Graph Problems

1. Rotting Oranges

You are given an m x n grid where each cell can have one of the three value

0 → Empty cell

1 → Fresh Orange

2 → Rotten orange.

Every minute, any fresh orange that is 4-directionally adjacent to a rotten orange becomes rotten.

Return the minimum number of minutes that must elapse until no cell has a fresh orange. If this is impossible, return -1.

I/P:

2	1	1
1	1	0
0	1	1

O/P :

Exp:

t=1

2	1	1
1	1	0
0	1	1

⇒

t=2

2	2	2
2	2	0
0	1	1

⇓ t=3

2	2	2
2	2	0
0	1	1

⇐

t=4

2	2	2
2	2	0
0	2	1

Solution: First thing ⇒ BFS or DFS ?

We are changing all 1 → 2 at a time, which is possible, which means we are going level by level, so we must use BFS.

Algo:

(i) We need to start BFS from any grid[i][j] == 2,

(ii) First traverse grid & push all nodes to queue where grid[i][j] == 2 & also push the time as t=0, as they are rotten at t=0

Also keep track how many fresh oranges are there i.e. grid[i][j] == 1.

queue<pair<pair<int, int>, int>> q;
(x) (y) (time)

(iii) Now, start BFS, take newX & newY, which we can get from all possible directions.

directions = {(1,0), (0,1), (-1,0) (0,-1)}

(iv) check for boundary conditions & also skip grid[i][j] == 0 i.e (empty cells) & visited[i][j] = true.

In this way we will ensure, newX, newY will only go to cell having 1.

why? Because. all previous (2's are marked as visited, so we wount be going to any newX, newY which has value 2 and also unvisited.

Now decrement fresh--, & and at the end check if fresh > 0 if yes return -1;

Also keep track how many fresh oranges are there i.e. grid[i][j] == 1.

queue<pair<pair<int, int>, int>> q;
(x) (y) (time)

(iii) Now, start BFS, take newx & newy, which we can get from all possible directions.

directions = {(1,0), (0,1), (-1,0) (0,-1)}

(iv) check for boundary conditions & also skip grid[i][j] == 0 i.e empty cells. & visited[i][j] = true.

In this way we will ensure, newx, newy will only go to cell having 1.

Why? Because. all previous 2's are marked as visited, so we wount be going to any newx, newy which has value 2 and also unvisited.

Now decrement fresh--, & and at the end check if fresh > 0 if yes

return -1;

C++ code

```
int orangesRotting(vector<vector<int>>& grid){
int n= grid.size(), m=grid[0].size(), ans=0, fresh=0;
vector<vector<bool>>visited(n, vector<bool>(m,false));
queue<pair<pair<int,int>,int>> q;

for(int i=0;i<n;i++) {
    for(int j=0; j<m; j++) {
      if(grid[i][j]==1) fresh++;
      else if (grid[i][j]==2) {
          q.push({{i,j},0});
          visited[i][j]=true;
      }
    }
}
while(!q.empty()){
int x=q.front().first.first, y=q.front().first.first, time= q.front().second;
int directions[4][2] = {{1,0},{0,1},{-1,0},{0,-1}};
q.pop();
for(int i=0; i<4; i++) {
int newx=x+directions[i][0], newy=y+directions[i][i];
if(newx<0 || newx>=n || newy<0 || newy>=m || visited[newx][newy]
                                  || grid[newx][newy]==0)
   continue;
else {
      fresh--;
      grid[newx][newy]=2;
      visited[newx][newy]=true;
      ans= max(ans, time+1);
      q.push({{newx,newy}, time+1});
}}}
return    fresh>0 ? -1:ans;
}
```

0/1 Matrix

Given an m×n binary matrix mat, return the distance of the nearest 0 for each cell.

The distance b/w 2 cells sharing a common edge is 1.

I/P:

0	0	0
0	1	0
0	0	0

O/P

0	0	0
0	1	0
0	0	0

I/P

0	0	0
0	1	0
1	1	1

O/P

0	0	0
0	1	0
1	2	1

Solution We will follow same approach, will push all 0's to queue & mark them visited. In BFS we shall check only unvisited cells, in this way we will ensure we are not processing ~~visiting~~ any 0's again and will update the distance as +1 This will work as we are starting from 0's. And the node having no 0's around, will also get reached.

Since, the nearby ones will get visited & and from there we can go to next, next.

C++ code

```
vector<vector<int>> updateMatrix(vector<vector<int>>& grid)
{
int n= grid.size(), m= grid[0].size();
vector<vector<bool>> visited[n, vector<boo>(m,false)];
queue<pair<int,int>> q;
for(int i=0; i<n; i++) {
    for(int j=0; j<m; j++) {
        if(grid[i][j]==0) {
        q.push({i,j});
        visited[i][j]=true;
    }
    }
}
while(!q.empty()) {
int x=q.front().first, y=q.front().second;
int directions[4][2] = {{1,0}, {0,1}, {-1,0}, {0,-1}};
q.pop();
for(int i=0; i<4; i++) {
int newx = x+ directions[i][0], newy = y+direction[i][1];
if(newx<0 || newx>=n || newy<0 || newy>=m || visited[newx][newy])
    continue;
else {
    grid[newx][newy] = grid[x][y]+1;
    visited[newx][newy]=true;
    q.push({newx, newy});
}
}
}
return grid;
}
```

