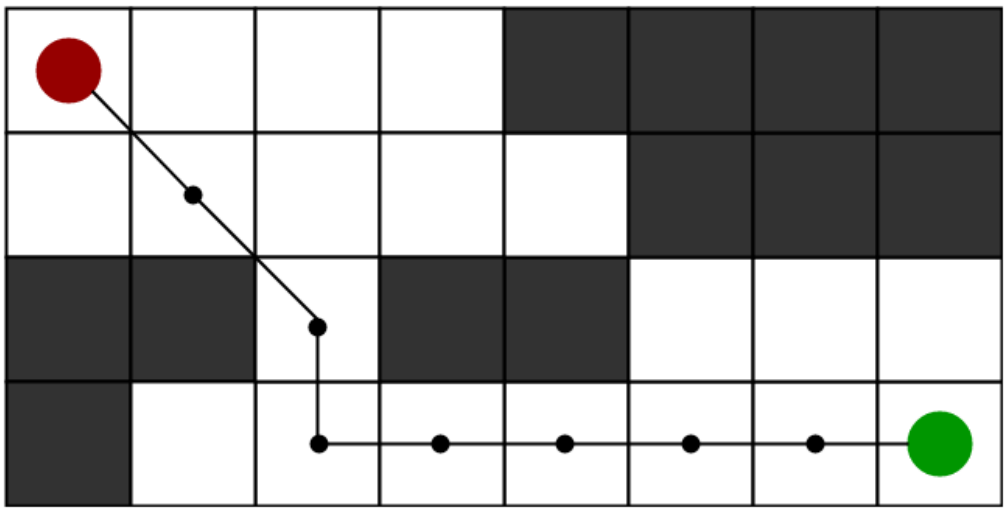
A* Search Algorithm

Motivation

To approximate the shortest path in real-life situations, like- in maps, games where there can be many hindrances.

We can consider a 2D Grid having several obstacles and we start from a source cell (colored red below) to reach towards a goal cell (colored green below)



What is A* Search Algorithm?

A* Search algorithm is one of the best and popular technique used in path-finding and graph traversals.

Why A* Search Algorithm?

Informally speaking, A* Search algorithms, unlike other traversal techniques, it has “brains”. What it means is that it is really a smart algorithm which separates it from the other conventional algorithms. This fact is cleared in

And it is also worth mentioning that many games and web-based maps use this algorithm to find the shortest path very efficiently (approximation).

Explanation

Consider a square grid having many obstacles and we are given a starting cell and a target cell. We want to reach the target cell (if possible) from the starting cell as quickly as possible. Here A* Search Algorithm comes to the rescue.

What A* Search Algorithm does is that at each step it picks the node according to a value-'**f**' which is a parameter equal to the sum of two other parameters – '**g**' and '**h**'. At each step it picks the node/cell having the lowest '**f**', and process that node/cell.

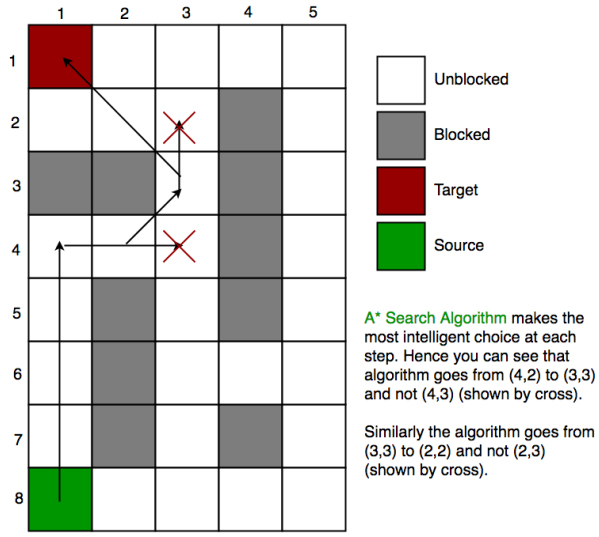
We define '**g**' and '**h**' as simply as possible below

g = the movement cost to move from the starting point to a given square on the grid, following the path generated to get there.

