**AI 1st BFS & DFS**

***Input***:

import java.io.\*;

import java.util.\*;

// This class represents a

// directed graph using adjacency

// list representation

class Graph {

private int V; // No. of vertices

// Array of lists for

// Adjacency List Representation

private LinkedList<Integer> adj[];

// Constructor

@SuppressWarnings("unchecked") Graph(int v)

{

V = v;

adj = new LinkedList[v];

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

adj[i] = new LinkedList();

}

// Function to add an edge into the graph

void addEdge(int v, int w)

{

adj[v].add(w); // Add w to v's list.

}

// A function used by DFS

void DFSUtil(int v, boolean visited[])

{

// Mark the current node as visited and print it

visited[v] = true;

System.out.print(v + " ");

// Recur for all the vertices adjacent to this

// vertex

Iterator<Integer> i = adj[v].listIterator();

while (i.hasNext()) {

int n = i.next();

if (!visited[n])

DFSUtil(n, visited);

}

}

// The function to do DFS traversal.

// It uses recursive

// DFSUtil()

void DFS(int v)

{

// Mark all the vertices as

// not visited(set as

// false by default in java)

boolean visited[] = new boolean[V];

// Call the recursive helper

// function to print DFS

// traversal

DFSUtil(v, visited);

}

// Driver Code

public static void main(String args[])

{

Graph g = new Graph(4);

g.addEdge(0, 1);

g.addEdge(0, 2);

g.addEdge(1, 2);

g.addEdge(2, 0);

g.addEdge(2, 3);

g.addEdge(3, 3);

System.out.println(

"Following is Depth First Traversal "

+ "(starting from vertex 2)");

g.DFS(2);

}

}

**Output**:

Graph Following is Depth First Traversal (starting from vertex 2)2 0 1 3

***Input*** :

import java.io.\*;

import java.util.\*;

// This class represents a directed graph using adjacency list

// representation

class Graph

{

private int V; // No. of vertices

private LinkedList<Integer> adj[]; //Adjacency Lists

// Constructor

Graph(int v)

{

V = v;

adj = new LinkedList[v];

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

adj[i] = new LinkedList();

}

// Function to add an edge into the graph

void addEdge(int v,int w)

{

adj[v].add(w);

}

// prints BFS traversal from a given source s

void BFS(int s)

{

// Mark all the vertices as not visited(By default

// set as false)

boolean visited[] = new boolean[V];

// Create a queue for BFS

LinkedList<Integer> queue = new LinkedList<Integer>();

// Mark the current node as visited and enqueue it

visited[s]=true;

queue.add(s);

while (queue.size() != 0)

{

// Dequeue a vertex from queue and print it

s = queue.poll();

System.out.print(s+" ");

// Get all adjacent vertices of the dequeued vertex s

// If a adjacent has not been visited, then mark it

// visited and enqueue it

Iterator<Integer> i = adj[s].listIterator();

while (i.hasNext())

{

int n = i.next();

if (!visited[n])

{

visited[n] = true;

queue.add(n);

}

}

}

}

// Driver method to

public static void main(String args[])

{

Graph g = new Graph(4);

g.addEdge(0, 1);

g.addEdge(0, 2);

g.addEdge(1, 2);

g.addEdge(2, 0);

g.addEdge(2, 3);

g.addEdge(3, 3);

System.out.println("Following is Breadth First Traversal "+

"(starting from vertex 2)");

