Problem on BFS/DFS

Q.5 <https://www.naukri.com/code360/problems/cycle-detection-in-undirected-graph_1062670?leftPanelTabValue=PROBLEM>

* There are 2 approaches for this problem
* 1.BFS
* 2.DFS

1.BFS

So what the basic idea is here we need to carry parent at start we can carry -1 because for starting node there is no parent (parent means last node where it come from)

* If any of call returning true then return true and also if some node is already visited but it’s not’s parent then return true because it has cycle
* Or else return false
* At start in main function create 2d vector for creating adjaceny list and visited array iterate to each vertex check if it not visited call BFS and it return true then return “Yes” for cycle
* At last return “No”

bool bfs(vector<vector<int>> &graph,int visited[],int node,int parent)

{

    queue<pair<int,int>> q;

    q.push({node,parent});

    visited[node]=1;

    while(!q.empty())

    {

        int node=q.front().first;

        int parent=q.front().second;

        q.pop();

        for(auto it: graph[node])

        {

            if(!visited[it])

            {

                q.push({it,node});

                visited[it]=1;

            }

            else if(parent!=it)

            {

                return true;

            }

        }

    }

    return false;

}

string cycleDetection (vector<vector<int>>& edges, int n, int m)

{

    // Write your code here.

    int visited[n+1]={0};

    vector<vector<int>> graph(n + 1, vector<int>());

    for (int i = 0; i < m; i++)

    {

        graph[edges[i][1]].push\_back(edges[i][0]);

        graph[edges[i][0]].push\_back(edges[i][1]);

    }

    for(int i=1;i<=n;i++)

    {

        if(!visited[i])

        {

            // if(dfs(graph,visited,i,-1)==true) return "Yes";

            if(bfs(graph,visited,i,-1)==true) return "Yes";

        }

    }

    return "No";

}

TC: O(N+2E) because for every node we are visiting it’s all adjacent node so there are N node and 2E adjacent node in graph

SC: O(N) recursive stack space +O(N2) for adjacency list

2. DFS :

So what the basic idea is here we need to carry parent at start we can carry -1 because for starting node there is no parent (parent means last node where it come from)

* If any of call returning true then return true and also if some node is already visited but it’s not’s parent then return true because it has cycle
* Or else return false
* At start in main function create 2d vector for creating adjaceny list and visited array iterate to each vertex check if it not visited call dfs and it return true then return “Yes” for cycle
* At last return “No”

bool dfs(vector<vector<int>>& adj,int visited[],int node,int parent)

{

    visited[node]=1;

    for(auto it:adj[node])

    {

        if(!visited[it])

        {

            if(dfs(adj, visited, it, node)==true) return true;

        }

        else if(it!=parent) return true;

    }

    return false;

}

string cycleDetection (vector<vector<int>>& edges, int n, int m)

{

    // Write your code here.

    int visited[n+1]={0};

    vector<vector<int>> graph(n + 1, vector<int>());

    for (int i = 0; i < m; i++)

    {

        graph[edges[i][1]].push\_back(edges[i][0]);

        graph[edges[i][0]].push\_back(edges[i][1]);

    }

    for(int i=1;i<=n;i++)

    {

        if(!visited[i])

        {

            if(dfs(graph,visited,i,-1)==true) return "Yes";

        }

    }

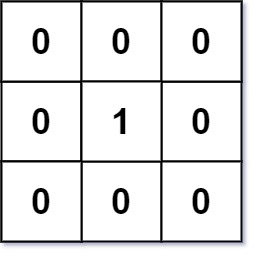
    return "No";

}

TC: O(N+2E) because for every node we are visiting it’s all adjacent node so there are N node and 2E adjacent node in graph

SC: O(N) recursive stack space +O(N2) for adjacency list

Q.7 <https://leetcode.com/problems/01-matrix/>

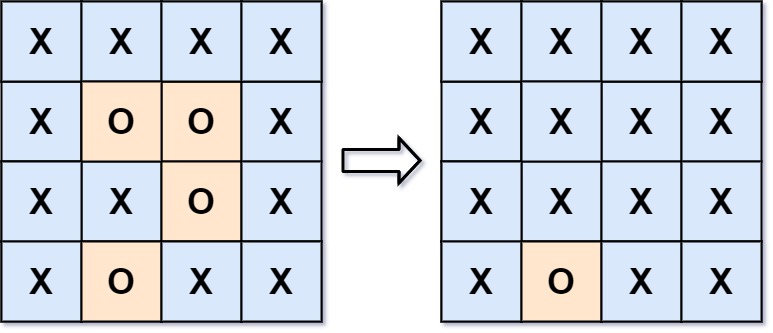
* 
* So here what we have to do calculate the nearest 0th distance from current cell
* And it cell is itself zero then distance is also 0
* So here we are going to use BFS and the logic behind it is first we are going to store zero 4th direction distance as 1 it it is not visited so before this we need to store index of all zero and along with it’s distance as zero also mark them as visited
* Every time when we are going to store distance of each cell what we have got from previous cell +1