h = the estimated movement cost to move from that given square on the grid to the final destination. This is often referred to as the heuristic, which is nothing but a kind of smart guess. We really don't know the actual distance until we find the path, because all sorts of things can be in the way (walls, water, etc.). There can be many ways to calculate this 'h' which are discussed in the later sections.

```
// A* Search Algorithm
1.  Initialize the open list
2.  Initialize the closed list
    put the starting node on the open
    list (you can leave its f at zero)
3.  while the open list is not empty
    a) find the node with the least f on
       the open list, call it "q"
    b) pop q off the open list

    c) generate q's 8 successors and set their
       parents to q

    d) for each successor
        i) if successor is the goal, stop search

        ii) else, compute both g and h for successor
          successor.g = q.g + distance between
                              successor and q
          successor.h = distance from goal to
          successor (This can be done using many
          ways, we will discuss three heuristics-
          Manhattan, Diagonal and Euclidean
          Heuristics)

          successor.f = successor.g + successor.h
        iii) if a node with the same position as
            successor is in the OPEN list which has a
           lower f than successor, skip this successor
        iV) if a node with the same position as
            successor  is in the CLOSED list which has
            a lower f than successor, skip this successor
            otherwise, add  the node to the open list
     end (for loop)
```

So suppose as in the below figure if we want to reach the target cell from the source cell, then the A* Search algorithm would follow path as shown below. Note that the below figure is made by considering Euclidean Distance as a heuristics.



Heuristics

We can calculate **g** but how to calculate h ?

We can do things.

A) Either calculate the exact value of h (which is certainly time consuming).

OR

B) Approximate the value of h using some heuristics (less time consuming).

We will discuss both of the methods.

A) **Exact Heuristics** –

We can find exact values of h, but that is generally very time consuming.

Below are some of the methods to calculate the exact value of h.

1) Pre-compute the distance between each pair of cells before running the A* Search Algorithm.

2) If there are no blocked cells/obstacles then we can just find the exact value of h without any pre-computation using the distance formula/Euclidean Distance

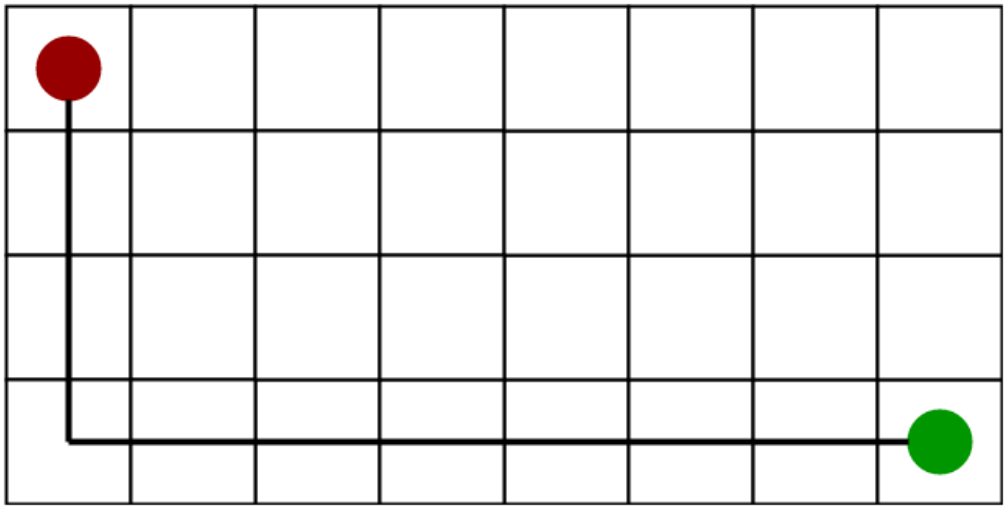
B) Approximation Heuristics –

- It is nothing but the sum of absolute values of differences in the goal's x and y coordinates and the current cell's x and y coordinates respectively, i.e.,

```
h = abs (current_cell.x - goal.x) +
    abs (current_cell.y - goal.y)
```

- When to use this heuristic? – When we are allowed to move only in four directions only (right, left, top, bottom)

The Manhattan Distance Heuristics is shown by the below figure (assume red spot as source cell and green spot as target cell).



