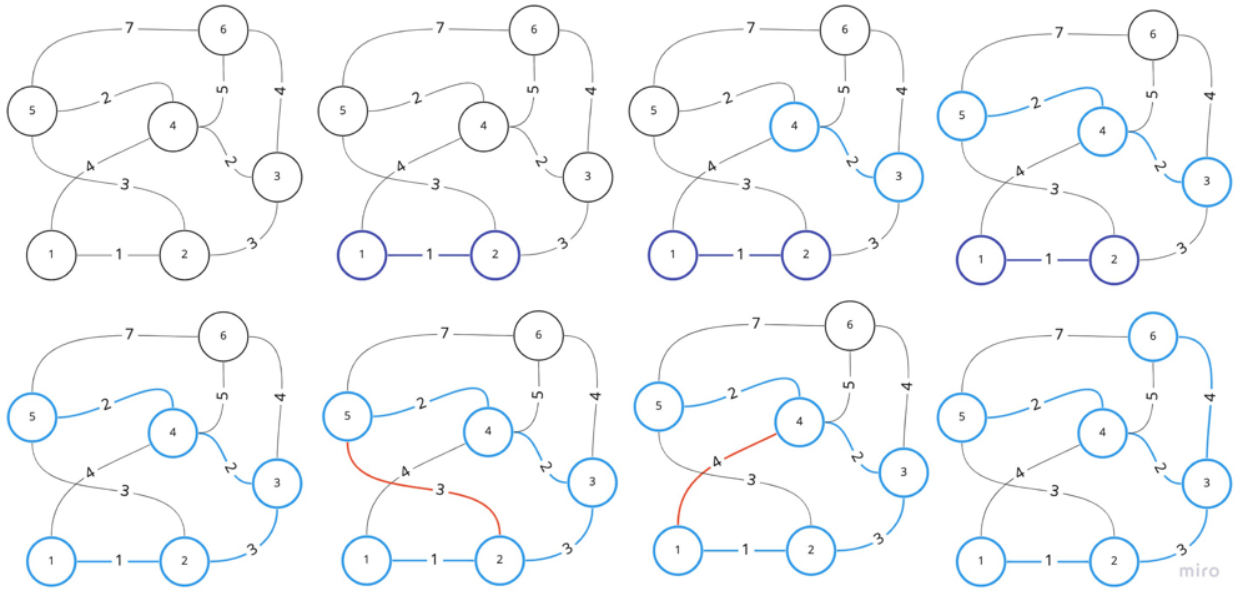
The above picture demonstrates the state of the parent array after multiple Union() calls, follows multiple Find() calls.
Notes:
1. We can use vector to hold the set of nodes or unordered_map<int, int> if we don't know the amount of nodes.
2. If the parent[id] == id, we know that id is the root node.
3. The data structure using two methods Union() - union to nodes/components, and Find() - find the root node.
4. We can do path compression, so after some number of Find() calls it will be O(1) to call Find() again.

Minimum Spanning Tree:
A minimum spanning tree (MST) is a subset of the edges of a connected, edge-weighted undirected graph that connects all the vertices together, without any cycles and with the minimum possible total edge weight.
Solution for Connecting Cities With Minimum Cost: https://leetcode.com/problems/connecting-cities-with-minimum-cost/.

```
class UnionFind {

public:

        UnionFind(int n) : parent(n) {

                iota(parent.begin(), parent.end(), 0);
```

```
        }
        int Find(int x) {
                int temp = x;
                while (temp != parent[temp]) {
                        temp = parent[temp];
                }
                while (x != temp) {
                        int next = parent[x];
                        parent[x] = temp;
                        x = next;
                }
                return temp;
        }
        bool Union(int x, int y) {
                int xx = Find(x);
                int yy = Find(y);
                if (xx == yy) return false;
                parent[xx] = yy;
                return true;
        }


private:
        vector<int> parent;
};
int minimumCost(int n, vector<vector<int>>& connections) {
        sort(connections.begin(), connections.end(), [](const auto& lhs, const auto&
rhs){
                return lhs[2] < rhs[2];
```

```
        });
        UnionFind uf(n + 1);
        int sum = 0, count = 0;
        for (auto& c : connections) {
                if (uf.Union(c[0], c[1])) {
                        count++;
                        sum += c[2];
                }
                if (count == n - 1) return sum; // Return earlier once graph is conn
ected.
        }
        return -1;
}
```