|  |
| --- |
| class Solution {  public:      vector<vector<int>> updateMatrix(vector<vector<int>>& mat) {           int n=mat.size();         int m=mat[0].size();         vector<vector<int>> visited(n,vector<int>(m,0));         vector<vector<int>> dist(n,vector<int>(m,0));         queue<pair<pair<int,int>,int>> q;         for(int i=0;i<n;i++)         {              for(int j=0;j<m;j++)              {                  if(mat[i][j]==0)                  {                      q.push({{i,j},0});                      visited[i][j]=1;                  }              }         }         int drow[]={-1,0,1,0};         int dcol[]={0,-1,0,1};         while(!q.empty())         {              int row=q.front().first.first;              int col=q.front().first.second;              int step=q.front().second;              q.pop();                dist[row][col]=step;              for(int i=0;i<4;i++)              {                  int nr=row+drow[i];                  int nc=col+dcol[i];                  if(nr>=0 && nr<n && nc>=0 && nc<m && !visited[nr][nc])                  {                      visited[nr][nc]=1;                      q.push({{nr,nc},step+1});                  }              }         }         return dist;      }  }; |

TC: O(M\*N\*4) SC:O(M\*N)

Q.8 <https://leetcode.com/problems/surrounded-regions/>

* Here we have to replace o with x but there are some constraints first the O which or on boundry that can’t be converted and also the O which we have to convert that is surrounded by x from 4 direction
* So for this we need to do dfs on first row and last row also on first col and last col check if there are O if yes call dfs on it and dfs will mark it’s 4 direction if it encounter O
* After doing this at last check in board which are not visited also they have O then replace it with X



|  |
| --- |
| class Solution {  public:      void dfs(int row,int col,vector<vector<char>> &board,vector<vector<int>> &visited,int drow[],int dcol[])      {          int n=board.size();          int m=board[0].size();          visited[row][col]=1;          for(int i=0;i<4;i++)          {              int nr=row+drow[i];              int nc=col+dcol[i];              if(nr>=0 && nr<n && nc>=0 && nc<m && !visited[nr][nc] && board[nr][nc]=='O')              {                  dfs(nr,nc,board,visited,drow,dcol);              }          }      }      void solve(vector<vector<char>>& board) {          int n=board.size();          int m=board[0].size();          int drow[]={-1,0,1,0};          int dcol[]={0,-1,0,1};            vector<vector<int>> visited(n,vector<int>(m,0));          // for first row and last row          for(int i=0;i<m;i++)          {              if(!visited[0][i] && board[0][i]=='O')              {                  dfs(0,i,board,visited,drow,dcol);              }              if(!visited[n-1][i] && board[n-1][i]=='O')              {                  dfs(n-1,i,board,visited,drow,dcol);              }          }          // for first col and last col          for(int i=0;i<n;i++)          {              if(!visited[i][0] && board[i][0]=='O')              {                  dfs(i,0,board,visited,drow,dcol);              }              if(!visited[i][m-1] && board[i][m-1]=='O')              {                  dfs(i,m-1,board,visited,drow,dcol);              }          }          for(int i=0;i<n;i++)          {              for(int j=0;j<m;j++)              {                  if(!visited[i][j] && board[i][j]=='O')                  {                      board[i][j]='X';                  }              }          }          }  }; |

TC: O(M\*N) SC:O(M\*N)

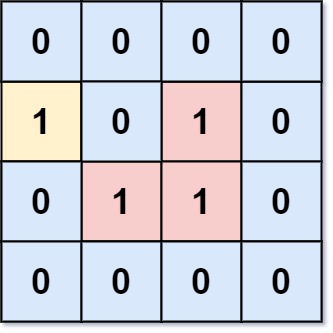
Q.9 <https://leetcode.com/problems/number-of-enclaves/description/>

You are given an m x 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***.