2) Diagonal Distance-

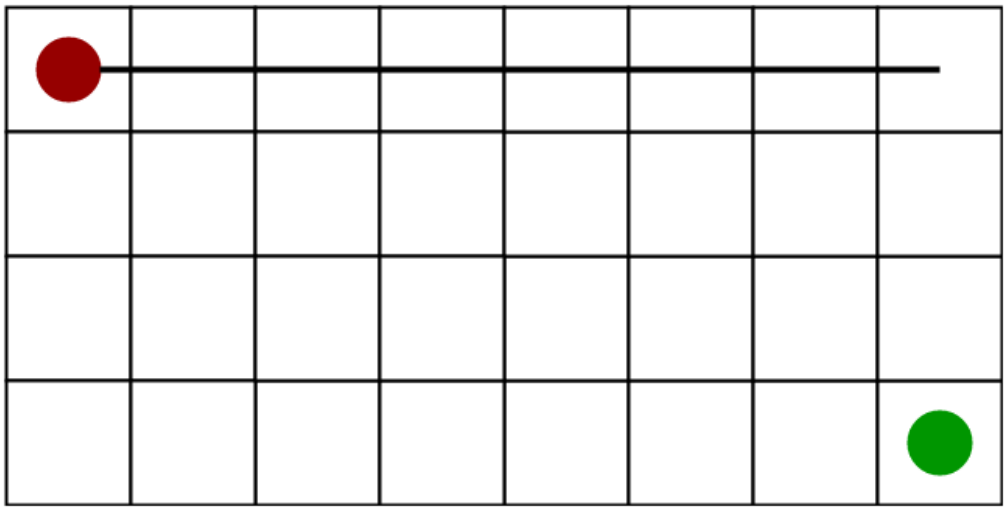
- It is nothing but the maximum of absolute values of differences in the goal's x and y coordinates and the current cell's x and y coordinates respectively, i.e.,

```
dx = abs(current_cell.x - goal.x)
dy = abs(current_cell.y - goal.y)
```

```
between each node (usually = sqrt(2) ).
```

- When to use this heuristic? – When we are allowed to move in eight directions only (similar to a move of a King in Chess)

The Diagonal Distance Heuristics is shown by the below figure (assume red spot as source cell and green spot as target cell).



