VISVESVARAYA TECHNOLOGICAL UNIVERSITY

Jnana Sangama, Belagavi-590018, Karnataka

Project Report
on

"A* Maze Solver"

Submitted by

USN	Name
1BI17EC089	Prathik P
1BI17EC092	Rahul Bharadwaj
1BI17EC122	Sourav B M
1BI17EC091	R D Karthik

Under the Guidance of

Dr. SUNEETHA K R

Associate Professor

Department of CS&E, BIT

Bengaluru-560004

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

BANGALORE INSTITUTE OF TECHNOLOGY

K.R. Road, V.V.Pura, Bengaluru-560 004

2019-20 (Odd)

VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belagavi-590018, Karnataka

BANGALORE INSTITUTE OF TECHNOLOGY

Bengaluru-560 004

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

Certificate

Certified that the Mini Project work entitled **"A* Maze Solver"** carried out by

USN	NAME
1BI17EC089	Prathik P
1BI17EC092	Rahul Bharadwaj
1BI17EC122	Sourav B M
1BI17EC091	R D Karthik

of V semester, Computer Science and Engineering branch as partial fulfillment of the course **Artificial Intelligence (17CS562)** prescribed by **Visvesvaraya Technological University, Belgaum** during the academic year 2019-20. It is certified that all corrections/suggestions indicated for Internal Assessment have been incorporated in the report.

The Mini Project report has been approved as it satisfies the academic requirements in respect of project work in Artificial Intelligence.

Dr. Suneetha K R
Associate Professor,
Department of CS&E, BIT,
Bengaluru-560004

CONTENTS

CHAPTER 1

INTRODUCTION

1.1 Introduction

In general, a maze has a complicated path like a labyrinth. To solve a maze is a game or puzzle which connects a path between a start and a goal. There are a lot of studies about shortest path algorithms.

Reaching a destination via a shortest route is a daily activity we all do. A* is a generic search algorithm used to find solutions for several problems, maze solving basically to be one of them which is the process of plotting an efficiently traversable path between points, called nodes. It is one of the most successful search algorithms. It is an informed search algorithm , as it uses information about path cost and also heuristics to find the solution.

A* search algorithm outperforms Dijkstra's to solve a maze in terms of search time and is based on the Dijkstra's method. The difference is to use a cost function of searching paths in the A* search algorithm to solve maze more quickly. This algorithm brings together feature of uniform-cost search and heuristic search. When a location is examined, the A* algorithm is completed when that location is the goal. In any other case, it helps make note of all that location neighbors for additional exploration. A* could possibly be the most popular pathfinding algorithm in game AI. The time complexity of this algorithm is depending on heuristic use. A*achieves *optimality* and *completeness*, two important property of search algorithms

1.2 Motivation

In the business world today, not a day passes without advancement in Artificial Intelligence (AI) sector as it has become mainstream. Industry experts are of the opinion that Artificial Intelligence is closely connected to popular culture thus driving the people to have improbable expectations and unrealistic fears about how it will alter the life and workplace in general.

AI is spreading throughout various industries. Some of them include education, healthcare, banking, finance, and many more. Industries that has adopted AI on a bigger level are more from the service industry. Even industries like web design and development being impacted by AI in a huge manner.

CHAPTER 2

PROBLEM STATEMENT

2.1 Problem Statement

Consider the problem of finding a route across the maze of ROWxCOL grid. The start position is (x,y) and the end position is (p,q). Movement is allowed by one square in any of the four directions. The standard movement cost is 1. There are barriers that occupy certain positions of the grid.

A route with the lowest cost should be found using the A* search algorithm (there are multiple optimal solutions with the same total cost).

Print the optimal route in text format, as well as the total cost of the route.

Optionally, draw the optimal route and the barrier positions.