**Example 1:**



* The logic is similar kind of above question if somhow we can reach boundry by using 4 direction then we are not going to count that 1 which represent a land
* So basically we have to return the land from where we can’t reach boundries
* The logic is preety simple I used bfS we also can you dfs
* First of all store all boundries land that 1’s into queue also mark them visted
* After that until queue gets empty pop out check it’s for direction from where we can reach any land inside if yes push that index into queue also mark them visited
* At last iterate through given grid check which land 1’s are remaining also they shouldn’t be visited just count it return it

|  |
| --- |
| class Solution {  public:      int numEnclaves(vector<vector<int>>& grid) {          int n=grid.size();          int m=grid[0].size();          vector<vector<int>> visited(n,vector<int>(m,0));          queue<pair<int,int>> q;          // first row and last          for(int i=0;i<m;i++)          {              if(grid[0][i]==1)              {                  q.push({0,i});                  visited[0][i]=1;              }              if(grid[n-1][i]==1)              {                  q.push({n-1,i});                  visited[n-1][i]=1;              }          }          //first col and last           for(int i=0;i<n;i++)          {              if(grid[i][0]==1)              {                  q.push({i,0});                  visited[i][0]=1;              }              if(grid[i][m-1]==1)              {                  q.push({i,m-1});                  visited[i][m-1]=1;              }          }          int drow[]={-1,0,1,0};          int dcol[]={0,-1,0,1};          while(!q.empty())          {              int r=q.front().first;              int c=q.front().second;              q.pop();              for(int i=0;i<4;i++)              {                  int nr=r+drow[i];                  int nc=c+dcol[i];                  if(nr>=0 && nr<n && nc>=0 && nc<m && grid[nr][nc]==1 && !visited[nr][nc])                  {                      q.push({nr,nc});                      visited[nr][nc]=1;                  }              }          }          int cnt=0;          for(int i=0;i<n;i++)          {              for(int j=0;j<m;j++)              {                  if(!visited[i][j] && grid[i][j]==1) cnt++;              }          }          return cnt;        }  }; |

TC: O(M\*N) SC:O(M\*N)

Q.10 <https://leetcode.com/problems/word-ladder/>

A **transformation sequence** from word beginWord to word endWord using a dictionary wordList is a sequence of words beginWord -> s1 -> s2 -> ... -> sk such that:

* Every adjacent pair of words differs by a single letter.
* Every si for 1 <= i <= k is in wordList. Note that beginWord does not need to be in wordList.
* sk == endWord

Given two words, beginWord and endWord, and a dictionary wordList, return *the****number of words****in the****shortest transformation sequence****from* beginWord *to* endWord*, or*0*if no such sequence exists.*

**Example 1:**

**Input:** beginWord = "hit", endWord = "cog", wordList = ["hot","dot","dog","lot","log","cog"]

**Output:** 5

🡪 from starting word and changing each single character from that word to get next word if that present in wordlist again for second word change character to get such word which is in wordlist and we have to do it until we found endWord it we won’t find endWord then return 0

* so here we are usin BFS we are going level wise in this
* take one set copy all wordlist
* take queue insert beginWord and step 1 in queue
* while queue is not empty
* pop word and step check if that word == endword if yes return step
* iterate the for loop from 0 to word length
* store ith character of word in some variable
* iterate for loop from a to z
* replace ith character with all alphabet and check if that present in set if yest store that word in queue also store step+1 also erase it from set
* after inside for loop replace word[i] with original character
* at last return 0 because we couldn’t able to make sequence

|  |
| --- |
| class Solution {  public:      int ladderLength(string beginWord, string endWord, vector<string>& wordList) {          queue<pair<string,int>> q;          unordered\_set<string> st(wordList.begin(),wordList.end());          q.push({beginWord,1});          while(!q.empty())          {              string word=q.front().first;              int step=q.front().second;              q.pop();              if(word==endWord)              {                  return step;              }              for(int i=0;i<word.size();i++)              {                  char original=word[i];                  for(char ch='a';ch<='z';ch++)                  {                      word[i]=ch;                      if(st.find(word)!=st.end())                      {                          q.push({word,step+1});                          st.erase(word);                      }                  }                  word[i]=original;              }          }          return 0;        }  }; |

TC : O(N\*word length\*26\*logN) SC:O(N) for set

N\*(this many can be in queue )word length \* 26 \* LogN for finding in set

Q.11 <https://www.naukri.com/code360/problems/distinct-islands_630460>

Before doing this question there is one preRequist of this question

Q. <https://leetcode.com/problems/number-of-islands/description/>

🡪 So here just we need to find no of Island so we are using BFS here

* iterate through graph if it has 1 and not visited call bfs and cnt how many time it call bfs that is your ans
* below is code