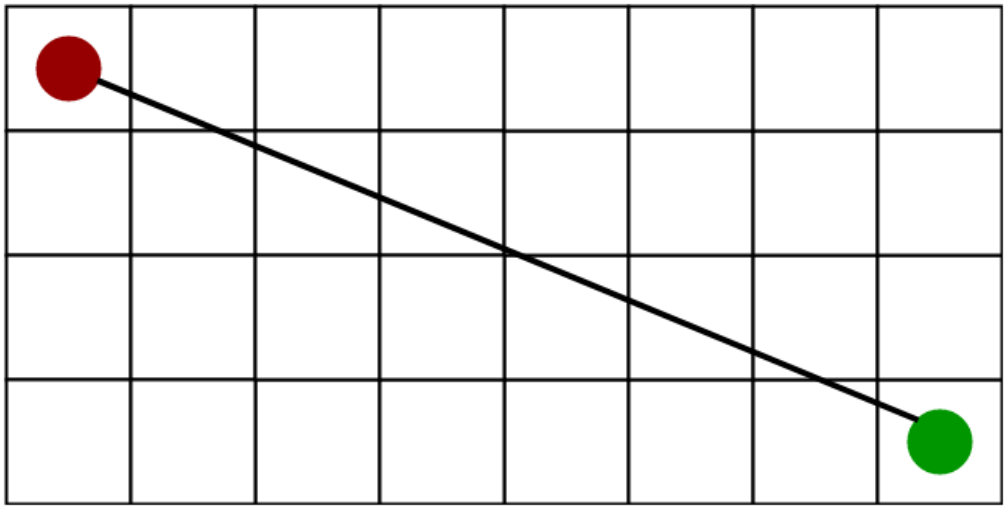
3) Euclidean Distance-

- As it is clear from its name, it is nothing but the distance between the current cell and the goal cell using the distance formula

```
h = sqrt ( (current_cell.x - goal.x)2 +
           (current_cell.y - goal.y)2 )
```

- When to use this heuristic? – When we are allowed to move in any directions.

The Euclidean Distance Heuristics is shown by the below figure (assume red spot as source cell and green spot as target cell).



Relation (Similarity and Differences) with other algorithms-

Dijkstra is a special case of A* Search Algorithm, where h = 0 for all nodes.

Implementation

We can use any data structure to implement open list and closed list but for best performance, we use a **set** data structure of C++ STL(implemented as Red-Black Tree) and a boolean hash table for a closed list.

The implementations are similar to Dijkstra's algorithm. If we use a Fibonacci heap to implement the open list instead of a binary heap/self-balancing tree, then the performance will become better (as Fibonacci heap takes O(1) average time to insert into open list and to decrease key)

Also to reduce the time taken to calculate g, we will use dynamic programming.

C++

```
// A C++ Program to implement A* Search Algorithm
#include <bits/stdc++.h>
using namespace std;

#define ROW 9
#define COL 10

// Creating a shortcut for int, int pair type
```

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

// A structure to hold the necessary 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
```

```
}

// 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");
```

```
    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;

    /*
```

```
 Note that 0 <= i <= ROW-1 & 0 <= j <= COL-1
 This open list is implemented 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 8 successor of this cell

         N.W   N   N.E
           \   |   /
            \  |  /
         W----Cell----E
              / | \
            /   |  \
         S.W    S   S.E

     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)
     N.E--> North-East  (i-1, j+1)
     N.W--> North-West  (i-1, j-1)
     S.E--> South-East  (i+1, j+1)
     S.W--> South-West  (i+1, j-1)*/
```

```
        // 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 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 - 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;
                }
            }
```

```
// 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 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 + 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;
        }
    }
}
```

```
            // 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 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;
                }
            }
        }
```

```
        // 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 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;
            }
        }
    }

    // 5th Successor (North-East)
```

```
    if (isValid(i - 1, j + 1) == true) {
        // If the destination cell is the same as the
        // current successor
        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;
        }

        // 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 + 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 it isn't on the open list, add it to
            // the 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 - 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;
            }
        }
    }
```

```
        // Only process this cell if this is a valid one
        if (isValid(i - 1, j - 1) == true) {
            // If the destination cell is the same as the
            // current successor
            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;
            }

            // 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 - 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 it isn’t on the open list, add it to
                // the 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 - 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;
                }
            }
        }
```

```
        // Only process this cell if this is a valid one
        if (isValid(i + 1, j + 1) == true) {
            // If the destination cell is the same as the
            // current successor
            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;
            }

            // 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 + 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 it isn't on the open list, add it to
                // the 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 + 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;
                }
            }
```

```
// Only process this cell if this is a valid one
if (isValid(i + 1, j - 1) == true) {
    // If the destination cell is the same as the
    // current successor
    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;
    }

    // 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 - 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 it isn't on the open list, add it to
        // the 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 + 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;
        }
```