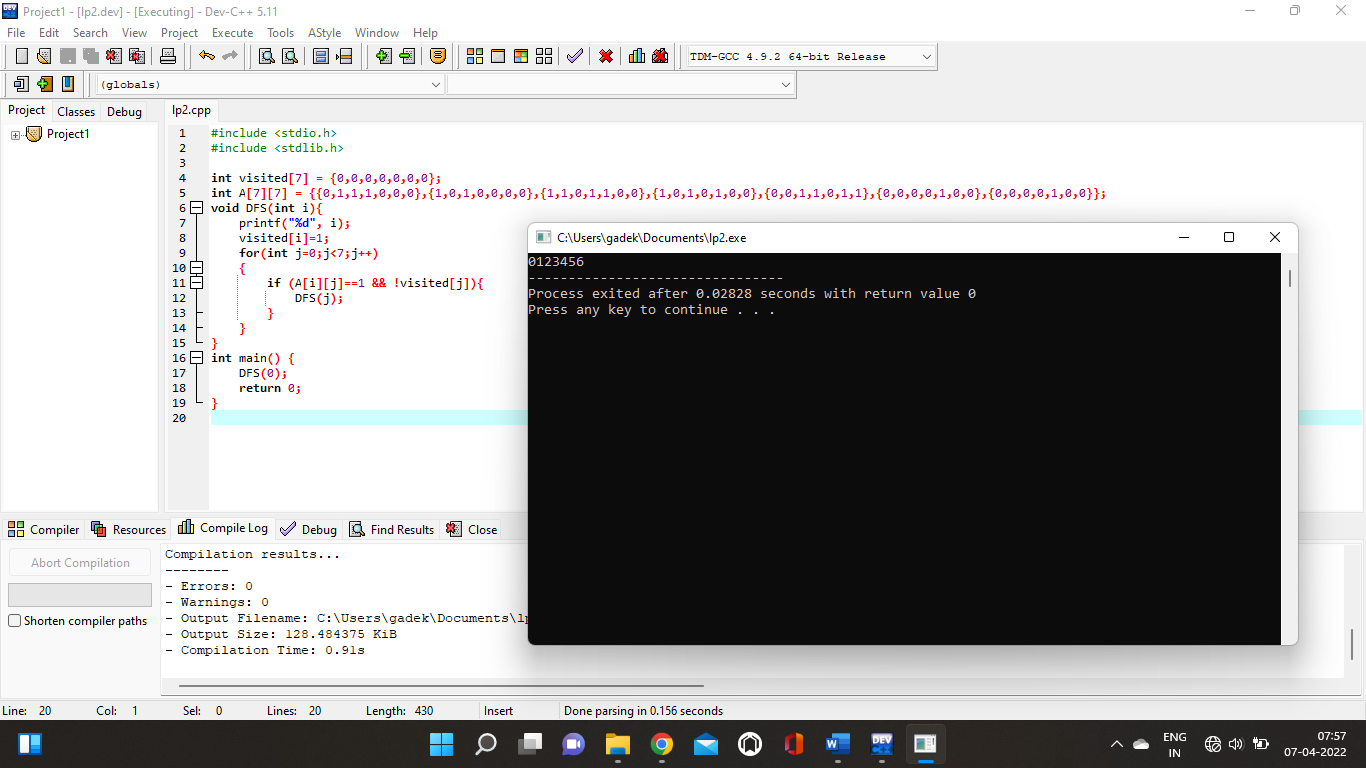
g.BFS(2);