2.2 Objectives

- The game environment will be handled as a graph. This involves breaking the map into different points or locations, which are usually called nodes. These nodes are used to record the progress of the search.
- In addition to holding the map location, each node has three other attributes that are fitness, goal, and heuristic commonly known as f, g, and h. Different values will be assigned to paths between the nodes.
- Typically, these values would represent the distances between the nodes. The cost between nodes doesn't have to be distance. The cost could be time, if you wanted to find the path that takes the shortest amount of time to traverse. A* using two lists (open list and closed list). The open list contains all the nodes in the map that have not been totally explored yet. The closed list contains of all the nodes that have been totally explored. In addition to the standard open/closed lists, marker arrays can be used to find out whether a state is in the open or closed list.

CHAPTER 3

SYSTEM REQUIREMENTS

3.1 Hardware requirements

PC with Windows/Linux OS/MacOS.

3.2 Software Requirements

Windows/Linux OS/MacOS with a C++ IDE.

CHAPTER 4

ARCHITECTURE

Now to understand how A* works, first we need to understand few terminologies:

- **Node** (also called **State**) — All potential position or stops with unique identification.
- **Transition** — The act of moving between states or nodes.
- **Starting Node —** Where to start searching from.
- **Goal Node—** The target to stop searching.
- **Search Space** — A collection of nodes, like all board positions of a board game.
- **Cost** — Numerical value (say distance, time taken or financial expense) for the path from a node to another node.

  g(n) — this represents the ***exact cost*** of the path from the **starting node** to any node **n.**
- **h(n)** — this represents the heuristic ***estimated cost*** from node **n** to the goal node.
- **f(n)** — lowest cost in the neighboring node n.

Each time A* enters a node, it calculates the cost f(n) of all of the neighboring nodes, and then enters the node with the lowest value of f(n). It is calculated with the formula: f(n)=g(n)+h(n). The purpose of g(n), h(n), and f(n) is to quantify how promising a path is.

4.1 Algorithm

Initialize open and closed lists

Make the start vertex current

Calculate heuristic distance of start vertex to destination (h)

Calculate f value for start vertex (f-g+h, where g - 0)

WHILE current vertex is not the destination

FOR each vertex adjacent to current

IF vertex not in closed list and not in open list THEN

Add vertex to open list

Calculate distance from start (g)

Calculate heuristic distance to destination (h)

Calculate f value (f g+h)

IF new f value < existing f value or there is no existing f value THEN

Update f value

Set parent to be the current vertex

END IF

END IF

NEXT adjacent vertex

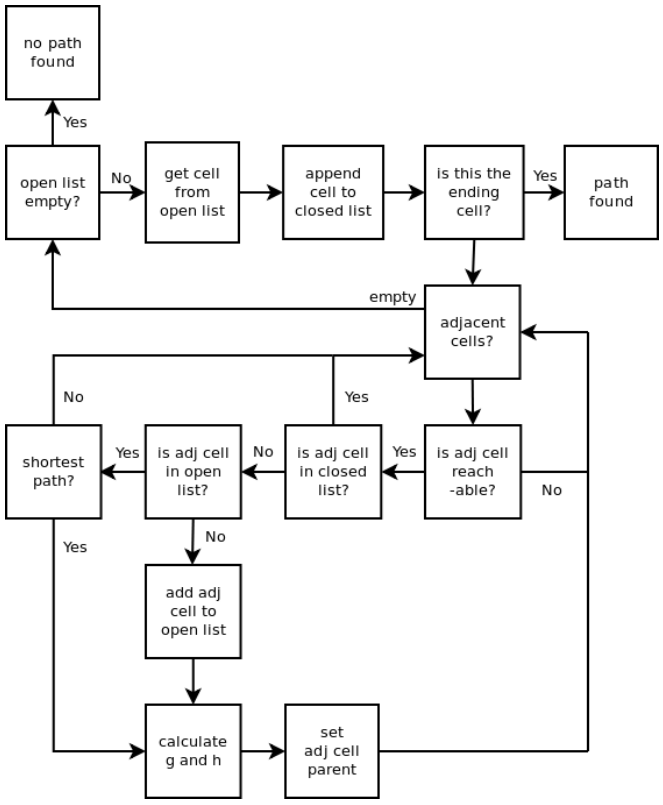

Figure 4.1 – Flowchart of the A Algorithm*

Add current vertex to closed list

Remove vertex with lowest f value from open list and make it current

END WHILE

CHAPTER 5

MODULE DESCRIPTIONS

5.1 Module Description

struct cell - A structure is created to hold the necessary parameters f(x), g(x) and h(x).