```
    // When the destination cell is not found and the open
    // list is empty, then we conclude that we failed to
    // reach the destination cell. This may happen when the
    // there is no way to destination cell (due to
    // blockages)
    if (foundDest == false)
        printf("Failed to find the Destination Cell\n");

    return;
}

// Driver program to test above function
int main()
{
    /* Description of the Grid-
     1--> The cell is not blocked
     0--> The cell is blocked    */
    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);
}
```

Java

```
import java.util.HashMap;
import java.util.Map;
import java.util.LinkedHashMap;
```

```
class Cell {
    int parent_i, parent_j;
    double f, g, h;

    Cell() {
        this.parent_i = 0;
        this.parent_j = 0;
        this.f = 0;
        this.g = 0;
        this.h = 0;
    }
}

public class AStarSearch {

    private static final int ROW = 9;
    private static final int COL = 10;

    public static void main(String[] args) {
        // Description of the Grid-
        // 1--> The cell is not blocked
        // 0--> The cell is blocked
        int[][] grid = {
                {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
        int[] src = {8, 0};

        // Destination is the left-most top-most corner
        int[] dest = {0, 0};

        aStarSearch(grid, src, dest);
    }

    private static boolean isValid(int row, int col) {
        return (row >= 0) && (row < ROW) && (col >= 0) && (col < COL);
    }
```

```
    }

    private static boolean isDestination(int row, int col, int[] dest) {
        return row == dest[0] && col == dest[1];
    }

    private static double calculateHValue(int row, int col, int[] dest) {
        return Math.sqrt((row - dest[0]) * (row - dest[0]) + (col - dest[1]) * (cc
    }

    private static void tracePath(Cell[][] cellDetails, int[] dest) {
        System.out.println("The Path is ");
        int row = dest[0];
        int col = dest[1];

        Map<int[], Boolean> path = new LinkedHashMap<>();

        while (!(cellDetails[row][col].parent_i == row && cellDetails[row][col].pa
            path.put(new int[]{row, col}, true);
            int temp_row = cellDetails[row][col].parent_i;
            int temp_col = cellDetails[row][col].parent_j;
            row = temp_row;
            col = temp_col;
        }

        path.put(new int[]{row, col}, true);
        List<int[]> pathList = new ArrayList<>(path.keySet());
        Collections.reverse(pathList);

        pathList.forEach(p -> {
            if (p[0] == 2 || p[0] == 1) {
                System.out.print("-> (" + p[0] + ", " + (p[1]) + ")");
            } else {
                System.out.print("-> (" + p[0] + ", " + p[1] + ")");
            }
        });
        System.out.println();
    }

    private static void aStarSearch(int[][] grid, int[] src, int[] dest) {
        if (!isValid(src[0], src[1]) || !isValid(dest[0], dest[1])) {
            System.out.println("Source or destination is invalid");
            return;
        }
```

```
if (isDestination(src[0], src[1], dest)) {
    System.out.println("We are already at the destination");
    return;
}

boolean[][] closedList = new boolean[ROW][COL];
Cell[][] cellDetails = new Cell[ROW][COL];

for (int i = 0; i < ROW; i++) {
    for (int j = 0; j < COL; j++) {
        cellDetails[i][j] = new Cell();
        cellDetails[i][j].f = Double.POSITIVE_INFINITY;
        cellDetails[i][j].g = Double.POSITIVE_INFINITY;
        cellDetails[i][j].h = Double.POSITIVE_INFINITY;
        cellDetails[i][j].parent_i = -1;
        cellDetails[i][j].parent_j = -1;
    }
}

int i = src[0], j = src[1];
cellDetails[i][j].f = 0;
cellDetails[i][j].g = 0;
cellDetails[i][j].h = 0;
cellDetails[i][j].parent_i = i;
cellDetails[i][j].parent_j = j;

Map<Double, int[]> openList = new HashMap<>();
openList.put(0.0, new int[]{i, j});

boolean foundDest = false;

while (!openList.isEmpty()) {
    Map.Entry<Double, int[]> p = openList.entrySet().iterator().next();
    openList.remove(p.getKey());

    i = p.getValue()[0];
    j = p.getValue()[1];
    closedList[i][j] = true;

    double gNew, hNew, fNew;

