**AI Assignment 3** **Amir Mohideen**

1. Using Breadth First Search:

C++ Code:

// C++ program to find the shortest path between

// a given source cell to a destination cell.

#include <bits/stdc++.h>

using namespace std;

#define ROW 9

#define COL 10

//To store matrix cell cordinates

struct Point

{

int x;

int y;

};

// A Data Structure for queue used in BFS

struct queueNode

{

Point pt; // The cordinates of a cell

int dist; // cell's distance of from the source

};

// check whether given cell (row, col) is a valid

// cell or not.

bool isValid(int row, int col)

{

// return true if row number and column number

// is in range

return (row >= 0) && (row < ROW) &&

(col >= 0) && (col < COL);

}

// These arrays are used to get row and column

// numbers of 4 neighbours of a given cell

int rowNum[] = {-1, 0, 0, 1};

int colNum[] = {0, -1, 1, 0};

// function to find the shortest path between

// a given source cell to a destination cell.

int BFS(int mat[][COL], Point src, Point dest)

{

// check source and destination cell

// of the matrix have value 1

if (!mat[src.x][src.y] || !mat[dest.x][dest.y])

return -1;

bool visited[ROW][COL];

memset(visited, false, sizeof visited);

// Mark the source cell as visited

visited[src.x][src.y] = true;

// Create a queue for BFS

queue<queueNode> q;

// Distance of source cell is 0

queueNode s = {src, 0};

q.push(s); // Enqueue source cell

// Do a BFS starting from source cell

while (!q.empty())

{

queueNode curr = q.front();

Point pt = curr.pt;

// If we have reached the destination cell,

// we are done

if (pt.x == dest.x && pt.y == dest.y)

return curr.dist;

// Otherwise dequeue the front cell in the queue

// and enqueue its adjacent cells

q.pop();

for (int i = 0; i < 4; i++)

{

int row = pt.x + rowNum[i];

int col = pt.y + colNum[i];

// if adjacent cell is valid, has path and

// not visited yet, enqueue it.

if (isValid(row, col) && mat[row][col] &&

!visited[row][col])

{

// mark cell as visited and enqueue it

visited[row][col] = true;

queueNode Adjcell = { {row, col},

curr.dist + 1 };

q.push(Adjcell);

}

}

}

// Return -1 if destination cannot be reached

return -1;

}

// Driver program to test above function

int main()

{

int mat[ROW][COL] =

{

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

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

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

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

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

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

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

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

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

};

Point source = {0, 0};

Point dest = {8, 9};

int dist = BFS(mat, source, dest);

if (dist != INT\_MAX)

cout << "Shortest Path is " << dist ;

else

cout << "Shortest Path doesn't exist";

return 0;

}

Output:

1. Using Uniform Cost Search:

C++ code:

#include <bits/stdc++.h>

using namespace std;

// graph

vector<vector<int> > graph;

// map to store cost of edges

map<pair<int, int>, int> cost;

// returns the minimum cost in a vector

vector<int> uniform\_cost\_search(vector<int> goal, int start)

{

// minimum cost upto goal state from starting state

vector<int> answer;

// priority queue

priority\_queue<pair<int, int> > queue;

// set the answer vector to max value

for (int i = 0; i < goal.size(); i++)

answer.push\_back(INT\_MAX);

// insert the starting index

queue.push(make\_pair(0, start));

// map to store visited node

map<int, int> visited;

int count = 0;

// while queue is not empty

while (queue.size() > 0) {

// get top element of priority queue

pair<int, int> p = queue.top();

// pop element

queue.pop();

// get original value

p.first \*= -1;

// check if element is part of goal list

if (find(goal.begin(), goal.end(), p.second) != goal.end()) {

// get position

int index = find(goal.begin(), goal.end(),

p.second) - goal.begin();

// if new goal is reached

if (answer[index] == INT\_MAX)

count++;

// if cost is less

if (answer[index] > p.first)

answer[index] = p.first;

// pop element

queue.pop();

// if all goals are reached

if (count == goal.size())

return answer;

}

// check for non-visited nodes which are adjacent to present node

if (visited[p.second] == 0)

for (int i = 0; i < graph[p.second].size(); i++) {

// value is multiplied by -1 so that least priority is at the top

queue.push(make\_pair((p.first +

cost[make\_pair(p.second, graph[p.second][i])]) \* -1,

graph[p.second][i]));

}

// mark as visited

visited[p.second] = 1;

}

return answer;

}

// main function

int main()

{

// create the graph

graph.resize(7);

// add edge

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

graph[0].push\_back(3);

graph[3].push\_back(1);

graph[3].push\_back(6);

graph[3].push\_back(4);

graph[1].push\_back(6);

graph[4].push\_back(2);

graph[4].push\_back(5);

graph[2].push\_back(1);

graph[5].push\_back(2);

graph[5].push\_back(6);

graph[6].push\_back(4);

// add the cost

cost[make\_pair(0, 1)] = 2;

cost[make\_pair(0, 3)] = 5;

cost[make\_pair(1, 6)] = 1;

cost[make\_pair(3, 1)] = 5;

cost[make\_pair(3, 6)] = 6;

cost[make\_pair(3, 4)] = 2;

cost[make\_pair(2, 1)] = 4;

cost[make\_pair(4, 2)] = 4;

cost[make\_pair(4, 5)] = 3;

cost[make\_pair(5, 2)] = 6;

cost[make\_pair(5, 6)] = 3;

cost[make\_pair(6, 4)] = 7;

// goal state

vector<int> goal;

// set the goal

// there can be multiple goal states

goal.push\_back(6);

// get the answer

vector<int> answer = uniform\_cost\_search(goal, 0);

// print the answer

cout << "Minimum cost from 0 to 6 is = "

<< answer[0] << endl;

return 0;

}