bool isValid(int row, int col) - A Utility Function to check whether given cell (row, col) is a valid cell or not.

bool isDestination(int row, int col, Pair dest) - A Utility Function to check whether destination cell has been reached or not.

double calculateHValue(int row, int col, Pair dest) - A Utility Function to calculate the 'h' heuristics when prompted.

void tracePath(cell cellDetails[][COL], Pair dest) - A Utility Function to trace the path from the source to destination after successfully tracing the shortest path.

void aStarSearch(int grid[][COL], Pair src, Pair dest) - A Function to find the shortest path between a given source cell to a destination cell according to A* Search Algorithm. Initializes the parameters of the starting node and creates an open list and closed list. Then Searches through them while removing and adding from these lists as required.

int main () - Driver program to test above function. Description of the Grid is given by 1 for the cells that are not blocked and 0 for the cells that are blocked.

In the main driver program the Source and Destination are declared.

5.2 Example

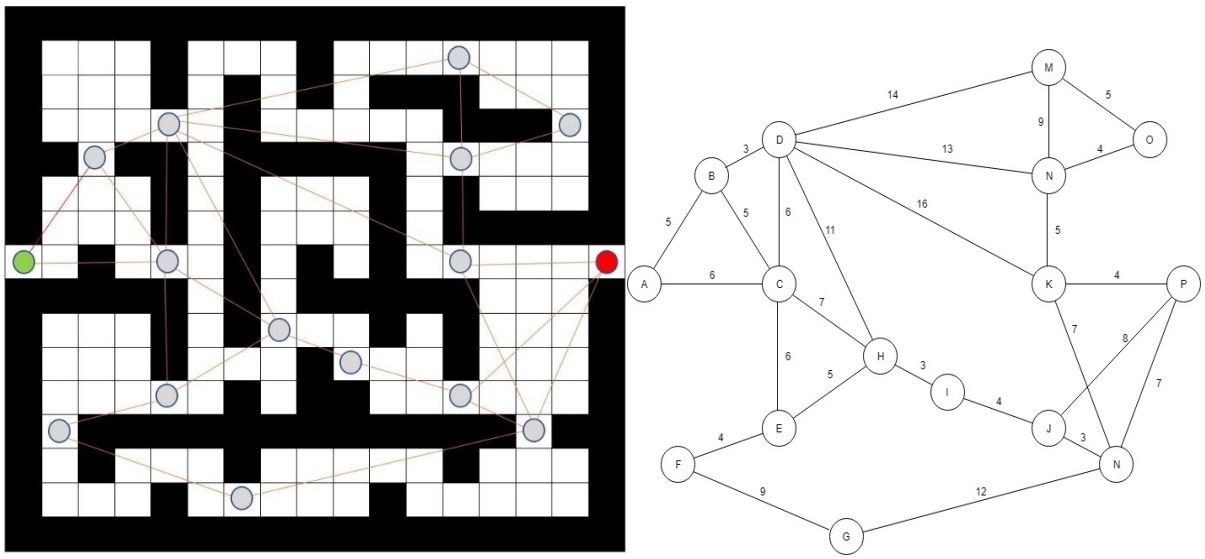

Figure 5.1 – Maze taken as an example and Maze represented as a Graph

Table 5.1 – Values of g(x), h(x) and f(x) with their parent nodes

Node	Distance from A (g)	Heuristic Distance (h)	f = g + h	Previous Node
A	0	16	16	
B	5	17	22	A
C	5	13	18	A
D	8	16	24	B
E	12	16	28	C
F		20		
G		17		
H	13	11	24	C
I	16	10	26	H
J	20	8	28	I
K	24	4	28	D
L	21	7	28	D
M	22	10	32	D
N	23	7	30	J
O		5		
P	28	0	28	J

