ARTIFICIAL INTELLIGENCE (18CSC305J) LAB

EXPERIMENT 5

To implement A* and Best First Search.

Harsh Goel
RA1811003010185
CSE-C1

Aim:

To implement A* and Best First Search.

Problem Description:

Breath-first search (BFS):

A* algorithm checks the cost for travelling to every node connected to the current node and it's heuristic value to find the best node to go to.

Best first search algorithm checks for every node connected to the current node it's heuristic value to find the best node to go to.

Problem Formulation:

In both the algorithms we will check all nodes which we can go from the current node.

In A* f = g + h (g is cost to travel to a node , h is it's heuristic value)

In Best first search f = h (h is it's heuristic value)

Algorithm:

A*:

1. We will start from the starting point and check for all eight neighbours which has least f(f=g+h) value.
2. Then we will go to that state and repeat the same process until we find the state.
3. If not found return not found.

Best first search:

1. We will start from the starting point and check for all neighbours which has least h value.
2. Then we will go to that state and repeat the same process until we find the state.
3. If not found return not found.

Source Code:

Language-C

A* :

```
#include <bits/stdc++.h>
using namespace std;

#define ROW 3
#define COL 3
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 (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) {

                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) {

                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)));

                    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) {

                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)));

                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) {

            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) {

                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) {

                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) {

            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 (foundDest == false)
        printf("Failed to find the Destination Cell\n");

    return;
}

int main()
{
    int grid[ROW][COL]
        = { {1,0,0},
            {1,1,1},
            {0,0,1} };

    Pair src = make_pair(0,0);
    Pair dest = make_pair(2,2);

    aStarSearch(grid, src, dest);

    return (0);
}
```

BFS:

```
#include <bits/stdc++.h>
using namespace std;
typedef pair<int, int> pi;
vector<vector<pi> > 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));
}

void best_first_search(int source, int target, int n)
{
```

```
    vector<bool> visited(n, false);
    priority_queue<pi, vector<pi>, greater<pi> > pq;
    pq.push(make_pair(0, source));
    visited[0]=true;
    while (!pq.empty()) {
        int x = pq.top().second;

        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(graph[x][i]);
            }
        }
    }
}

int main()
{
    int v = 4;
    graph.resize(v);

    addedge(0,1,3);
    addedge(0,2,6);
    addedge(1,3,5);
    addedge(2,3,6);

    int source = 0;
    int target = 3;

    best_first_search(source, target, v);

    return 0;
}
```

TEST CASE:

Case 1: A*

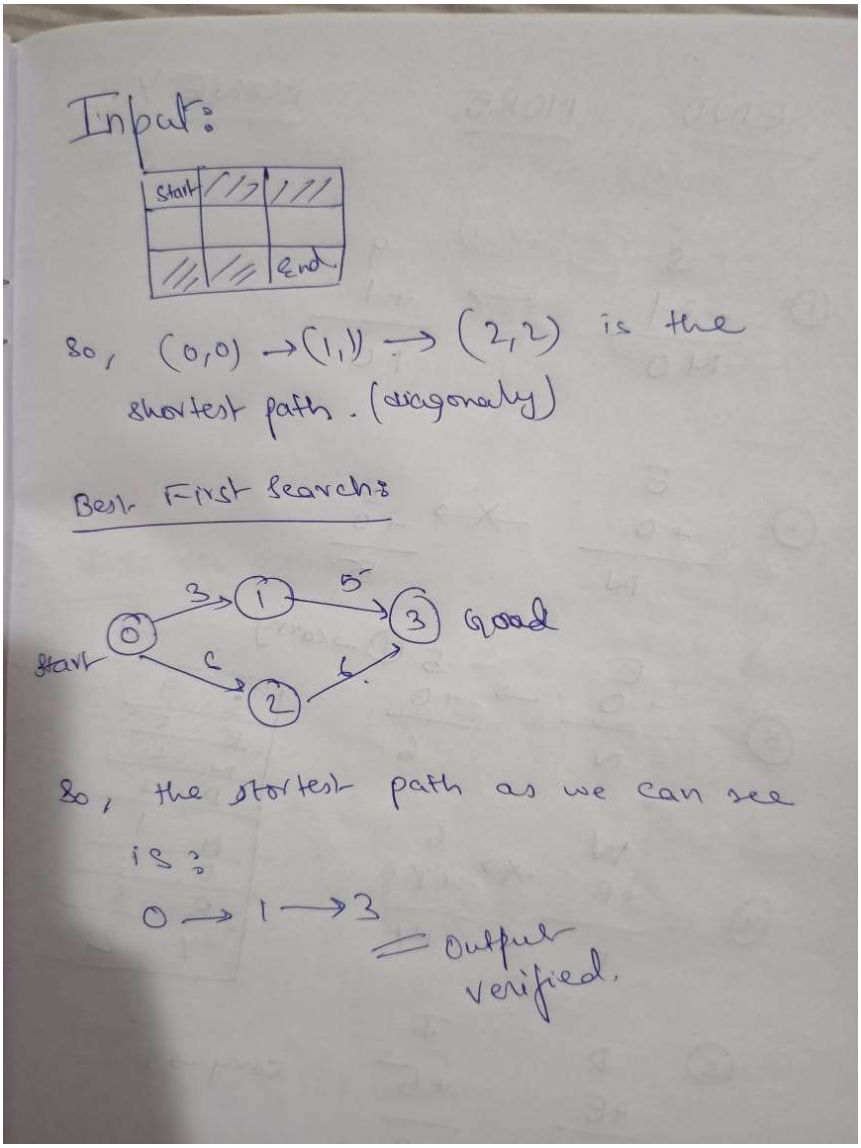
```
The destination cell is found

The Path is -> (0,0) -> (1,1) -> (2,2)
--------------------------------
```

Case 2: BFS

```
0 1 3
--------------------------------
```

Verification:



Result: Hence, successfully implemented both A* and Best First Search algorithms and verified test cases.