    // 1st Successor (North)
    if (isValid(i - 1, j)) {
        if (isDestination(i - 1, j, dest)) {
```

```
                    foundDest = true;
                    return;
                } else if (!closedList[i - 1][j] && isUnBlocked(grid, i - 1, j)) {
                    gNew = cellDetails[i][j].g + 1;
                    hNew = calculateHValue(i - 1, j, dest);
                    fNew = gNew + hNew;

                    if (cellDetails[i - 1][j].f == Double.POSITIVE_INFINITY

|| cellDetails[i - 1][j].f > fNew) {
                        openList.put(fNew, new int[]{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;
                    }
                }
            }

            // 2nd Successor (South)
            if (isValid(i + 1, j)) {
                if (isDestination(i + 1, j, dest)) {
                    cellDetails[i + 1][j].parent_i = i;
                    cellDetails[i + 1][j].parent_j = j;
                    System.out.println("The destination cell is found");
                    tracePath(cellDetails, dest);
                    foundDest = true;
                    return;
                } else if (!closedList[i + 1][j] && isUnBlocked(grid, i + 1, j)) {
                    gNew = cellDetails[i][j].g + 1;
                    hNew = calculateHValue(i + 1, j, dest);
                    fNew = gNew + hNew;

                    if (cellDetails[i + 1][j].f == Double.POSITIVE_INFINITY || cel
                        openList.put(fNew, new int[]{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 (isDestination(i, j + 1, dest)) {
            cellDetails[i][j + 1].parent_i = i;
            cellDetails[i][j + 1].parent_j = j;
            System.out.println("The destination cell is found");
            tracePath(cellDetails, dest);
            foundDest = true;
            return;
        } else if (!closedList[i][j + 1] && isUnBlocked(grid, i, j + 1)) {
            gNew = cellDetails[i][j].g + 1;
            hNew = calculateHValue(i, j + 1, dest);
            fNew = gNew + hNew;

            if (cellDetails[i][j + 1].f == Double.POSITIVE_INFINITY || cel
                openList.put(fNew, new int[]{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;
            }
        }
    }

    // 4th Successor (West)
    if (isValid(i, j - 1)) {
        if (isDestination(i, j - 1, dest)) {
            cellDetails[i][j - 1].parent_i = i;
            cellDetails[i][j - 1].parent_j = j;
            System.out.println("The destination cell is found");
            tracePath(cellDetails, dest);
            foundDest = true;
            return;
        } else if (!closedList[i][j - 1] && isUnBlocked(grid, i, j - 1)) {
            gNew = cellDetails[i][j].g + 1;
            hNew = calculateHValue(i, j - 1, dest);
            fNew = gNew + hNew;

            if (cellDetails[i][j - 1].f == Double.POSITIVE_INFINITY || cel
                openList.put(fNew, new int[]{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;
```

```
        // 5th Successor (North-East)
        if (isValid(i - 1, j + 1)) {
            if (isDestination(i - 1, j + 1, dest)) {
                cellDetails[i - 1][j + 1].parent_i = i;
                cellDetails[i - 1][j + 1].parent_j = j;
                System.out.println("The destination cell is found");
                tracePath(cellDetails, dest);
                foundDest = true;
                return;
            } else if (!closedList[i - 1][j + 1] && isUnBlocked(grid, i - 1, j
                gNew = cellDetails[i][j].g + 1.414;
                hNew = calculateHValue(i - 1, j + 1, dest);
                fNew = gNew + hNew;

                if (cellDetails[i - 1][j + 1].f == Double.POSITIVE_INFINITY ||
                    openList.put(fNew, new int[]{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;
                }
            }
        }