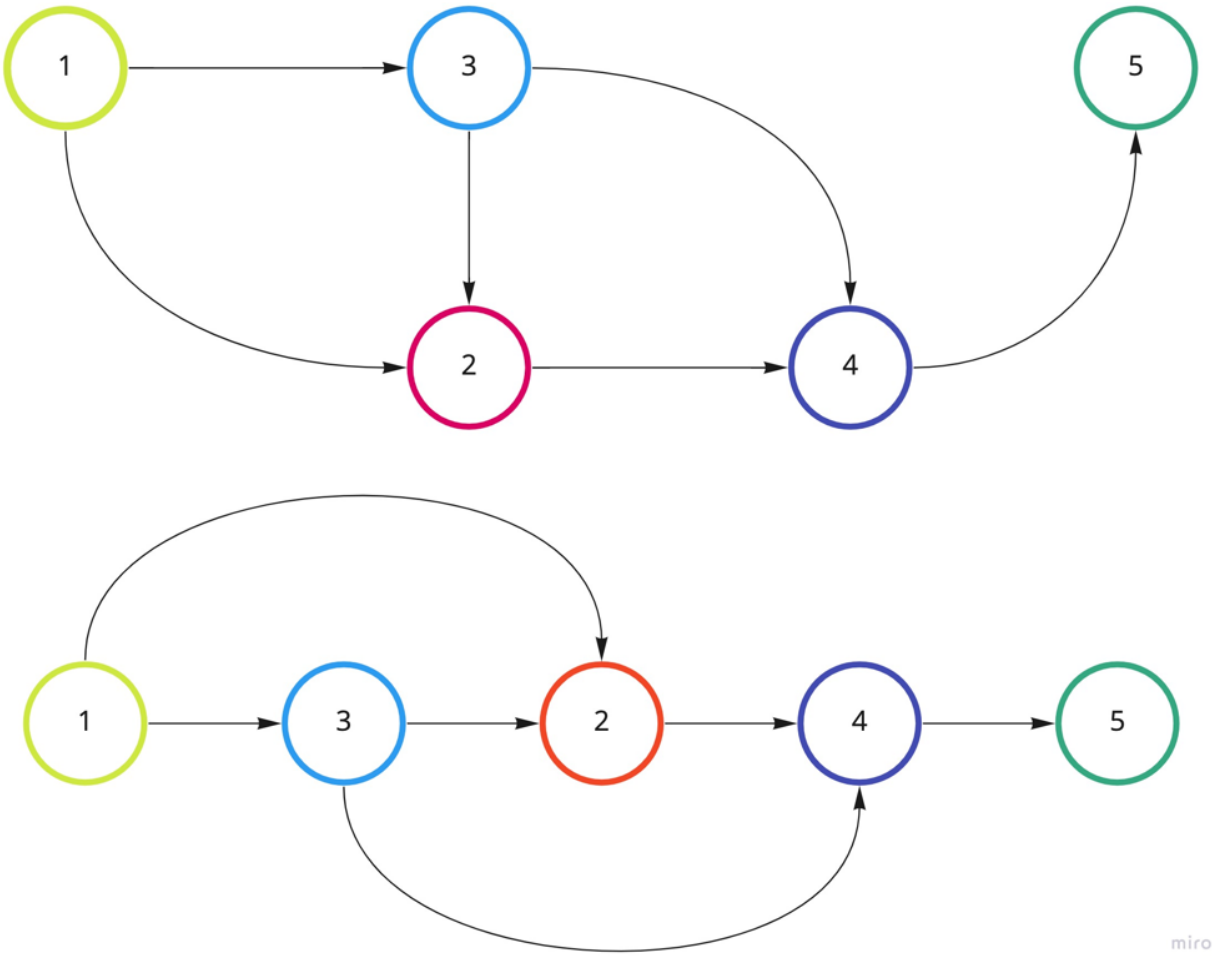
Visualization of Kruskal's algorithm: we will try to union nodes if they are not connected.

Notes:

1. One of the implementation of MST algorithm use Union Find algorithm (Kruskal's Algorithm).
2. We need to sort elements by the weight before appying the algorithm, or we can use min priority_queue.

