**Artificial Intelligence Lab**

# Saket Kumar Baranwal (RA1911003010414)-G1

# Aim:- To Develope Best first search Algorithm for real world problems.

# *BEST FIRST SEARCH*

# Description:

# In BFS and DFS, when we are at a node, we can consider any of the adjacent as next node. So both BFS and DFS blindly explore paths without considering any cost function. The idea of Best First Search is to use an evaluation function to decide which adjacent is most promising and then explore.

# Algorithm:

# • Define a list, OPEN, consisting solely of a single node, the start node, *s*.

# • IF the list is empty, return failure.

# • Remove from the list the node *n* with the best score (the node where *f* is the minimum), and move it to a list, CLOSED.

# • Expand node *n*.

# • IF any successor to *n* is the goal node, return success and the solution (by tracing the path from the goal node to *s*).

# • FOR each successor node:

# 1.apply the evaluation function, *f*, to the node.

# 2. IF the node has not been in either list, add it to OPEN.

# • looping structure by sending the algorithm back to the second step.

# Code:-

# #include <bits/stdc++.h>

# using namespace std;

# typedef pair<int, int> pi;

# vector<vector<pi> > graph;

# // Function for adding edges to graph

# void addedge(int x, int y, int cost)

# {

# graph[x].push\_back(make\_pair(cost, y));

# graph[y].push\_back(make\_pair(cost, x));

# }

# // Function For Implementing Best First Search

# // Gives output path having lowest cost

# void best\_first\_search(int source, int target, int n)

# {

# vector<bool> visited(n, false);

# // MIN HEAP priority queue

# priority\_queue<pi, vector<pi>, greater<pi> > pq;

# // sorting in pq gets done by first value of pair

# pq.push(make\_pair(0, source));

# int s = source;

# visited[s] = true;

# while (!pq.empty()) {

# int x = pq.top().second;

# // Displaying the path having lowest cost

# cout << x << " ";

# pq.pop();

# if (x == target)

# break;

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

# if (!visited[graph[x][i].second]) {

# visited[graph[x][i].second] = true;

# pq.push(make\_pair(graph[x][i].first,graph[x][i].second));

# }

# }

# }

# }

# // Driver code to test above methods

# int main()

# {

# // No. of Nodes

# int v = 14;

# graph.resize(v);

# // The nodes shown in above example(by alphabets) are

# // implemented using integers addedge(x,y,cost);

# addedge(0, 1, 3);

# addedge(0, 2, 6);

# addedge(0, 3, 5);

# addedge(1, 4, 9);

# addedge(1, 5, 8);

# addedge(2, 6, 12);

# addedge(2, 7, 14);

# addedge(3, 8, 7);

# addedge(8, 9, 5);

# addedge(8, 10, 6);

# addedge(9, 11, 1);

# addedge(9, 12, 10);

# addedge(9, 13, 2);

# int source = 0;

# int target = 9;

# // Function call

# best\_first\_search(source, target, v);

# return 0;

# }

# Output:-

# 0 1 3 2 8 9

# 

# Aim:- To study the A\* Search Algorithm.

# *A\* Best First Search*

# Description:

# A\* is an informed search algorithm, or a best-first search, meaning that it is formulated in terms of weighted graphs: starting from a specific starting node of a graph, it aims to find a path to the given goal node having the smallest cost (least distance travelled, shortest time, etc.). It does this by maintaining a tree of paths originating at the start node and extending those paths one edge at a time until its termination criterion is satisfied.