Cycle Detection in undirected graph (BFS)

```
bool BFS(vector<vector<int>> &adj, vector<int> &visited, int start,
                                                    int &V){
queue<pair<int,int>> q;
q.push ({start, -1});
visited [start] = 1;
while(!q.empty()) {
  int node = q.front().first; parent= q.front().second;
  q.pop();
for(int i=0; i<adj[node].size(); i++) {
        if(visited[adj[node][i]] = 0) {
        q.push ({adj[node][i], node});
        visited [adj[node][i]] = 1;
    } else if(visited[adj[node][i]] ==1 && adj[node][i]!=
                                                    parent)
        return true;
    }
}
return false;
}

bool isCycle (vector<vector<int>> &adj) {
  bool ans;
  int V = adj.size();
  vector<int> visited (V,0);
for(int i=0; i<V; i++) {
if(!visited[i] && BFS (adj, visited, i, V)) return true;
}
return false;
}
```

Cycle Detection undirected DFS

```
bool DFS(vector<vector<int>> &adj, vector<int> &visited,
         int node, int parent, int &V) {
    visited[node] = 1;
    for(int i=0; i< adj[node].size(); i++) {
        if(visited[adj[node][i]] == 0) {
            if(DFS(adj, visited, adj[node][i], node, V))
                return true;
        }
        else if(visited[adj[node][i]] == 1 &&
                adj[node][i] != parent)
            return true;
    }
    return false;
}
```

Number of Enclaves

You are given an $m \times n$ binary matrix grid, where 0 represents a sea cell and 1 represents a land cell.

A move consists of walking from one land cell to another adjacent (4-directionally) land cell or walking off the boundary of the grid.

Return the number of land cells in grid for which we cannot walk off the boundary of the grid in any number of moves.

I/P1

0	0	0	0
1	0	1	0
0	1	1	0
0	0	0	0

⇒ o/p

0	0	0	0
1	0	0	0
0	0	0	0
0	0	0	0

Solution2

Similar to ~~solo~~ surrounded region.

```
void visit(vector<vector<int>>& grid, int i, int j, int n, int m)
{
    grid[i][j] = 0;
    if (i >= 1 && grid[i-1][j] == 1) visit(grid, i-1, j, n, m);
    if (j >= 1 && grid[i][j-1] == 1) visit(grid, i, j-1, n, m);
    if (i < n-1 && grid[i+1][j] == 1) visit(grid, i+1, j, n, m);
    if (j < m-1 && grid[i][j+1] == 1) visit(grid, i, j+1, n, m);
}

int numEnclaves(vector<vector<int>>& grid) {
    int n = grid.size(), m = grid[0].size();
    for(int i=0; i<n; i++) {
        if (grid[i][0] == 1) visit(grid, i, 0, n, m);
        if (grid[i][m-1] == 1) visit(grid, i, m-1, n, m);
    }
    for (int i=0; i<m; i++) {
        if (grid[0][i] == 1) visit(grid, 0, i, n, m);
        if (grid[n-1][i] == 1) visit(grid, n-1, i, n, m);
    }
    for(int i=1; i<n; i++) {
        for (int j=1; j<m; j++) {
            if (grid[i][j] == 1) ans++;
        }
    }
    return ans;
}
```

Find the number of Island

Given a grid of size n*m consisting of '0's (water) & '1's (Land). Find the number of islands.

An island is either surrounded by water or the boundary of a grid & is formed by connecting adjacent lands horizontally or vertically or diagonally i.e. in all 8 directions.

I.P:- (i) 0 1 / 1 0 / 1 1 / 1 0 — O/P = 1

(ii) 0 1 1 1 0 0 0 / 0 0 1 1 0 1 0 — O/P = 2

Solution:- Just need to find connected components

```
int numIslands(vector<vector<char>>& grid) {
int n= grid.size(), m=grid[0].size(); ans=0;
vector<vector<bool>> visited(n, vector<bool>(m, false));
for(int i=0; i<n; i++) {
   for(int j=0; j<m; j++) {
      if(!visited[i][j] && grid[i][j]=='1') {
         BFS(grid, visited, i, j, n, m);
         ans++;
      }
   }
}
return ans;
}
```

```
void BFS (vector<vector<char>> &grid, int sr, int sc,
          vector<vector<bool>> &visited, int n, int m) {

  queue<pair<int,int>> q;
  q.push ({sr, sc});
  visited [sr] [sc] = true;

 While(!q.empty()) {
int x = q.front().first,  y = q.front().second;
int directions [8][2] = {{1,0}, {0,1}, {-1,0}, {0,-1},
                         {1,1}, {1,-1}, {-1,1}, {-1,-1}};

q.pop();

for(int i=0; i<8; i++) {
 int  newx = x+directions [i] [0];
 int  newy = y+ directions [i] [1];
if(newx<0 || newx>=n || newy<0 || newy>=m ||
   visited [newx] [newy] || grid [newx] [newy])
        continue;
else {
      visited [newx] [newy] =true;
      q.push ({newx, newy});

   }
 }
}
}
```

Number of distinct Islands

Given a boolean 2D matrix grid of size $n \times m$. You have to find the number of distinct islands where a group of connected 1s (horizontally or vertically) forms an island. 2 islands are considered to be distinct if & only if one island is not equal to another. (Not rotated or reflected).