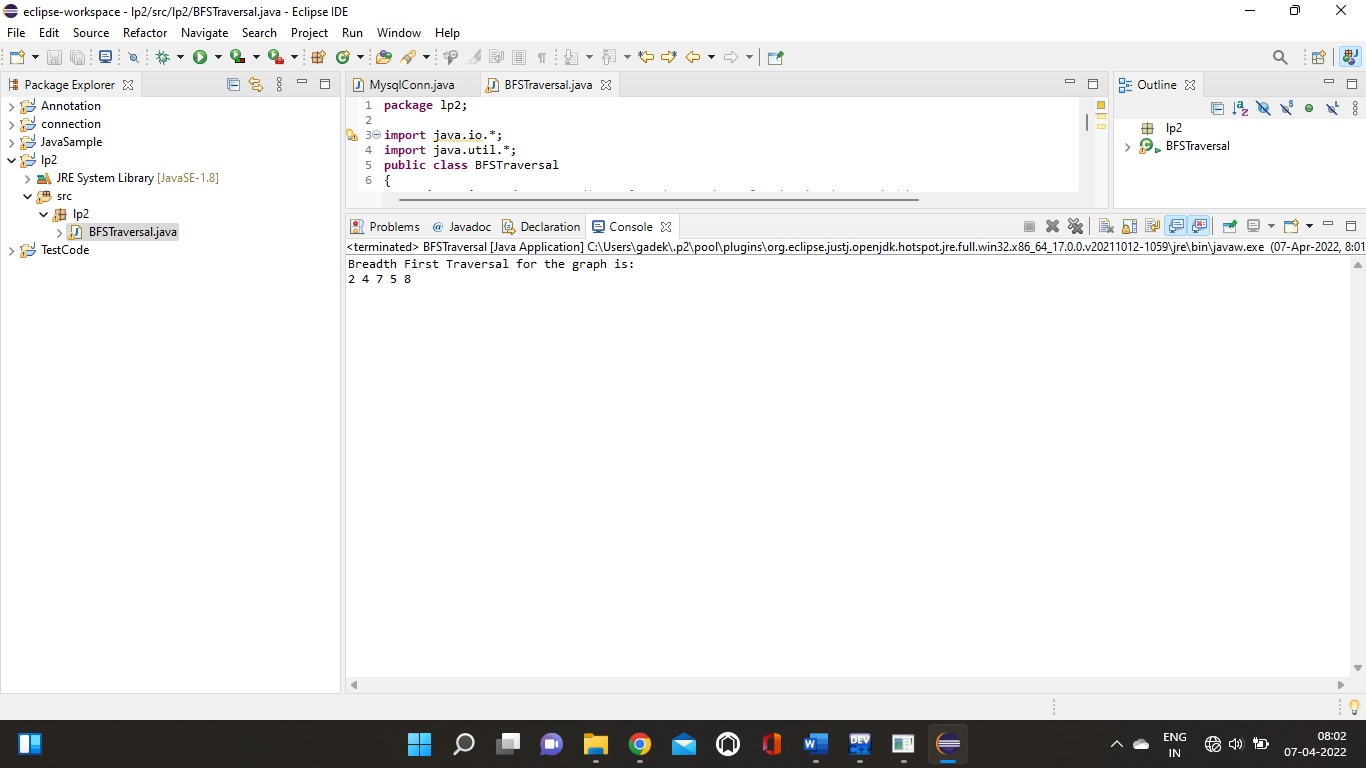
}

}

***Output*** :

Graph Following is Breadth First Traversal (starting from vertex 2)2 0 3 1





**AI 2nd ) A\* Algorithm**

***INPUT***

#include <stdlib.h>

#include <stdio.h>

#include <string.h>

#include <float.h>

#include <iso646.h>

#include <math.h>

#define map\_size\_rows 10

#define map\_size\_cols 10

char map[map\_size\_rows][map\_size\_cols] = {

{1, 1, 1, 1, 1, 1, 1, 1, 1, 1},

{1, 0, 0, 0, 0, 0, 0, 0, 0, 1},

{1, 0, 0, 0, 0, 0, 0, 0, 0, 1},

{1, 0, 0, 0, 0, 1, 1, 1, 0, 1},

{1, 0, 0, 1, 0, 0, 0, 1, 0, 1},

{1, 0, 0, 1, 0, 0, 0, 1, 0, 1},

{1, 0, 0, 1, 1, 1, 1, 1, 0, 1},

{1, 0, 0, 0, 0, 0, 0, 0, 0, 1},

{1, 0, 0, 0, 0, 0, 0, 0, 0, 1},

{1, 1, 1, 1, 1, 1, 1, 1, 1, 1}

};

struct stop {

double col, row;

int \* n;

int n\_len;

double f, g, h;

int from;

};

int ind[map\_size\_rows][map\_size\_cols] = {

{-1, -1, -1, -1, -1, -1, -1, -1, -1, -1},

{-1, -1, -1, -1, -1, -1, -1, -1, -1, -1},

{-1, -1, -1, -1, -1, -1, -1, -1, -1, -1},

{-1, -1, -1, -1, -1, -1, -1, -1, -1, -1},

{-1, -1, -1, -1, -1, -1, -1, -1, -1, -1},

{-1, -1, -1, -1, -1, -1, -1, -1, -1, -1},

{-1, -1, -1, -1, -1, -1, -1, -1, -1, -1},

{-1, -1, -1, -1, -1, -1, -1, -1, -1, -1},

{-1, -1, -1, -1, -1, -1, -1, -1, -1, -1},

{-1, -1, -1, -1, -1, -1, -1, -1, -1, -1}

};

struct route {

int x;

int y;

double d;