# Code:-

# #include <bits/stdc++.h>

# using namespace std;

# #define ROW 9

# #define COL 10

# typedef pair<int, int> Pair;

# typedef pair<double, pair<int, int> > pPair;

# struct cell {

# int parent\_i, parent\_j;

# double f, g, h;

# };

# bool isValid(int row, int col)

# {

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

# && (col < COL);

# }

# bool isUnBlocked(int grid[][COL], int row, int col)

# {

# if (grid[row][col] == 1)

# return (true);

# else

# return (false);

# }

# bool isDestination(int row, int col, Pair dest)

# {

# if (row == dest.first && col == dest.second)

# return (true);

# else

# return (false);

# }

# double calculateHValue(int row, int col, Pair dest)

# {

# return ((double)sqrt(

# (row - dest.first) \* (row - dest.first)

# + (col - dest.second) \* (col - dest.second)));

# }

# void tracePath(cell cellDetails[][COL], Pair dest)

# {

# printf("\nThe Path is ");

# int row = dest.first;

# int col = dest.second;

# stack<Pair> Path;

# while (!(cellDetails[row][col].parent\_i == row

# && cellDetails[row][col].parent\_j == col)) {

# Path.push(make\_pair(row, col));

# int temp\_row = cellDetails[row][col].parent\_i;

# int temp\_col = cellDetails[row][col].parent\_j;

# row = temp\_row;

# col = temp\_col;

# }

# Path.push(make\_pair(row, col));

# while (!Path.empty()) {

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

# Path.pop();

# printf("-> (%d,%d) ", p.first, p.second);

# }

# return;

# }

# void aStarSearch(int grid[][COL], Pair src, Pair dest)

# {

# if (isValid(src.first, src.second) == false) {

# printf("Source is invalid\n");

# return;

# }

# // If the destination is out of range

# if (isValid(dest.first, dest.second) == false) {

# printf("Destination is invalid\n");

# return;

# }

# if (isUnBlocked(grid, src.first, src.second) == false

# || isUnBlocked(grid, dest.first, dest.second)

# == false) {

# printf("Source or the destination is blocked\n");

# return;

# }

# if (isDestination(src.first, src.second, dest)

# == true) {

# printf("We are already at the destination\n");

# return;

# }

# bool closedList[ROW][COL];

# memset(closedList, false, sizeof(closedList));

# cell cellDetails[ROW][COL];

# int i, j;

# for (i = 0; i < ROW; i++) {

# for (j = 0; j < COL; j++) {

# cellDetails[i][j].f = FLT\_MAX;

# cellDetails[i][j].g = FLT\_MAX;

# cellDetails[i][j].h = FLT\_MAX;

# cellDetails[i][j].parent\_i = -1;

# cellDetails[i][j].parent\_j = -1;

# }

# }

# i = src.first, j = src.second;

# cellDetails[i][j].f = 0.0;

# cellDetails[i][j].g = 0.0;

# cellDetails[i][j].h = 0.0;

# cellDetails[i][j].parent\_i = i;

# cellDetails[i][j].parent\_j = j;

# set<pPair> openList;

# openList.insert(make\_pair(0.0, make\_pair(i, j)));

# bool foundDest = false;

# while (!openList.empty()) {

# pPair p = \*openList.begin();

# openList.erase(openList.begin());

# i = p.second.first;

# j = p.second.second;

# closedList[i][j] = true;

# double gNew, hNew, fNew;

# if (isValid(i - 1, j) == true) {

# if (isDestination(i - 1, j, dest) == true) {

# // Set the Parent of the destination cell

# cellDetails[i - 1][j].parent\_i = i;

# cellDetails[i - 1][j].parent\_j = j;

# printf("The destination cell is found\n");

# tracePath(cellDetails, dest);

# foundDest = true;

# return;

# }

# else if (closedList[i - 1][j] == false

# && isUnBlocked(grid, i - 1, j)

# == true) {

# gNew = cellDetails[i][j].g + 1.0;

# hNew = calculateHValue(i - 1, j, dest);

# fNew = gNew + hNew;

# if (cellDetails[i - 1][j].f == FLT\_MAX

# || cellDetails[i - 1][j].f > fNew) {

# openList.insert(make\_pair(

# fNew, make\_pair(i - 1, j)));

# cellDetails[i - 1][j].f = fNew;