Topological sort
Is a linear ordering of its vertices such that for every directed edge uv from vertex u to vertex v, u comes before v in the ordering.

```
    // Kahn's Algorithm

    vector<vector<int>> adj(numCourses);

    vector<int> indegree(numCourses, 0);

    for (auto& p : prerequisites) {

        indegree[p[1]]++;

        adj[p[0]].push_back(p[1]);

    }

    queue<int> q;

    for (int i = 0; i < numCourses; i++) {

        if (indegree[i] == 0) q.push(i);

    }

    int prereq = 0;

    while (!q.empty()) {

        int el = q.front();

        q.pop();

        prereq++;

        for (auto& next : adj[el]) {

            if (--indegree[next] == 0) {

                q.push(next);

            }

        }

    }

    return prereq == numCourses;
```



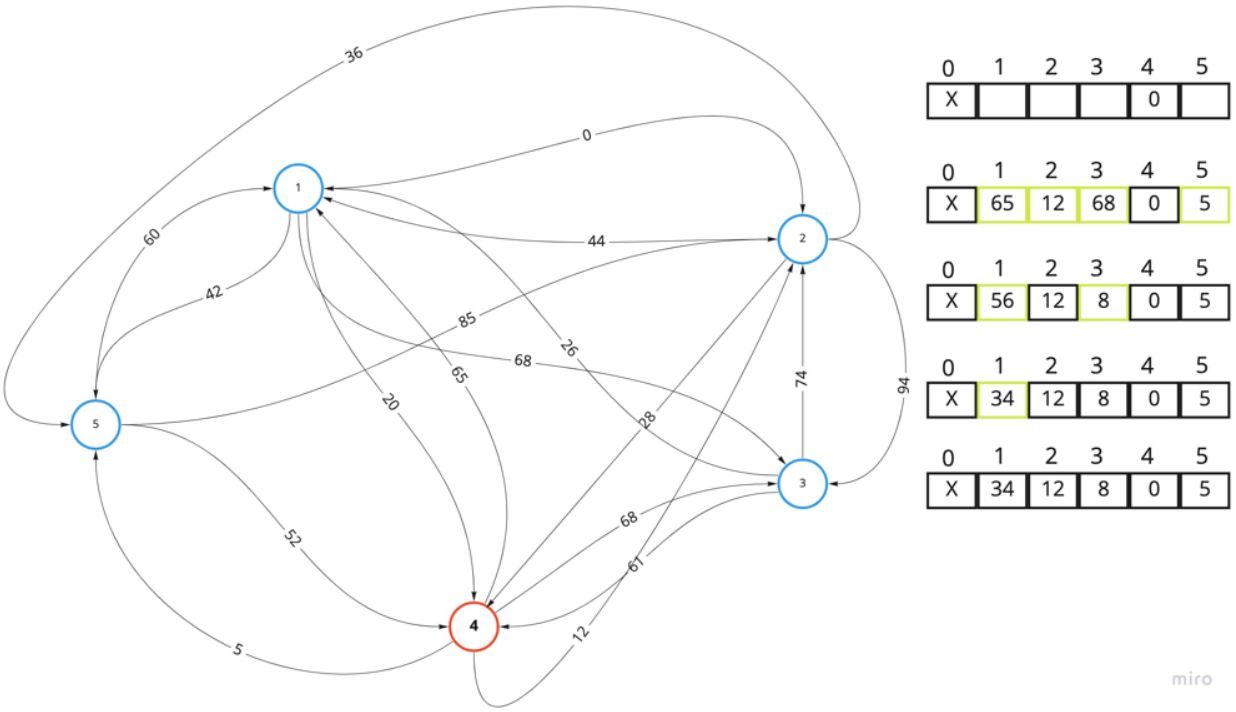
The above picture demonstrates linear order of the given graph. The vertices can be tasks, and edges can represent some contraints, such as U should be finished before V in (U -> V).
Notes:
1. We will have the indegree array to count, which nods should be visited first.
2. We will have a queue to push the nodes that don't have any dependencies.

Bellman Ford

Is an algorithm that computes shortest paths from a single source vertex to all of the other vertices in a weighted digraph.

```
class Solution {

public:

    int networkDelayTime(vector<vector<int>>& times, int n, int k) {

        vector<int> dist(n + 1, INT_MAX);

        dist[k] = 0;
```

```
        for (int i = 1; i <= n; i++) {
            for (auto& t : times) {
                if (dist[t[0]] != INT_MAX && dist[t[1]] > dist[t[0]] + t[2]) {
                    dist[t[1]] = dist[t[0]] + t[2];
                }
            }
        }
        int res = 0;
        for (int i = 1; i <= n; i++) {
            res = max(res, dist[i]);
        }
        return res == INT_MAX ? -1 : res;
    }
};
```



The above picture demonstrates how the dist array incrementally updated with better values. If not a single value is updated during iteration - we can stop earlier.