|  |
| --- |
| class Solution {  public:  void bfs(int row,int col,vector<vector<char>> &grid,vector<vector<int>> &visited)      {          int n=grid.size();          int m=grid[0].size();            visited[row][col]=1;          queue<pair<int,int>> q;          q.push({row,col});          int cnt=0;          int drow[]={-1,0,1,0};          int dcol[]={0,-1,0,1};          while(!q.empty())          {              int r=q.front().first;              int c=q.front().second;              q.pop();                        for(int i=0;i<4;i++)                  {                      int nrow=r+drow[i];                      int ncol=c+dcol[i];                        if(nrow>=0 && nrow<n && ncol>=0 && ncol<m && grid[nrow][ncol]=='1' &&                      !visited[nrow][ncol])                      {                          visited[nrow][ncol]=1;                          q.push({nrow,ncol});                        }                  }                }        }      int numIslands(vector<vector<char>>& grid) {           int n=grid.size();          int m=grid[0].size();          int cnt=0;          int preLen=0;          vector<vector<int>> visited(n,vector<int>(m,0)) ;            for(int i=0;i<n;i++)          {              for(int j=0;j<m;j++)              {                  if(grid[i][j]=='1' && !visited[i][j])                  {                        bfs(i,j,grid,visited);                      cnt++;                    }              }          }            return cnt;        }  }; |

TC:O( N\*M) SC:O(N\*M)

* so now come to actual question we have to find distinct Island
* You are given a two-dimensional array/list of integers consisting of 0s and 1s. In the list, 1 represents land and 0 represents water.
* The task is to find the number of distinct islands where a group of connected 1s(horizontally or vertically) forms an island.
* **Note:**
* Two islands are considered to be the same if and only if one island is equal to another(not rotated or reflected) i.e if we can translate one island on another without rotating or reflecting then it would be considered as the same islands.
* Same logic like above code just for distinct we are using the logic is suppose 1 Island start from 0 0 and 0 1 ,0 2,1 0,1,2 has 1 value that means there is Island
* So here we are substring starting source row and col from each row and col row-row0,col-col0

Where as row0 and col0 are starting row and col of that Island because if some Island are same we’ll get same vector

* Same here we are using BFS logic just we are passing pair typr vector to bfs which will store row-row0,col-col0
* After BFS completion we are going store that vector into set so we can get unqiue Island count
* At last return size of set

|  |
| --- |
| #include<vector>  #include<set>  void dfs(int row,int col,int\*\* arr,vector<vector<int>> &visited,vector<pair<int,int>> &temp,int row0,int col0,int n,int m)  {      visited[row][col]=1;      temp.push\_back({row-row0,col-col0});      int drow[]={-1,0,1,0};      int dcol[]={0,-1,0,1};      for(int i=0;i<4;i++)      {          int r=row+drow[i];          int c=col+dcol[i];          if(r>=0 && r<n && c>=0 && c<m && arr[r][c]==1 && !visited[r][c])          {              dfs(r,  c, arr, visited, temp,  row0,  col0,  n,  m);          }      }  }  int distinctIslands(int\*\* arr, int n, int m)  {      //Write your code here      vector<vector<int>> visited(n,vector<int>(m,0));      set<vector<pair<int,int>>> st;      for(int i=0;i<n;i++)      {          for(int j=0;j<m;j++)          {              if(arr[i][j]==1 && !visited[i][j])              {                  vector<pair<int,int>> temp;                  dfs(i,j,arr,visited,temp,i,j,n,m);                  st.insert(temp);              }          }      }      return st.size();  } |

TC:O( N\*M) SC:O(N\*M)

Q.12 <https://leetcode.com/problems/is-graph-bipartite/description/>

* Bipartite graph means no adjacent have same color
* Here we are using 0 and 1 for coloring
* We take one color array of V vertex and fill it with -1
* Iterate through V vertex check in color array if it has
* Call dfs mark the color
* Check it’s adjacent when passing color just do !col so it will reverse 0 to 1 and 1 to 0
* If it already visited and it has col which we want assign return false from their also if dfs is returning false directly return false at last return true;

|  |
| --- |
| class Solution {  public:      bool dfs(vector<vector<int>> &graph,int color[],int node,int col)      {          color[node]=col;          for(auto it:graph[node])          {              if(color[it]==-1)              {                  if(dfs(graph,color,it,!col)==false) return false;              }              else if(color[it]==col) return false;          }          return true;      }      bool isBipartite(vector<vector<int>>& graph) {          int V=graph.size();          int color[V];          for(int i=0;i<V;i++) color[i]=-1;          for(int i=0;i<V;i++)          {              if(color[i]==-1)              {                  if(dfs(graph,color,i,0)==false) return false;              }          }          return true;        }  }; |

TC:O(V+ 2E) no of vertes and edges

SC:O(V)