Ex: (i) grid[][] =

1	1	0	0	0
1	1	0	0	0
0	0	0	1	1
0	0	0	1	1

O/P 1

(ii)

1	1	0	1	1
1	0	0	0	0
0	0	0	0	1
1	1	0	1	1

O/P : 3

Solution:- Similar to previous problem, but we need to identify the shape of islands. To find shape we can find relative order i.e subtract i,j from the starting node x,y. And store it in a sorted order. We can create a set to store the shapes to avoid duplicacy. At the end we will have unique shapes. i.e island.

C++ code

```
void BFS(vector<vector<int>>& grid, vector<vector<bool>>
    &visited, int sr, int sc, int n, int m,
    vector<pair<int,int>>& shape) {
queue<pair<int,int>> q;
q.push({sr,sc});
visited[sr][sc] = true;
while(!q.empty()) {
int x = q.front().first, y = q.front().second;
int directions[4][2] = {{1,0},{0,1},{-1,0},{0,-1}};
q.pop();
shape.push({x-sr, y-sc});
for(int i=0; i<4; i++) {
int newx = x+directions[i][0], newy = y+directions[i][1];
if(newx<0 || newx>=n || newy<0 || newy>=m || visited[newx]
    [newy]
    || grid[newx][newy] != 1) continue;
else {
    visited[newx][newy] = true;
    q.push({newx,newy});
}}}
```

```
int countDistinctIslands(vector<vector<int>>& grid) {
int n= grid.size(), m= grid[0].size(), ans=0;
set<vector<pair<int,int>>> uniqueshape;
vector<vector<bool>> visited(n, vector<bool>(m,false));
for(int i=0; i<n;i++){
    for(int j=0; j<m;j++){
        if(!visited[i][j] && grid[i][j]==1){
            vector<pair<int,int>> shape;
            BFS(grid, visited, i,j, n,m, shape);
            sort(shape.begin(), shape.end());
            uniqueshape.insert(shape);
        }
    }
}
return uniqueshape.size();
}
```

Course Schedue II

There are a total of numcourses you have to take, labeled from 0 to numcourses-1, you are given an array prerequisites where prerequisites[i] = [ai, bi] indicates that we must take course bi first if you want to take course a.

b ⟶ a

Return the ordering of courses you should take to finish all course If there are many valid answers, return any of them. If it is impossible to finish all courses, return an emtty array.

Solution2 Same like course-I, just create adjacency list & return the ans array.

```
vector<int> findOrder(int numCourses, vector<vector<int>>
                                    &prerequisites){
int V= numCourses;
vector<int> indegree(V,0), ans, visited(V,0);
queue<int> q;
vector<vector<int>> adj(numCourses);
for(const auto &preq : prerequisites) {
    adj[preq[1]].push_back(preq[0]);
}
for(int i=0; i<adj.size(); i++) {
    for(auto it: adj[i]) indegree[it]++;
}
for(int i=0; i<V; i++) {
    if(indegree[i]==0) q.push(i);
}
while(!q.empty()) {
  int node = q.front();
  q.pop();
  ans.push_back(node);
  for(auto it: adj[node]) {
        indegree[it]--;
            if(indegree[it]==0) q.push(it)
    }
}
if(ans.size()==V) return ans;
vector<int> list;
return list;
}
```

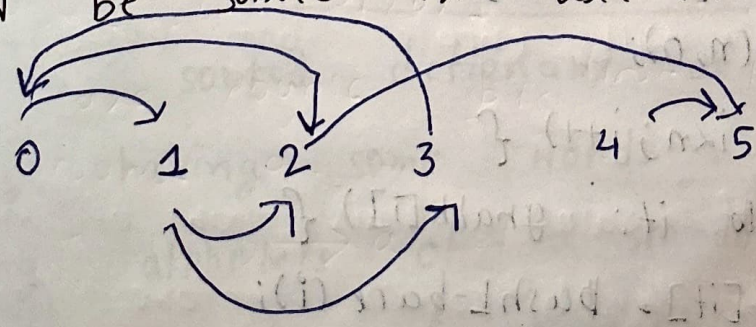
Find eventual safe states

There is a directed graph of n nodes with each node labeled from 0 to n-1. The graph is represented by a 0-indexed 2D integer array graph where graph[i] is an integer array of nodes adjacent to node i, meaning there is an edge from node i to each node in graph[i].

A node is a terminal node if there are no outgoing edges. A node is a safe node if every possible path starting from that node leads to a terminal node (or another safe node).

Return an array containing all the safe nodes of the graph. The answer should be sorted in ascending order.

I/P:



O/P: 2, 4, 5, 6

5, 6 are terminal nodes, Every path from 2, 4, 5, 6 leads to 5 or 6.

Sol:- The idea is to reach terminal node And terminal node has outdegree 0.

So, if we can reverse the graph & start from these nodes, a simple topological sort will give us the answer.

We need topological sort, because every possible path from the safe nodes should end at terminal, which means it's a dependency. With toposort when Indegree == 0, that means no edges from there, which will ensure the ~~for~~ safe nodes.