Notes:
1. We will use the array to hold the distance between particular start node and all others.
2. We will try to improve distance n times between all nodes in the graph.

Floyd Warshall

Is an algorithm for finding shortest paths in a directed weighted graph with positive or negative edge weights.

```
class Solution {

public:

    int networkDelayTime(vector<vector<int>>& times, int n, int k) {

        vector<vector<long>> dist(n, vector<long>(n, INT_MAX));

        for (auto& t : times)

            dist[t[0] - 1][t[1] - 1] = t[2];

        for (int i = 0; i < n; i++)

            dist[i][i] = 0;

        for (int k = 0; k < n; k++) {

            for (int i = 0; i < n; i++) {

                for (int j = 0; j < n; j++) {

                    dist[i][j] = min(dist[i][j], dist[i][k] + dist[k][j]);

                }

            }

        }

        long res = INT_MIN;

        for (int i = 0; i < n; i++) {

            if (dist[k - 1][i] == INT_MAX) return -1;

            res = max(res, dist[k - 1][i]);

        }

        return (int)res;

    }
```

```
};
```

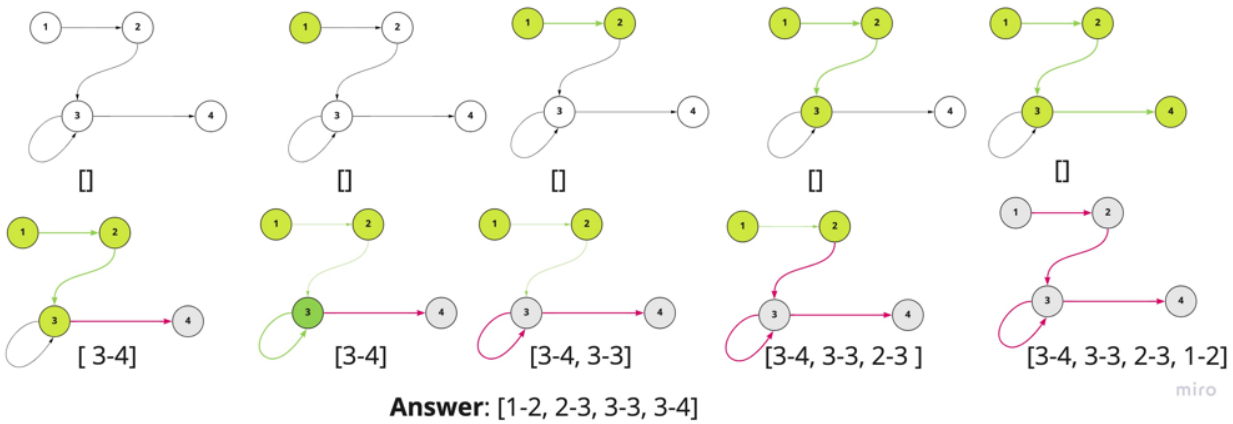
Vizualization for Floyd Warshall is slightly different from Bellman Ford, but the idea stays the same.
Notes:
1. We will use vector<vector> to keep track of distance between nodes i and j.
2. We will have a 3 loops, checks if we can improve the distance between i and j by using k node.

Eulearian Path

Is an algorithm that finds a path that uses every ***edge*** in a graph only once.
The algorithm below is a solution for the "**Reconstruct Itinerary**": https://leetcode.com/problems/reconstruct-itinerary/

```
class Solution {

public:

    void dfs(unordered_map<string, multiset<string>>& graph,

             vector<string>& res, string start) {

        while (graph[start].size() > 0) {

            auto next = *graph[start].begin();

            graph[start].erase(graph[start].begin());

            dfs(graph, res, next);

        }

        res.push_back(start);

    }

    vector<string> findItinerary(vector<vector<string>>& tickets) {

        unordered_map<string, multiset<string>> graph;

        for (const auto& t : tickets) graph[t[0]].insert(t[1]);

        vector<string> res;

        dfs(graph, res, "JFK");

        reverse(res.begin(), res.end());

        return res;

    }
```

```
};
```



Answer: [1-2, 2-3, 3-3, 3-4]

The above picture is the visualization of eulerian path algorithm. You can observe how the result is constructed on the backtracking.