# cellDetails[i - 1][j].g = gNew;

# cellDetails[i - 1][j].h = hNew;

# cellDetails[i - 1][j].parent\_i = i;

# cellDetails[i - 1][j].parent\_j = j;

# }

# }

# }

# if (isValid(i + 1, j) == true) {

# 

# if (isDestination(i + 1, j, dest) == true) {

# cellDetails[i + 1][j].parent\_i = i;

# cellDetails[i + 1][j].parent\_j = j;

# printf("The destination cell is found\n");

# tracePath(cellDetails, dest);

# foundDest = true;

# return;

# }

# else if (closedList[i + 1][j] == false

# && isUnBlocked(grid, i + 1, j)

# == true) {

# gNew = cellDetails[i][j].g + 1.0;

# hNew = calculateHValue(i + 1, j, dest);

# fNew = gNew + hNew;

# if (cellDetails[i + 1][j].f == FLT\_MAX

# || cellDetails[i + 1][j].f > fNew) {

# openList.insert(make\_pair(

# fNew, make\_pair(i + 1, j)));

# cellDetails[i + 1][j].f = fNew;

# cellDetails[i + 1][j].g = gNew;

# cellDetails[i + 1][j].h = hNew;

# cellDetails[i + 1][j].parent\_i = i;

# cellDetails[i + 1][j].parent\_j = j;

# }

# }

# }

# if (isValid(i, j + 1) == true) {

# if (isDestination(i, j + 1, dest) == true) {

# // Set the Parent of the destination cell

# cellDetails[i][j + 1].parent\_i = i;

# cellDetails[i][j + 1].parent\_j = j;

# printf("The destination cell is found\n");

# tracePath(cellDetails, dest);

# foundDest = true;

# return;

# }

# else if (closedList[i][j + 1] == false

# && isUnBlocked(grid, i, j + 1)

# == true) {

# gNew = cellDetails[i][j].g + 1.0;

# hNew = calculateHValue(i, j + 1, dest);

# fNew = gNew + hNew;

# if (cellDetails[i][j + 1].f == FLT\_MAX

# || cellDetails[i][j + 1].f > fNew) {

# openList.insert(make\_pair(

# fNew, make\_pair(i, j + 1)));

# // Update the details of this cell

# cellDetails[i][j + 1].f = fNew;

# cellDetails[i][j + 1].g = gNew;

# cellDetails[i][j + 1].h = hNew;

# cellDetails[i][j + 1].parent\_i = i;

# cellDetails[i][j + 1].parent\_j = j;

# }

# }

# }

# if (isValid(i, j - 1) == true) {

# if (isDestination(i, j - 1, dest) == true) {

# // Set the Parent of the destination cell

# cellDetails[i][j - 1].parent\_i = i;

# cellDetails[i][j - 1].parent\_j = j;

# printf("The destination cell is found\n");

# tracePath(cellDetails, dest);

# foundDest = true;

# return;

# }

# else if (closedList[i][j - 1] == false

# && isUnBlocked(grid, i, j - 1)

# == true) {

# gNew = cellDetails[i][j].g + 1.0;

# hNew = calculateHValue(i, j - 1, dest);

# fNew = gNew + hNew;

# if (cellDetails[i][j - 1].f == FLT\_MAX

# || cellDetails[i][j - 1].f > fNew) {

# openList.insert(make\_pair(

# fNew, make\_pair(i, j - 1)));

# // Update the details of this cell

# cellDetails[i][j - 1].f = fNew;

# cellDetails[i][j - 1].g = gNew;

# cellDetails[i][j - 1].h = hNew;

# cellDetails[i][j - 1].parent\_i = i;

# cellDetails[i][j - 1].parent\_j = j;

# }

# }

# }

# if (isValid(i - 1, j + 1) == true) {

# if (isDestination(i - 1, j + 1, dest) == true) {

# // Set the Parent of the destination cell

# cellDetails[i - 1][j + 1].parent\_i = i;

# cellDetails[i - 1][j + 1].parent\_j = j;

# printf("The destination cell is found\n");

# tracePath(cellDetails, dest);

# foundDest = true;

# return;

# }

# else if (closedList[i - 1][j + 1] == false

# && isUnBlocked(grid, i - 1, j + 1)

# == true) {

# gNew = cellDetails[i][j].g + 1.414;

# hNew = calculateHValue(i - 1, j + 1, dest);

# fNew = gNew + hNew;

# if (cellDetails[i - 1][j + 1].f == FLT\_MAX

# || cellDetails[i - 1][j + 1].f > fNew) {

# openList.insert(make\_pair(

# fNew, make\_pair(i - 1, j + 1)));

# cellDetails[i - 1][j + 1].f = fNew;

# cellDetails[i - 1][j + 1].g = gNew;

# cellDetails[i - 1][j + 1].h = hNew;

# cellDetails[i - 1][j + 1].parent\_i = i;

# cellDetails[i - 1][j + 1].parent\_j = j;