C++ code

```
vector<int> eventualSafeNodes(vector<vector<int>>& graph) {
int n= graph.size();
vector<vector<int>> adj(n);
vector<int> in(n,0);
for(int i=0; i<n; i++) {
    for(auto it: graph[i]) {
        adj[it].push_back(i);
        in[i]++;
    }
}
```

```
queue<int> q;
vector<int> ans;
for(int i=0; i<n; i++) {
  if(in[i]==0) q.push(i);
}
while (!q.empty()) {
  int node = q.front();
  q.pop();
  ans.push_back(node);
  for(auto it: adj[node]) {
    in[it]--;
    if(in[it]==0) q.push(it);
  }
}
sort(ans.begin(), ans.end());
return ans;
}
```

Alien Dictionary

Given a sorted dictionary of an alien language having some words dict & K starting alphabets of a standard dictionary. Find the order of characters in the alien language. If no valid ordering of letters is possible, then return an empty string.

Note: Many orders may be possible for a particular test case, thus you may return any valid order. Output will be "true" if the order of string returned by the function is correct else "false" denotes incorrect string returned.

Examples

I/p: dict[] = ["baa", "abcd", "abca", "cab", "cad"]
K = 4.

O/p: true, (b → d → a → c)

b a a] ⇒ b > a
a b c d] ⇒ d > a
a b c a] ⇒ a > c
e a b] ⇒ b > d
c a d]

b → a
b → a | d → a
b → a → c | d → a → c
b → d → a → c

Solution: Here, we shall try to create an ordering of dependency. We shall visit every pair of order & check first mismatched character & will create an edge b/w them.

during this, if same ~~words~~ chars of words with higher length comes before lesser one, we return false,

a b c d
a b c ⇒ abc should have abbcard early

Then when the graph formed, we shall try to find cycle, because if we say $e \to a$ & again find $\to$ $a \to c$, this will mean invalid dictionary.

After that we shall return the topo sort as answer.

c++ code

```
bool isCyclic(int V, vector<vector<int>> &adj, vector<int> &ans) {
vector<int> indegree(V, 0);
queue<int> q;
vector<int> visited(V, 0);
for(int i=0; i<V; i++) {
    for(auto it: adj[i]) indegree[it]++;
}
for(int i=0; i<V; i++) {
  if(indegree[i]==0) q.push(i);
}
while(!q.empty()) {
 int node = q.front();
 q.pop();
 ans.push_back(node);
 for(auto it: adj[node]) {
     indegree[it]--;
 if(indegree[it]==0) q.push(it);
 }
}
return ans.size()==V? false: true;
}
```

```
string findOrder(vector<string> dict, int k) {
vector<int> ans;
int n = dict.size();
bool isPossible = false;
vector<vector<int>> adj(k);
for(int i=0; i<n-1; i++) {
string s1 = dict[i], s2 = dict[i+1];
int len = min(s1.size(), s2.size());
bool edgeAdded = false;
for(int j=0; j<len; j++) {
    if(s1[j] != s2[j]) {
    adj[s1[j] - 'a'].push_back(s2[j] - 'a');
    edgeAdded = true;
    break;
    }
}
if(!edgeAdded && s1.size() > s2.size()) return "";
}
isPossible = isCyclic(k, adj, ans);
if(isPossible) return "";
string res = "";
for(int i=0; i<ans.size(); i++) {
    res += 'a' + ans[i];
}
return res;
}
```

Shortest Path / Dijsakstra

Shortest Path in Binary ~~Matrix~~ Maze/Matrix

Given an $n \times n$ binary matrix grid, return the length of the shortest clear path in the matrix. If there is no clear path, return -1.

A clear path in a binary matrix is a path from top-left cell (i.e (0,0)) to the bottom-right cell (i.e. $(n-1, n-1)$) such that

(a) All visitable cells of the path are 0.

(b) All the adjacent cells of the path are 8-directionally connected.

The length of a clear ~~path~~ path is the number of visited cells of this path.

Ex:
I/P:

0 →	0	0
1	1	0
1	1	0

O/P: 3

(ii)

0	1
1	0

O/P 2

Solution: So, it is a simple source (0,0) to destination $(n-1, n-1)$ shortest path. We need to keep following ~~this~~ things

(i) If grid[i][j] == 1, then we need to skip.

(ii) Need to check boundaries.

(iii) The distance is 1 from 1 cell to other, & at each valid path we just do +1 to distance, so we can apply dijkstra but without priority queue.

c++ code

```
int shortestPathBinaryMatrix(vector<vector<int>>& grid) {
    int n = grid.size();
    vector<vector<int>> distance(n, vector<int>(n, INT_MAX));
    queue<pair<int,int>> q;
    q.push({0,0});
    distance[0][0] = 1;
    if(grid[0][0] == 1) return -1;
    int directions[8][2] = {{0,1},{1,0},{1,1},{-1,0},{0,-1},{-1,-1},{-1,1},{1,-1}};
    while(!q.empty()) {
        int x = q.front().first, y = q.front().second;
        int d = distance[x][y];
        q.pop();
        for(int i=0; i<8; i++) {
            int newx = x + directions[i][0], newy
            int newY = y + directions[i][1];
            if(newx>=n || newx<0 || newy>=n || newy<0 ||
               grid[newx][newy] == 1) continue;
            if (d+1 < distance[newx][newy]) {
                q.push({newx, newy});
                distance[newx][newy] = d+1;
            }
        }
    }
    return distance[n-1][n-1] == INT_MAX ? -1:
                distance[n-1][n-1];
}
```