        // 6th Successor (North-West)
        if (isValid(i - 1, j - 1)) {
            if (isDestination(i - 1, j - 1, dest)) {
                cellDetails[i - 1][j - 1].parent_i = i;
                cellDetails[i - 1][j - 1].parent_j = j;
                System.out.println("The destination cell is found");
                tracePath(cellDetails, dest);
                foundDest = true;
                return;
            } else if (!closedList[i - 1][j - 1] && isUnBlocked(grid, i - 1, j
                gNew = cellDetails[i][j].g + 1.414;
                hNew = calculateHValue(i - 1, j - 1, dest);
                fNew = gNew + hNew;

                if                          (cellDetails[i - 1][j - 1].f == Doub
                    openList.put(fNew, new int[]{i - 1, j - 1});

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

```
            }
        }
    }

    // 7th Successor (South-East)
    if (isValid(i + 1, j + 1)) {
        if (isDestination(i + 1, j + 1, dest)) {
            cellDetails[i + 1][j + 1].parent_i = i;
            cellDetails[i + 1][j + 1].parent_j = j;
            System.out.println("The destination cell is found");
            tracePath(cellDetails, dest);
            foundDest = true;
            return;
        } else if (!closedList[i + 1][j + 1] && isUnBlocked(grid, i + 1, j
            gNew = cellDetails[i][j].g + 1.414;
            hNew = calculateHValue(i + 1, j + 1, dest);
            fNew = gNew + hNew;

            if (cellDetails[i + 1][j + 1].f == Double.POSITIVE_INFINITY ||
                openList.put(fNew, new int[]{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;
            }
        }
    }

    // 8th Successor (South-West)
    if (isValid(i + 1, j - 1)) {
        if (isDestination(i + 1, j - 1, dest)) {
            cellDetails[i + 1][j - 1].parent_i = i;
            cellDetails[i + 1][j - 1].parent_j = j;
            System.out.println("The destination cell is found");
            tracePath(cellDetails, dest);
            foundDest = true;
            return;
        } else if (!closedList[i + 1][j - 1] && isUnBlocked(grid, i + 1, j
            gNew = cellDetails[i][j].g + 1.414;
            hNew = calculateHValue(i + 1, j - 1, dest);
            fNew = gNew + hNew;

            if (cellDetails[i + 1][j - 1].f == Double.POSITIVE_INFINITY ||
```

```
                        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)
            System.out.println("Failed to find the destination cell");
    }
}
```