};

int main() {

int i, j, k, l, b, found;

int p\_len = 0;

int \* path = NULL;

int c\_len = 0;

int \* closed = NULL;

int o\_len = 1;

int \* open = (int\*)calloc(o\_len, sizeof(int));

double min, tempg;

int s;

int e;

int current;

int s\_len = 0;

struct stop \* stops = NULL;

int r\_len = 0;

struct route \* routes = NULL;

for (i = 1; i < map\_size\_rows - 1; i++) {

for (j = 1; j < map\_size\_cols - 1; j++) {

if (!map[i][j]) {

++s\_len;

stops = (struct stop \*)realloc(stops, s\_len \* sizeof(struct stop));

int t = s\_len - 1;

stops[t].col = j;

stops[t].row = i;

stops[t].from = -1;

stops[t].g = DBL\_MAX;

stops[t].n\_len = 0;

stops[t].n = NULL;

ind[i][j] = t;

}

}

}

s = 0;

e = s\_len - 1;

for (i = 0; i < s\_len; i++) {

stops[i].h = sqrt(pow(stops[e].row - stops[i].row, 2) + pow(stops[e].col - stops[i].col, 2));

}

for (i = 1; i < map\_size\_rows - 1; i++) {

for (j = 1; j < map\_size\_cols - 1; j++) {

if (ind[i][j] >= 0) {

for (k = i - 1; k <= i + 1; k++) {

for (l = j - 1; l <= j + 1; l++) {

if ((k == i) and (l == j)) {

continue;

}

if (ind[k][l] >= 0) {

++r\_len;

routes = (struct route \*)realloc(routes, r\_len \* sizeof(struct route));

int t = r\_len - 1;

routes[t].x = ind[i][j];

routes[t].y = ind[k][l];

routes[t].d = sqrt(pow(stops[routes[t].y].row - stops[routes[t].x].row, 2) + pow(stops[routes[t].y].col - stops[routes[t].x].col, 2));

++stops[routes[t].x].n\_len;

stops[routes[t].x].n = (int\*)realloc(stops[routes[t].x].n, stops[routes[t].x].n\_len \* sizeof(int));

stops[routes[t].x].n[stops[routes[t].x].n\_len - 1] = t;

}

}

}

}

}

}

open[0] = s;

stops[s].g = 0;

stops[s].f = stops[s].g + stops[s].h;

found = 0;

while (o\_len and not found) {

min = DBL\_MAX;

for (i = 0; i < o\_len; i++) {

if (stops[open[i]].f < min) {

current = open[i];

min = stops[open[i]].f;

}

}

if (current == e) {

found = 1;

++p\_len;

path = (int\*)realloc(path, p\_len \* sizeof(int));

path[p\_len - 1] = current;

while (stops[current].from >= 0) {

current = stops[current].from;

++p\_len;

path = (int\*)realloc(path, p\_len \* sizeof(int));

path[p\_len - 1] = current;

}

}

for (i = 0; i < o\_len; i++) {

if (open[i] == current) {

if (i not\_eq (o\_len - 1)) {

for (j = i; j < (o\_len - 1); j++) {

open[j] = open[j + 1];

}

}

--o\_len;

open = (int\*)realloc(open, o\_len \* sizeof(int));

break;

}

}

++c\_len;

closed = (int\*)realloc(closed, c\_len \* sizeof(int));

closed[c\_len - 1] = current;

for (i = 0; i < stops[current].n\_len; i++) {

b = 0;

for (j = 0; j < c\_len; j++) {

if (routes[stops[current].n[i]].y == closed[j]) {

b = 1;

}

}

if (b) {

continue;