Path with Minimum Effort

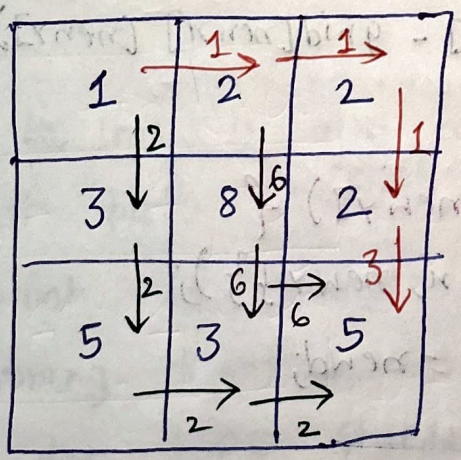
You are a hiker preparing for an upcoming hike. You are given heights, a 2D array of size rows x columns, where heights [row] [col] represents the height of cell (row, col). You are situated in the top-left cell, (0,0). & you have to travel to the bottom-right cell (rows-1, colums-1) (i.e. 0 indexed). You can move up, down, left, right, & you wish to find a route that requires the minimum effort.

A route's effort is the maximum absolute difference in heights b/w 2 consecutive cells of the route.

Return the minimum effort required to travel from the top-left cell to the bottom-right cell.

Ex-1

I/P



O/P 2

Solution: It's also source → dest shortest path problem i.e dijkastra, But instead of adding the distance, just a track maximum distance & path & update ~~update~~ on the current distance array accordingly.

C++ code

```
int minimumEffortPath(vector<vector<int>>& grid) {
int n = grid.size(), m = grid[0].size();
vector<vector<int>> distance(n, vector<int>(m, INT_MAX));
priority_queue<pair<int, pair<int, int>>,
  vector<pair<int, pair<int, int>>>,
  greater<>> pq;
pq.push({0, {0, 0}});
distance[0][0] = 0;
int directions[4][2] = {{0,1}, {1,0}, {0,-1}, {-1,0}};
while(!pq.empty()) {
int x = pq.top().second.first, y = pq.top().second.second;
int d = distance[x][y];
pq.pop();
for(int i=0; i<4; i++) {
int newx = x + directions[i][0]; newY = y + directions[i][1];
if(newx>=n || newx<0 || newY>=m || newY<0) continue;
int newd = abs(grid[x][y] - grid[newx][newY]);
newd = max(d, newd);
if(newd < distance[newx][newY]) {
    pq.push({newd, {newx, newY}});
    distance[newx][newx] = newd;
  }
}
}
return distance[n-1][m-1];
}
```

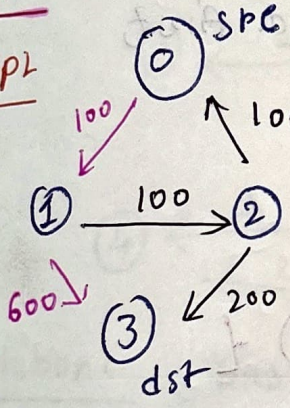
Cheapest Flights Within K Stops

There are n cities connected by some number of flights. You are given an array flights where flights[i] = [from_i, to_i, price_i], indicats that there is a flight from city from_i to city to_i with cost price_i.

You are also given 3 integers src, dest, & k return the cheapest price from src to dst with at most k stops. If there are no such route, return -1.

Ex:

I/P:



K = 1

O/P: 700

Soln: This is also a source to destination shortest path problem, but here main constraint is number of stops. So, instead of storing distance on priority queue, we can store the stops. Starting from src, at each stop we can add +1 to stops & store it in queue, since stops will be increasing by 1 only we don't need priority queue, we can use simple queue, & when stop > K, we can skip. In queue we shall store.

{ stops, city, price }

C++ code

```
int findCheapestPrice(int n, vector<vector<int>>& flights,
                      int src, int dst, int k) {
    vector<vector<pair<int,int>>> adj(n);
    vector<int> prices(n, INT_MAX);
    queue<pair<int, pair<int,int>>> q;
    q.push({0, {src, 0}});
    prices[src] = 0;
    for(int i=0; i<flights.size(); i++)
        adj[flights[i][0]].push_back({flights[i][1],
                                      flights[i][2]});
    while(!q.empty()) {
        auto it = q.front();
        q.pop();
        int stops = it.first, node = it.second.first,
        cost = it.second.second;
        if(stops > k)  continue;
        for(auto it: adj[node]) {
            if(prices[it.first] > cost + it.second) {
                prices[it.first] = cost + it.second;
                q.push({stops+1, {it.first, prices[it.first]}});
            }
        }
    }
    return prices[dst] == INT_MAX ? -1 : prices[dst];
}
```

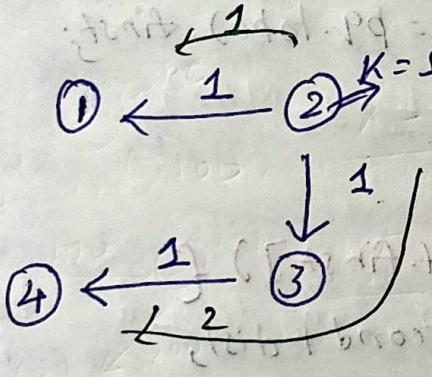
Network Delay Time

You are given a network of n nodes, labeled from 1 to n. You are also given times, a list of travel times as directed edges times[i] = (ui, vi, wi),

ui = source, vi = destination, wi = time for a signal to travel from ui → vi;

We will send a signal from a given node k. Return the minimum time it takes for all the n nodes to recieve the signal. If its impossible for all the n nodes to recieve the signal, return -1.