Notes:

1. The algorithm almost identical to the dfs traversal with one main instrumentation: we are building the path on the backtrack of the dfs algorithm: `res.push_back(start);`
2. That is why we should reverse the list at the end of the traversal: `reverse(res.begin(), res.end());`
3. In the above implementation we are using multiset (because of the problem), but the general implementation may use vector<> and additional vector<> to track the outgoing degrees, and use it for two main purposes: as index in the adj list, and to track how many node we not visited yet.

BFS problems

- Flood Fill: https://leetcode.com/problems/flood-fill/
- Number of Islands: https://leetcode.com/problems/number-of-islands/
- Word Ladder I: https://leetcode.com/problems/word-ladder/
- Word Ladder II: https://leetcode.com/problems/word-ladder-ii/
- Evaluate Division: https://leetcode.com/problems/evaluate-division/
- Get Watched Videos by Your Friends: https://leetcode.com/problems/get-watched-videos-by-your-friends/
- Cut Off Trees for Golf Event: https://leetcode.com/problems/cut-off-trees-for-golf-event/

DFS problems

- Number of Islands: https://leetcode.com/problems/number-of-islands/
- Flood Fill: https://leetcode.com/problems/flood-fill/
- Longest Increasing Path in a Matrix: https://leetcode.com/problems/longest-increasing-path-in-a-matrix/
- Evaluate Division: https://leetcode.com/problems/evaluate-division/
- Robot Room Cleaner: https://leetcode.com/problems/robot-room-cleaner/
- Most Stones Removed with Same Row or Column: https://leetcode.com/problems/most-stones-removed-with-same-row-or-column/

- Reconstruct Itinerary: https://leetcode.com/problems/reconstruct-itinerary/
- Tree Diameter: https://leetcode.com/problems/tree-diameter/
- Accounts Merge: https://leetcode.com/problems/accounts-merge/

Connected components problems

- Number of Provinces: https://leetcode.com/problems/number-of-provinces/
- Number of Connected Components in an Undirected Graph: https://leetcode.com/problems/number-of-connected-components-in-an-undirected-graph/
- Number of Operations to Make Network Connected: https://leetcode.com/problems/number-of-operations-to-make-network-connected/
- Accounts Merge: https://leetcode.com/problems/accounts-merge/
- Critical Connections in a Network: https://leetcode.com/problems/critical-connections-in-a-network/

Dijkstra's problems

- Path With Maximum Minimum Valued: https://leetcode.com/problems/path-with-maximum-minimum-value/
- Network delay time: https://leetcode.com/problems/network-delay-time/
- Path with Maximum Probability: https://leetcode.com/problems/path-with-maximum-probability/
- Path With Minimum Effort: https://leetcode.com/problems/path-with-minimum-effort/
- Cheapest Flights Within K Stops: https://leetcode.com/problems/cheapest-flights-within-k-stops/

Union Find problems

- Number of Islands: https://leetcode.com/problems/number-of-islands/
- Largest Component Size by Common Factor: https://leetcode.com/problems/largest-component-size-by-common-factor/
- Most Stones Removed with Same Row or Column: https://leetcode.com/problems/most-stones-removed-with-same-row-or-column/
- Number of Connected Components in an Undirected Graph: https://leetcode.com/problems/number-of-connected-components-in-an-undirected-graph/

Minimum Spanning Tree problems

- Connecting Cities With Minimum Cost: https://leetcode.com/problems/connecting-cities-with-minimum-cost/
- Min Cost to Connect All Points: https://leetcode.com/problems/min-cost-to-connect-all-points/

Topological sort problems

- Course Schedule : https://leetcode.com/problems/course-schedule/
- Course Schedule II: https://leetcode.com/problems/course-schedule-ii/
- Sequence Reconstruction: https://leetcode.com/problems/sequence-reconstruction/
- Alien Dictionary: https://leetcode.com/problems/alien-dictionary/solution/

Floyd Warshall problems

- Find the City With the Smallest Number of Neighbors at a Threshold Distance: https://leetcode.com/problems/find-the-city-with-the-smallest-number-of-neighbors-at-a-threshold-distance/
- Network delay time: https://leetcode.com/problems/network-delay-time/

Bellman Ford problems

- Network delay time: https://leetcode.com/problems/network-delay-time/