CHAPTER 6

IMPLEMENTATION DETAILS

6.1 Source Code

```
// A C++ Program to implement A* Search Algorithm
#include<bits/stdc++.h>
using namespace std;
#define ROW 16
#define COL 17
// Creating a shortcut for int, int pair type
typedef pair<int, int> Pair;
// Creating a shortcut for pair<int, pair<int, int>> type
typedef pair<double, pair<int, int> > pPair;
// A structure to hold the neccesary parameters
struct cell{
	// Row and Column index of its parent
	// Note that 0 <= i <= ROW-1 & 0 <= j <= COL-1
	int parent_i, parent_j;
	// f = g + h
	double f, g, h;
};
// A Utility Function to check whether given cell (row, col) is a valid cell or not.
bool isValid(int row, int col){
	// Returns true if row number and column number is in range
	return (row >= 0) && (row < ROW) &&
		(col >= 0) && (col < COL);
}
```

```
// A Utility Function to check whether the given cell is blocked or not
bool isUnBlocked(int grid[][COL], int row, int col){
	// Returns true if the cell is not blocked else false
	if (grid[row][col] == 1)
		return (true);
	else
		return (false);
}
// A Utility Function to check whether destination cell has been reached or not
bool isDestination(int row, int col, Pair dest){
	if (row == dest.first && col == dest.second)
		return (true);
	else
		return (false);
}
// A Utility Function to calculate the 'h' heuristics.
double calculateHValue(int row, int col, Pair dest){
	// Return using the distance formula
	return((double)sqrt((row-dest.first)*(row-dest.first)+(col-dest.second)*(col-
dest.second)));
}
// A Utility Function to trace the path from the source to destination
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;
}
// A Function to find the shortest path between a given source cell to a destination cell
//according to A* Search Algorithm
void aStarSearch(int grid[][COL], Pair src, Pair dest){
        // If the source is out of range
        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;
    }
    // Either the source or the destination is blocked
    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 the destination cell is the same as source cell
    if (isDestination(src.first, src.second, dest) == true){
        printf ("We are already at the destination\n");
        return;
    }
    // Create a closed list and initialise it to false which means that no cell has been included
    //yet. This closed list is implemented as a boolean 2D array
    bool closedList[ROW][COL];
    memset(closedList, false, sizeof (closedList));
    // Declare a 2D array of structure to hold the details of that cell
    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;
```

```
        }
}
// Initialising the parameters of the starting node
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;
/* Create an open list having information as- <f, <i, j>>
where f = g + h, and i, j are the row and column index of that cell
Note that 0 <= i <= ROW-1 & 0 <= j <= COL-1
This open list is implenented as a set of pair of pair.*/
set<pPair> openList;
// Put the starting cell on the open list and set its 'f' as 0
openList.insert(make_pair (0.0, make_pair (i, j)));
// We set this boolean value as false as initially the destination is not reached.
bool foundDest = false;
while (!openList.empty()){
        pPair p = *openList.begin();
        // Remove this vertex from the open list
        openList.erase(openList.begin());

        // Add this vertex to the closed list
        i = p.second.first;
        j = p.second.second;
```

```
            closedList[i][j] = true;
        /*Generating all the 4 successor of this cell
                        N
                    W--Cell--E
                        S
            Cell-->Popped Cell (i, j)
            N --> North      (i-1, j)
            S --> South      (i+1, j)
            E --> East       (i, j+1)
            W --> West       (i, j-1) */
            // To store the 'g', 'h' and 'f' of the 4 successors
            double gNew, hNew, fNew;
            //----------- 1st Successor (North) ------------
            // Only process this cell if this is a valid one
            if (isValid(i-1, j) == true) {
                    // If the destination cell is the same as the current successor
                    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;
                    }
                    // If the successor is already on the closed list or if it is blocked, then
                    //ignore it. Else do the following
```

```
            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 it isn't on the open list, add it to the open list. Make the
                    //current square the parent of this square. Record f, g, and h costs
                    //of the square cell OR If it is on the open list already, check to
                    //see if this path to that square is better, using 'f' cost as measure.
                   if(cellDetails[i-1][j].f==FLT_MAX||cellDetails[i-1][j].f>fNew){