Ex:



O/P - 2

Solution: This is also a source to destination shortest path problem. But we need to find shortest path (time) from source to all nodes. And, among all the distances need to take maximum, because this is the time by which all node will recieve the signal. But if any distance is ∞, then it's unreachable, then return -1.

C++ code

```
int networkDelayTime(vector<vector<int>>& times, int k,
                     int n) {
vector<vector<pair<int,int>>> adj(n);
for(int i=0; i<n; i++) {
adj[times[i][0]-1].push_back
({times[i][1]-1, times[i][2]});
priority_queue<pair<int,int>, vector<pair<int,int>>,
               greater<>> pq;
int v=n;
pq.push({0,k-1});
vector<int> distance(v,INT_MAX);
distance[k-1]=0;
while(!pq.empty()) {
  int node=pq.top().second, dist=pq.top().first;
  pq.pop();
  for(auto it: adj[node]) {
    if(dist+it.second<distance[it.first]) {
      distance[it.first]=it.second+dist;
      pq.push({distance[it.first], it.first});
    }
  }
}
int result=-1;
for(auto it: distance)
  result=max(it, result);
return result==INT_MAX ? -1: result;
}
```

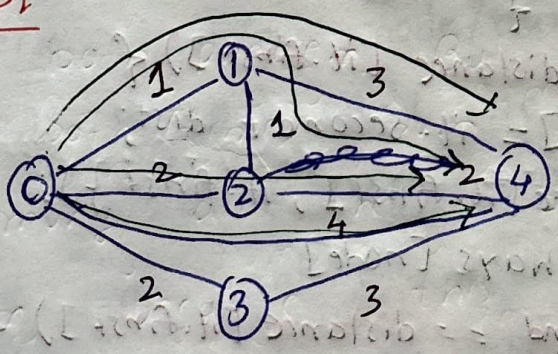
Number of ways to Arrive at Destination

You are in a city that consists of n intersections numbered from 0 to n-1 with bi-directional roads b/w some intersections. The inputs are generated such that you can reach any intersection from any other intersection & that there is at most one road b/w any two intersections.

You are given an integer n and a 2D integer array roads where roads[i] = [ui, vi, timei] means there is a road b/w ui & vi & take timei to travel. You want to know in how many ways one can travel from 0 → n-1 with shortest amount of time.

Return ans mod 10^9+7

Ex1 I/P:



O/P = 4

Solution: This also need Dijakstra as shortest path needed. Here whenever we find a new distance == old shortest distance we know, we got a ~~path~~ another path. So, we shall store a number of path arrays & will increment it.

Whenever we find a new shortest distance we just reset number of path to 1.

C++ code

```
int countPaths(int n, vector<vector<int>>& roads) {
vector<vector<pair<int,int>>> adj(n);
const int MOD = 1e9+7;
for(int i=0; i<roads.size(); i++) {
  adj[roads[i][0]].push_back({roads[i][1], roads[i][2]});
  adj[roads[i][1]].push_back({roads[i][0], roads[i][2]});
}                                    //Roads are bidirectional.
vector<long long> ways(n, 0);
ways[0] = 1;
int src = 0; V = adj.size();
priority_queue<pair<long long, long long>, vector<pair<
  long long, long long>>, greater<>> pq;
pq.push({0, src});
vector<long long> distance (V, LLong_MAX);
distance [src] = 0;
While (!pq.empty()) {
    long long node = pq.top.second, long long dis = pq.top.
    pq.pop();                                    first;
for(auto it: adj[node]) {
  if(dist + it.second < distance [it.first]) {
     distance [it.first] = it.second + dist;
    pq.push ({distance [it.first], it.first});
    ways[it.first] = ways [node]
  } else if {dist+it.second == distance [it.first]) {
    ways [it.first] = (ways [it.first] + ways [node]) % MOD;
  }
}
}
return ways [n-1];
}
```

Minimum multiplications to reach End

Given start, end & an array arr of n numbers. At each step, start is multiplied with any number in the array & then mod operation with 100000 is done to get the new start.

Find minimum steps in which end can be achieved starting from start. If it is not possible to reach end, then return -1.

Ex

I/P: arr [] = {2, 5, 7} O/P: 2
start = 3, end = 30

3 × 2 = 6 × 5 = 30

Solution

We shall also apply dijkstra, since we can multiply all array elements to current node (starting from start node).

And since we need to do mod 100000, that means we have limited numbers to reach i.e. 99999. So, at each step, we can check which node/number we can go to among 99999. And, we can store steps as distance by which we are reaching a certain node. These steps will get increased by 1 only, so we don't need priority queue here, simple queue will work.