}

tempg = stops[current].g + routes[stops[current].n[i]].d;

b = 1;

if (o\_len > 0) {

for (j = 0; j < o\_len; j++) {

if (routes[stops[current].n[i]].y == open[j]) {

b = 0;

}

}

}

if (b or (tempg < stops[routes[stops[current].n[i]].y].g)) {

stops[routes[stops[current].n[i]].y].from = current;

stops[routes[stops[current].n[i]].y].g = tempg;

stops[routes[stops[current].n[i]].y].f = stops[routes[stops[current].n[i]].y].g + stops[routes[stops[current].n[i]].y].h;

if (b) {

++o\_len;

open = (int\*)realloc(open, o\_len \* sizeof(int));

open[o\_len - 1] = routes[stops[current].n[i]].y;

}

}

}

}

for (i = 0; i < map\_size\_rows; i++) {

for (j = 0; j < map\_size\_cols; j++) {

if (map[i][j]) {

putchar(0xdb);

} else {

b = 0;

for (k = 0; k < p\_len; k++) {

if (ind[i][j] == path[k]) {

++b;

}

}

if (b) {

putchar('x');

} else {

putchar('.');

}

}

}

putchar('\n');

}

if (not found) {

puts("IMPOSSIBLE");

} else {

printf("path cost is %d:\n", p\_len);

for (i = p\_len - 1; i >= 0; i--) {

printf("(%1.0f, %1.0f)\n", stops[path[i]].col, stops[path[i]].row);

}

}

for (i = 0; i < s\_len; ++i) {

free(stops[i].n);

}

free(stops);

free(routes);

free(path);

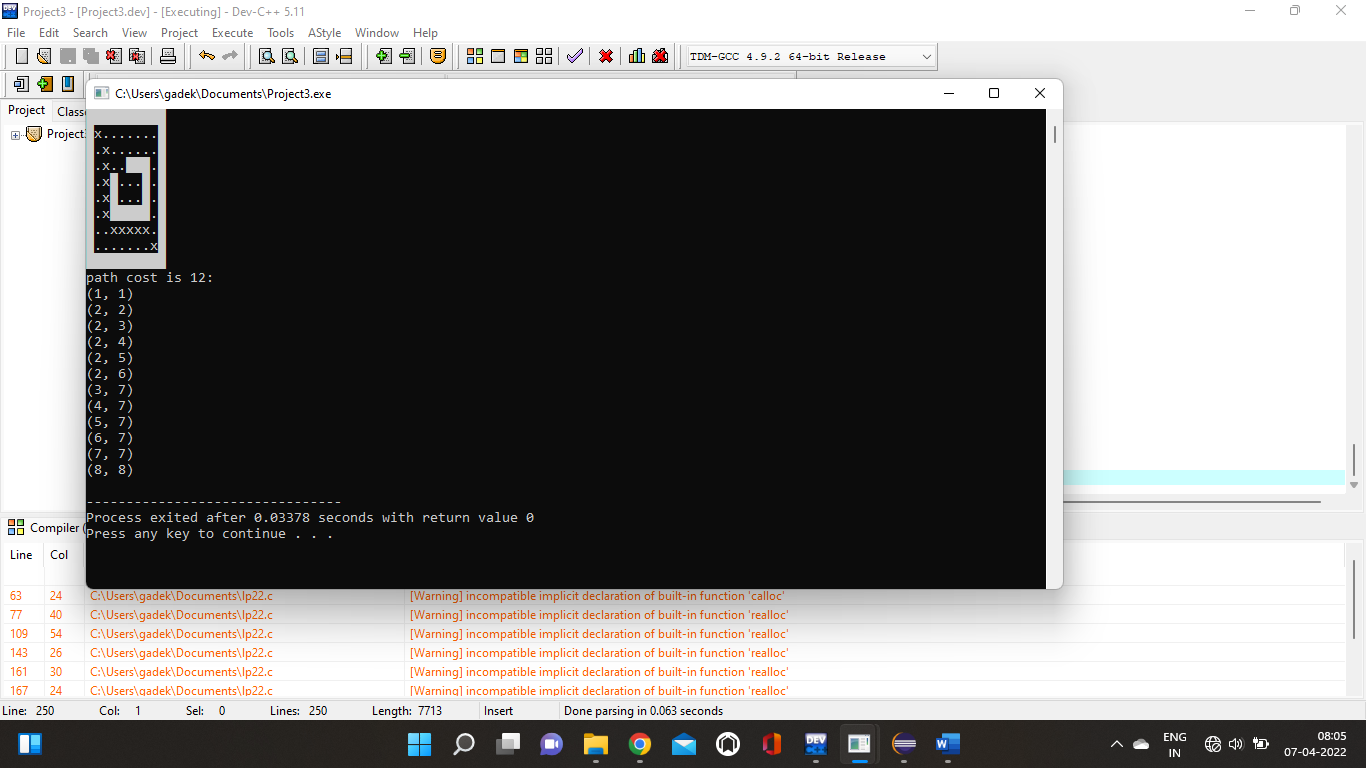
free(open);