                            openList.insert( make_pair(fNew,make_pair(i-1, j)));
                            // Update the details of this cell
                            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;
                    }
            }
    }
    //----------- 2nd Successor (South) ------------
    // Only process this cell if this is a valid one
    if (isValid(i+1, j) == true) {
            // If the destination cell is the same as the current successor
            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;
            }
            // If the successor is already on the closed list or if it is blocked, then
            //ignore it. Else do the following
            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 it isn't on the open list, add it to the open list. Make the
                //current square the parent of this square. Record f, g, and h costs
                //of the square cell OR If it is on the open list already, check to
                //see if this path to that square is better, using 'f' cost as measure
            if(cellDetails[i+1][j].f==FLT_MAX||cellDetails[i+1][j].f>fNew) {
                        openList.insert( make_pair (fNew, make_pair (i+1, j)));
                        // Update the details of this cell
                        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;
                }
            }
        }
        //----------- 3rd Successor (East) ------------
        // Only process this cell if this is a valid one
```

```
if (isValid (i, j+1) == true) {
        // If the destination cell is the same as the current successor
        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;
        }
        // If the successor is already on the closed list or if it is blocked, then
        //ignore it. Else do the following
        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 it isn't on the open list, add it to the open list. Make the
                //current square the parent of this square. Record f, g, and h costs
                //of the square cell OR If it is on the open list already, check to
                //see if this path to that square is better, using 'f' cost as measure.
        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;
                }
            }
        }
        //----------- 4th Successor (West) ------------
        // Only process this cell if this is a valid one
        if (isValid(i, j-1) == true){
            // If the destination cell is the same as the current successor
            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;
            }
            // If the successor is already on the closed list or if it is blocked, then
            //ignore it. Else do the following
            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 it isn't on the open list, add it to open list. Make the current
                //square the parent of this square. Record the f, g, and h costs of
                //the square cell OR If it is on the open list already, check to see
                //if this path to that square is better, using 'f' cost as the measure.
```

```
                        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;
                        }
                }
        }
    }
    // When destination cell is not found and the open list is empty, we conclude that we
    //failed to reach the destiantion cell. This may happen when there is no way to reach
    //destination cell (due to blockages)
    if (foundDest == false)
        printf("Failed to find the Destination Cell\n");
    return;
}
int main(){
    // 1--> The cell is not blocked
    // 0--> The cell is blocked
int grid[ROW][COL] ={
        {0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0},
        {0,1,1,1,0,1,1,1,0,1,1,1,1,1,1,0},
        {0,1,1,1,0,1,0,1,0,1,0,0,0,1,1,0},
        {0,1,1,1,1,1,0,1,1,1,1,1,0,0,0,1,0},
```

```
            {0,0,1,0,0,1,0,0,0,0,0,1,1,1,1,1,0},
            {0,1,1,1,0,1,0,1,1,1,0,1,0,1,1,1,0},
            {0,1,1,1,0,1,0,1,1,1,0,1,0,0,0,0,0},
            {1,1,0,1,1,1,0,1,0,1,0,1,1,1,1,1,1},
            {0,0,0,0,0,1,0,1,0,0,0,0,0,1,1,1,0},
            {0,1,1,1,0,1,0,1,1,1,0,1,0,1,1,1,0},
            {0,1,1,1,0,1,1,1,0,1,1,1,0,1,1,1,0},
            {0,1,1,1,1,1,0,1,0,0,1,1,1,1,1,1,0},
            {0,1,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0},
            {0,1,0,1,1,1,0,1,1,1,1,1,0,1,1,1,0},
            {0,1,1,1,0,1,1,1,1,1,0,1,1,1,1,1,0},
            {0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0}
        };
        Pair src = make_pair(7, 0);
        Pair dest = make_pair(7, 16);
        aStarSearch(grid, src, dest);
        return(0);
}
```

CHAPTER 7

RESULTS

7.1. Output

A custom maze is created and solved with the help of A* algorithm and is implemented with the help of a C++ code.