# }

# }

# }

# if (isValid(i - 1, j - 1) == true) {

# if (isDestination(i - 1, j - 1, dest) == true) {

# // Set the Parent of the destination cell

# cellDetails[i - 1][j - 1].parent\_i = i;

# cellDetails[i - 1][j - 1].parent\_j = j;

# printf("The destination cell is found\n");

# tracePath(cellDetails, dest);

# foundDest = true;

# return;

# }

# else if (closedList[i - 1][j - 1] == false

# && isUnBlocked(grid, i - 1, j - 1)

# == true) {

# gNew = cellDetails[i][j].g + 1.414;

# hNew = calculateHValue(i - 1, j - 1, dest);

# fNew = gNew + hNew;

# if (cellDetails[i - 1][j - 1].f == FLT\_MAX

# || cellDetails[i - 1][j - 1].f > fNew) {

# openList.insert(make\_pair(

# fNew, make\_pair(i - 1, j - 1)));

# // Update the details of this cell

# cellDetails[i - 1][j - 1].f = fNew;

# cellDetails[i - 1][j - 1].g = gNew;

# cellDetails[i - 1][j - 1].h = hNew;

# cellDetails[i - 1][j - 1].parent\_i = i;

# cellDetails[i - 1][j - 1].parent\_j = j;

# }

# }

# }

# if (isValid(i + 1, j + 1) == true) {

# if (isDestination(i + 1, j + 1, dest) == true) {

# cellDetails[i + 1][j + 1].parent\_i = i;

# cellDetails[i + 1][j + 1].parent\_j = j;

# printf("The destination cell is found\n");

# tracePath(cellDetails, dest);

# foundDest = true;

# return;

# }

# else if (closedList[i + 1][j + 1] == false

# && isUnBlocked(grid, i + 1, j + 1)

# == true) {

# gNew = cellDetails[i][j].g + 1.414;

# hNew = calculateHValue(i + 1, j + 1, dest);

# fNew = gNew + hNew;

# if (cellDetails[i + 1][j + 1].f == FLT\_MAX

# || cellDetails[i + 1][j + 1].f > fNew) {

# openList.insert(make\_pair(

# fNew, make\_pair(i + 1, j + 1)));

# // Update the details of this cell

# cellDetails[i + 1][j + 1].f = fNew;

# cellDetails[i + 1][j + 1].g = gNew;

# cellDetails[i + 1][j + 1].h = hNew;

# cellDetails[i + 1][j + 1].parent\_i = i;

# cellDetails[i + 1][j + 1].parent\_j = j;

# }

# }

# }

# if (isValid(i + 1, j - 1) == true) {

# if (isDestination(i + 1, j - 1, dest) == true) {

# // Set the Parent of the destination cell

# cellDetails[i + 1][j - 1].parent\_i = i;

# cellDetails[i + 1][j - 1].parent\_j = j;

# printf("The destination cell is found\n");

# tracePath(cellDetails, dest);

# foundDest = true;

# return;

# }

# else if (closedList[i + 1][j - 1] == false

# && isUnBlocked(grid, i + 1, j - 1)

# == true) {

# gNew = cellDetails[i][j].g + 1.414;

# hNew = calculateHValue(i + 1, j - 1, dest);

# fNew = gNew + hNew;

# if (cellDetails[i + 1][j - 1].f == FLT\_MAX

# || cellDetails[i + 1][j - 1].f > fNew) {

# openList.insert(make\_pair(

# fNew, make\_pair(i + 1, j - 1)));

# // Update the details of this cell

# cellDetails[i + 1][j - 1].f = fNew;

# cellDetails[i + 1][j - 1].g = gNew;

# cellDetails[i + 1][j - 1].h = hNew;

# cellDetails[i + 1][j - 1].parent\_i = i;

# cellDetails[i + 1][j - 1].parent\_j = j;

# }

# }

# }

# }

# if (foundDest == false)

# printf("Failed to find the Destination Cell\n");

# return;

# }

# int main()

# {

# int grid[ROW][COL]

# = { { 1, 0, 1, 1, 1, 1, 0, 1, 1, 1 },

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

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

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

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

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

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

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

# { 1, 1, 1, 0, 0, 0, 1, 0, 0, 1 } };

# // Source is the left-most bottom-most corner

# Pair src = make\_pair(8, 0);

# // Destination is the left-most top-most corner

# Pair dest = make\_pair(0, 0);

# aStarSearch(grid, src, dest);

# return (0);

# }

# Output:-

# 