free(closed);

return 0;

}

***Output***:

path cost is 12:

(1, 1)

(2, 2)

(2, 3)

(2, 4)

(2, 5)

(2, 6)

(3, 7)

(4, 7)

(5, 7)

(6, 7)

(7, 7)

(8, 8)

**Al 3rd : 3D search algorithm with Kruskel algorithm**

***Input*** :

class Graph:

def \_\_init\_\_(self, vertices):

self.V = vertices

self.graph = []

def add\_edge(self, u, v, w):

self.graph.append([u, v, w])

# Search function

def find(self, parent, i):

if parent[i] == i:

return i

return self.find(parent, parent[i])

def apply\_union(self, parent, rank, x, y):

xroot = self.find(parent, x)

yroot = self.find(parent, y)

if rank[xroot] < rank[yroot]:

parent[xroot] = yroot

elif rank[xroot] > rank[yroot]:

parent[yroot] = xroot

else:

parent[yroot] = xroot

rank[xroot] += 1

# Applying Kruskal algorithm

def kruskal\_algo(self):

result = []

i, e = 0, 0

self.graph = sorted(self.graph, key=lambda item: item[2])

parent = []

rank = []

for node in range(self.V):

parent.append(node)

rank.append(0)

while e < self.V - 1:

u, v, w = self.graph[i]

i = i + 1

x = self.find(parent, u)

y = self.find(parent, v)

if x != y:

e = e + 1

result.append([u, v, w])

self.apply\_union(parent, rank, x, y)

for u, v, weight in result:

print("%d - %d: %d" % (u, v, weight))

g = Graph(6)

g.add\_edge(0, 1, 4)

g.add\_edge(0, 2, 4)

g.add\_edge(1, 2, 2)

g.add\_edge(1, 0, 4)

g.add\_edge(2, 0, 4)

g.add\_edge(2, 1, 2)

g.add\_edge(2, 3, 3)

g.add\_edge(2, 5, 2)

g.add\_edge(2, 4, 4)

g.add\_edge(3, 2, 3)

g.add\_edge(3, 4, 3)

g.add\_edge(4, 2, 4)

g.add\_edge(4, 3, 3)

g.add\_edge(5, 2, 2)

g.add\_edge(5, 4, 3)

g.kruskal\_algo()