At the end check if distance [end] == ∞, which means it

c++ code

```
int minimumMultiplications(vector<int>& arr, int start, int end) {
int v = 99999, n = arr.size(), MOD = 100000;
vector<int> distance(v, INT_MAX);
queue<pair<int, int>> q;
distance[start] = 0;
q.push({0, start});

while (!q.empty()) {
int d = q.front().first, node = q.front().second;
q.pop();
for(int i=0; i<n; i++) {
    int newNode = (node * arr[i]) % MOD;
    if (distance[newNode] > d+1) {
        distance[newNode] = d+1;
        q.push({d+1, newNode});
    }
}
}

return distance[end] == INT_MAX? -1 : distance[end];
}
```

Find the city with the smallest number of Neighbours at a Threshold Distance :-

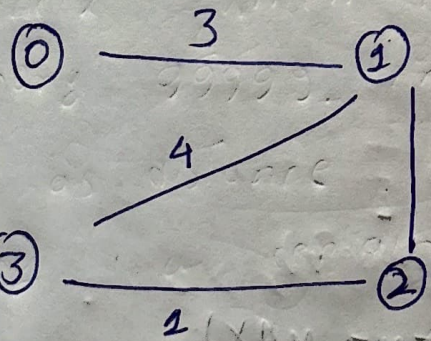
There are n cities numbered from 0 to n-1.
Given the array edges where edges[i] = [from_i, to_i, weight_i] represents a bidirectional & weighted edge between cities from_i and to_i & given integer distanceThreshold.

Return the city with the smallest number of cities that are reachable through some path & whose distance is at most distanceThreshold, If there are multiple such cities, return the city with the greatest number.

Notice that the distance of a path connecting cities i & j is equal to the sum of the edges weights along that path.

Example

I/P



distanceThreshold = 4.

O/P: 3

0 → 1, 2
1 → 0, 2, 3
2 → 0, 1, 3
3 → 1, 2

→ Return 3 as 3 > 0.

Solution:- simple Dijkstra, just calculate source to shortest path to all other cities. ~~At the end take only those cities's~~ After this take a count array, store how many cities we can reach from a certain node/city i.

At the end take max i from count array where count[i] is max.

c++ code

```
int findTheCity(int n, vector<vector<int>>& edges,
                int distanceThreshold) {
vector<vector<pair<int, int>>> adj(n);
for (int i=0; i<edges.size(); i++) {
  adj[edges[i][0]].push_back({edges[i][1], edges[i][2]});
  adj[edges[i][1]].push_back({edges[i][0], edges[i][2]});
}
priority_queue<pair<int,int>, vector<pair<int, int>>,
               greater<>> pq;

vector<int> count(n, 0);
int v=n;
for (int i=0; i<n; i++) {
  pq.push({0, 1});
  vector<int> distance(v, INT_MAX);
  distance[i]=0;
while (!pq.empty()) {
  int node = pq.top().second;
  int dist = pq.top().first;
  pq.pop();
```

```
for (auto it: adj[v]) {
    if (dist + it.second < distance[it.first]) {
        distance[it.first] = it.second + dist;
        pq.push({distance[it.first], it.first});
    }
  }
}
for (int j=0; j<n; j++) {
    if (i!=j && distance[j] <= distanceThreshold)
        count[i]++;
  }
}
int numberOfCities = INT_MAX;
int res;
for(int i=0; i<n; i++) {
    if (count[i] <= numberOfCities) {
        numberOfCities = count[i];
        res = i;
    }
}
return res;
}
```

Prim's Algo

C++

```
int spanningTree(int V, vector<vector<int>> adj[]){
priority_queue<pair<int,int>, vector<pair<int,int>>,
                greater<>> pq;
vector<int> visited(V,0);
pq.push({0,0});
int ans=0;
while(!pq.empty()) {
int weight = pq.top().first;
int node = pq.top().second;
pq.pop();
if(visited[node]==1) continue;
visited[node]=1;
ans += weight;
for(auto it: adj[node]) {
  int newNode = it[0];
  int weight = it[1];
  if(visited[newNode]==0) {
      pq.push({weight,newNode});
  }
}
}
return ans;
}
```

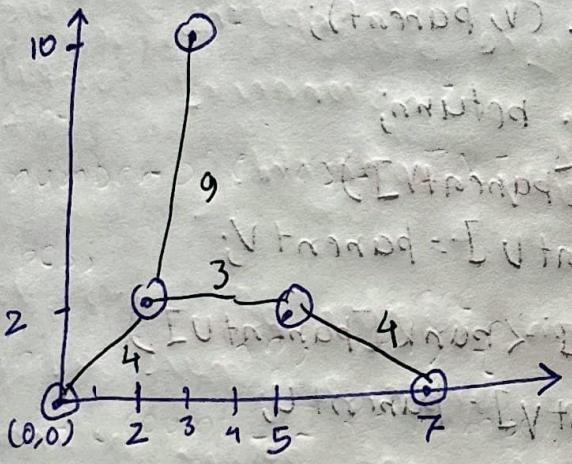
Min cost to connect All points

You are given an array points representing integer coordinates of some points on a 2D plane, where points[i] = [xi, yi]

The cost of connecting 2 points [xi, yi] & [xj, yj] = |xi - xj| + |yi - yj|. Return min cost to make all points connected (exactly one simple path b/w any 2 points).