The destination cell is found

The Path is -> (7,0) -> (7,1) -> (6,1) -> (6,2) -> (6,3) -> (7,3) -> (7,4) -> (7,5) -> (8,5) -> (9,5) -> (10,5) -> (10,6) -> (10,7) -> (9,7) -> (9,8) -> (9,9) -> (10,9) -> (10,10) -> (10,11) -> (11,11) -> (11,12) -> (11,13) -> (10,13) -> (9,13) -> (9,14) -> (8,14) -> (8,15) -> (7,15) -> (7,16)

CHAPTER 8

APPLICATIONS

- Games - Games with bots use a modified version of the A* algorithm to find their path to the player/object.
- Town Planning - Used to optimize the placement of essential utilities from a home unit.
- Space Exploration – Used in rovers to calculate the most efficient path around a crater/rock.
- flight path optimization – Optimum use of airspace and fuel by a whole fleet is maintained by planning the most effective paths for flights to take.
- Rescue Robots - Robots are widely used in many areas, where Human life cannot be compromised. In Rescue operations, the robots need to search for the trapped humans and find the efficient path between the robot and the trapped person. This path is planned using a Hybrid A*
- NLP - Parsing using stochastic grammars.

CHAPTER 9

CONCLUSION AND FUTURE ENHANCEMENT

9.1 Conclusion

A-star (A*) is a mighty algorithm in Artificial Intelligence with a wide range of usage. However, it is only as good as its heuristic function (which can be highly variable considering the nature of a problem). A* is the most popular choice for pathfinding because it's reasonably flexible.

9.2 Future Enhancement

A* has a low efficiency in the path selection process of very complex networks, if the search space is large and the entity is irregular, this algorithm is often unfavorable to search the optimal path, expanding a lot of redundant nodes and consuming much time. The presence of local minima deviates original A* from the best path.

A* algorithm considers only the local costs rather than the global costs and results in some substandard results, we may say that it has erroneously moved in the wrong direction. Further examination of nodes, existing in the linked list, by generating their children or the subsequent nodes could provide greater information regarding the cost and may lead to the best strategy for query optimization.

A closed list need not be maintained and that a simple onList flag can be set for each node, thereby removing the need to search a closed list repeatedly.

The A-star algorithm and it's variations like Iterative Deepening A-star (IDA-star), Jump Point Search (JPS), etc. are the most popular algorithm in the field of Path-finding in Maze and are enhanced versions of A* that are optimized for their usage.

BIBLIOGRAPHY

[1] Shrawan Kr. Sharm and B.L.Pal, "Shortest Path Searching for Road Network using A* Algorithm", IJCSMC, Vol. 4, Issue. 7, July 2015, pg.513 – 522

[2] Masoud Nosrati, Ronak Karimi and Hojat Allah Hasanvand, "Investigation of the * (Star) Search Algorithms: Characteristics, Methods and Approaches", World Applied Programming, Vol (2), No (4), April 2012. 251-256

[3] Xiao Cui and Hao Shi, "A*-based Pathfinding in Modern Computer Games", IJCSNS International Journal of Computer Science and Network Security, VOL.11 No.1, pg125-130, January 2011

[4] https://www.geeksforgeeks.org/a-search-algorithm/

[5] https://stackabuse.com/basic-ai-concepts-a-search-algorithm/

[6] https://www.laurentluce.com/posts/solving-mazes-using-python-simple-recursivity-and-a-search/

[7] https://rosettacode.org/wiki/A*_search_algorithm#C.2B.2B