***Output***:

1 - 2: 2

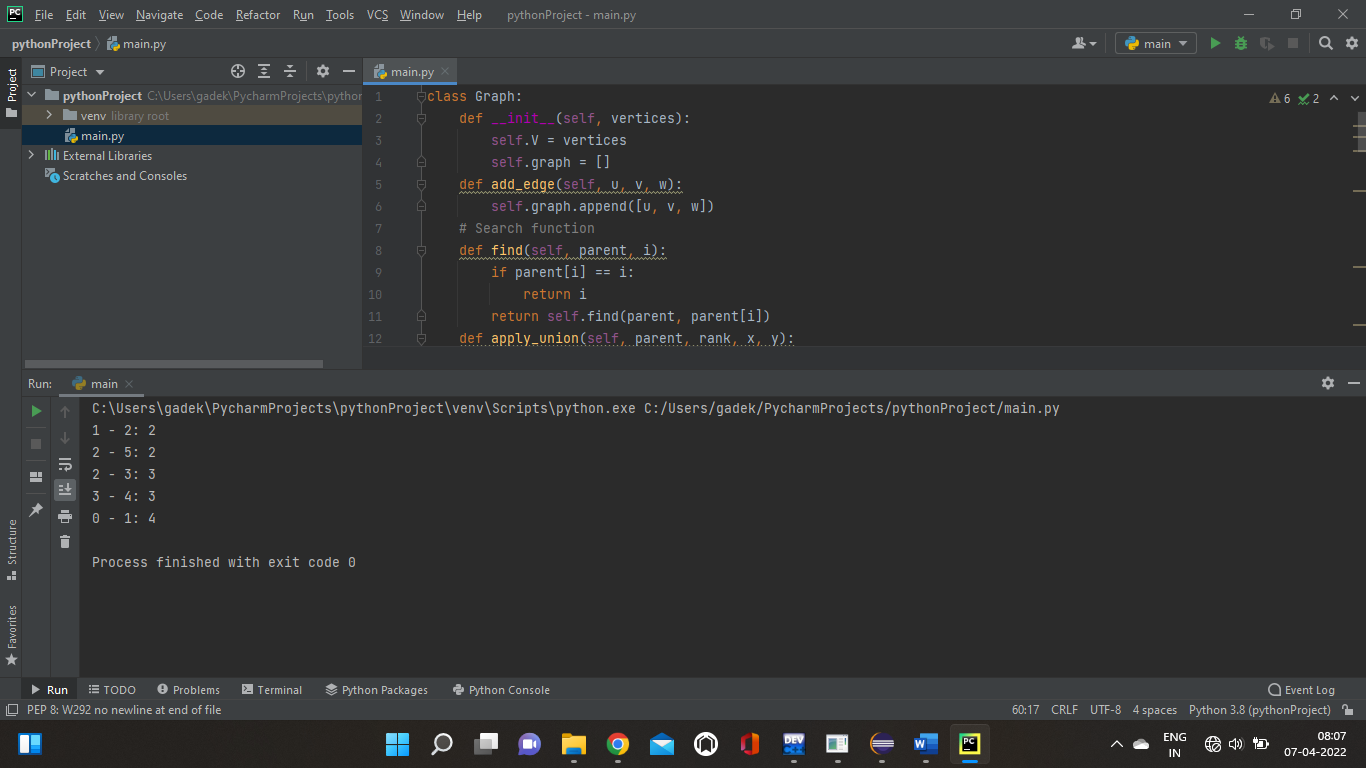
2 - 5: 2

2 - 3: 3

3 - 4: 3

0 - 1: 4

>



**AI : 04**

//C++ code for Graph Colouring using Backtracking

#include<bits/stdc++.h>

using namespace std;

#define V 4

void printSolution(int color[]);

bool isSafe(bool graph[V][V], int color[])

{

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

for (int j = i + 1; j < V; j++)

if (graph[i][j] && color[j] == color[i])

return false;

return true;

}

bool graphColoring(bool graph[V][V], int m, int i,

int color[V])

{

if (i == V) {

if (isSafe(graph, color)) {

printSolution(color);

return true;

}

return false;

}

Assign each color from 1 to m

for (int j = 1; j <= m; j++) {

color[i] = j;

if (graphColoring(graph, m, i + 1, color))

return true;

color[i] = 0;

}

return false;

}

void printSolution(int color[])

{

cout << "Solution Exists:" " Following are the assigned colors \n";

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

cout << " " << color[i];

cout << "\n";

}

int main()

{

bool graph[V][V] = {

{ 0, 1, 1, 1 },

{ 1, 0, 1, 0 },

{ 1, 1, 0, 1 },

{ 1, 0, 1, 0 },

};

int m = 3;

int color[V];

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

color[i] = 0;

if (!graphColoring(graph, m, 0, color))

cout << "Solution does not exist";

return 0;

}