Ex

I/P



O/P = 20

Sol: Basically, this finding cost of MST, & edge cost for any x & y is |xi - xj| + |yi - yj|. We just need to create graph & apply prim.

C++ code

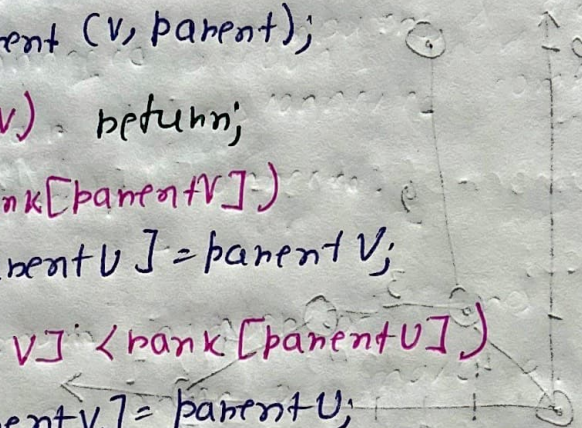
```
int n = points.size(); v = n;
vector<vector<pair<int,int>>> adj(v);
for(int i=0; i<n; i++) {
  for(int j=0; j<n; j++) {
  if(i==j) continue;
int cost = abs(points[i][0] - points[j][0]) +
           abs(points[i][1] - points[j][1]);
  adj[i].push_back({j, cost});
  ... apply Prims.
```

Khushkal Algo C++

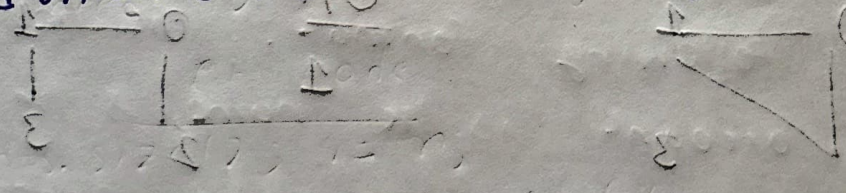
```
int findParent(int node, vector<int>& parent) {
    if (node == parent[node])
        return node;
    return parent[node] = findParent(parent[node], parent);
}

void findUnion(int u, int v, vector<int>& parent,
               vector<int>& rank) {
    int parentU = findParent(u, parent);
    int parentV = findParent(v, parent);
    if (parentU == parentV) return;
    if (rank[parentU] < rank[parentV])
        parent[parentU] = parentV;
    else if (rank[parentV] < rank[parentU])
        parent[parentV] = parentU;
    else {
        parent[parentU] = parentV;
        rank[parentU]++;
    }
}

bool comp(vector<int> a, vector<int> b) {
    return a[2] < b[2];
}
```



```
int spanningTree (int V, vector<vector<int>>adj[]){
vector<int> rank (V, 0);
vector<int> parents (V);
for(int i=0; i<V; i++) parents[i] = i;
vector<pair<int, pair<int,int>>> edges;
for(int i=0; i<V; i++) {
    for(auto it: adj[i])
        edges.push_back({it[1],{i, it[0]}});
}
sort(edges.begin(), edges.end());
int ans =0;
for(auto it: edges) {
int weight = it.first, u = it.second.first;
int v = it.second.second;
if(findParent(u, parents) != findParent(v, Parents)){
    ans += weight;
    findUnion(u,v, parents, rank);
}
}
return ans;
}
```



Number of Operations to Make Network connected

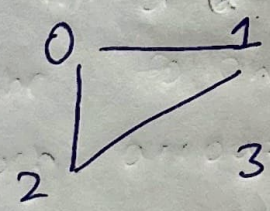
There are n computers numbered from 0 to n-1 connected by ethernet cables connections forming a network where connections[i] = [a_i, b_i] represents a connection b/w computers a_i & b_i. Any computer can reach any other computer directly or indirectly through the network.

You are given an initial computer network connections. You can extract certain cables b/w 2 directly connected computers, & place them b/w any pair of disconnected computers to make them directly connected.

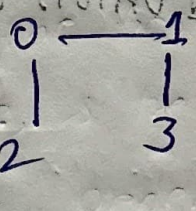
Return the minimum number of times you need to do this in order to make all the computers connected. If not possible, return -1

Ex

Ex I/P



O/P



Solution:

We can delete one edge & use it to connect a disconnected node. So, if we can somehow count extra edges, without which it doesnot create a disconnected node, we can use those to connect, disconnected components. So,

extra edges > number of components.

Extraedge → While doing findUnion, if we found (parent U == parent V) means, U, V belong to same component, so u → v is an extraedge.

component → Whenever parent[i] == i; means this is a new ultimate component.

c++ code

```
void findUnion( ____ ____ ____ ____ ____ ,int &extraEdge)
{
 if(parent U == parent V) extraedge ++; return;
}
int makeConnected (int n, vector<vector<int>>& connections) {
 vector<int> rank (n,0);
 vector<int> parent (n,0);
 int extraEdge = 0;
for(int i=0; i<n; i++) parent[i] = i;
for(int i=0; i<connections.size(); i++) {
 findUnion (connections[i][0], connections[i][1],
                parent, rank, extraedge);
}
int components =0;
for(int i=0; i<n; i++) {
    if (parents[i] == i) components ++;
}
if( components - 1 > extraedge) return -1;
return components -1;
}
```