C#

```
using System;
using System.Collections.Generic;

public class AStarSearch
{
    // Creating a shortcut for KeyValuePair<int, int>
    public struct Pair
    {
        public int first, second;

        public Pair(int x, int y)
        {
            first = x;
            second = y;
        }
    }

    // A structure to hold the necessary parameters
    public struct Cell
    {
        // Row and Column index of its parent
        // Note that 0 <= i <= ROW-1 & 0 <= j <= COL-1
        public int parent_i, parent_j;
        // f = g + h
        public double f, g, h;
    }

    // A Function to find the shortest path between
```

```
int ROW = grid.GetLength(0);
int COL = grid.GetLength(1);

// If the source or destination is out of range
if (!IsValid(src.first, src.second, ROW, COL) || !IsValid(dest.first, dest
{
    Console.WriteLine("Source or destination is invalid");
    return;
}

// Either the source or the destination is blocked
if (!IsUnBlocked(grid, src.first, src.second) || !IsUnBlocked(grid, dest.f
{
    Console.WriteLine("Source or the destination is blocked");
    return;
}

// If the destination cell is the same as the source cell
if (src.first == dest.first && src.second == dest.second)
{
    Console.WriteLine("We are already at the destination");
    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 = new bool[ROW, COL];

// Declare a 2D array of structure to hold the details
// of that cell
Cell[,] cellDetails = new Cell[ROW, COL];

for (int i = 0; i < ROW; i++)
{
    for (int j = 0; j < COL; j++)
    {
        cellDetails[i, j].f = double.MaxValue;
        cellDetails[i, j].g = double.MaxValue;
        cellDetails[i, j].h = double.MaxValue;
        cellDetails[i, j].parent_i = -1;
        cellDetails[i, j].parent_j = -1;
    }
}

// Initialising the parameters of the starting node
```

```
cellDetails[x, y].h = 0.0;
cellDetails[x, y].parent_i = x;
cellDetails[x, y].parent_j = y;

/*
    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 implemented as a SortedSet of tuple (f, (i, j)).
    We use a custom comparer to compare tuples based on their f values.
*/
SortedSet<(double, Pair)> openList = new SortedSet<(double, Pair)>(
    Comparer<(double, Pair)>.Create((a, b) => a.Item1.CompareTo(b.Item1)))

// Put the starting cell on the open list and set its
// 'f' as 0
openList.Add((0.0, new Pair(x, y)));

// We set this boolean value as false as initially
// the destination is not reached.
bool foundDest = false;

while (openList.Count > 0)
{
    (double f, Pair pair) p = openList.Min;
    openList.Remove(p);

    // Add this vertex to the closed list
    x = p.pair.first;
    y = p.pair.second;
    closedList[x, y] = true;

    // Generating all the 8 successors of this cell
    for (int i = -1; i <= 1; i++)
    {
        for (int j = -1; j <= 1; j++)
        {
            if (i == 0 && j == 0)
                continue;

            int newX = x + i;
            int newY = y + j;

            // If this successor is a valid cell
```

```
                    // current successor
                    if (IsDestination(newX, newY, dest))
                    {
                        cellDetails[newX, newY].parent_i = x;
                        cellDetails[newX, newY].parent_j = y;
                        Console.WriteLine("The destination cell is found");
                        TracePath(cellDetails, dest);
                        foundDest = true;
                        return;
                    }

                    // If the successor is already on the closed
                    // list or if it is blocked, then ignore it.
                    if (!closedList[newX, newY] && IsUnBlocked(grid, newX, new
                    {
                        double gNew = cellDetails[x, y].g + 1.0;
                        double hNew = CalculateHValue(newX, newY, dest);
                        double 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 the
                        // f, g, and h costs of the square cell
                        if (cellDetails[newX, newY].f == double.MaxValue || ce
                        {
                            openList.Add((fNew, new Pair(newX, newY)));

                            // Update the details of this cell
                            cellDetails[newX, newY].f = fNew;
                            cellDetails[newX, newY].g = gNew;
                            cellDetails[newX, newY].h = hNew;
                            cellDetails[newX, newY].parent_i = x;
                            cellDetails[newX, newY].parent_j = y;
                        }
                    }
                }
            }
        }
    }

    // When the destination cell is not found and the open
    // list is empty, then we conclude that we failed to
    // reach the destination cell. This may happen when the
    // there is no way to destination cell (due to
    // blockages)
    if (!foundDest)
```

```
// A Utility Function to check whether given cell (row, col)
// is a valid cell or not.
public static bool IsValid(int row, int col, 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
public static bool IsUnBlocked(int[,] grid, int row, int col)
{
    // Returns true if the cell is not blocked else false
    return grid[row, col] == 1;
}

// A Utility Function to check whether destination cell has
// been reached or not
public static bool IsDestination(int row, int col, Pair dest)
{
    return (row == dest.first && col == dest.second);
}

// A Utility Function to calculate the 'h' heuristics.
public static double CalculateHValue(int row, int col, Pair dest)
{
    // Return using the distance formula
    return Math.Sqrt(Math.Pow(row - dest.first, 2) + Math.Pow(col - dest.secor
}

// A Utility Function to trace the path from the source
// to destination
public static void TracePath(Cell[,] cellDetails, Pair dest)
{
    Console.WriteLine("\nThe Path is ");
    int ROW = cellDetails.GetLength(0);
    int COL = cellDetails.GetLength(1);

    int row = dest.first;
    int col = dest.second;

    Stack<Pair> Path = new Stack<Pair>();

    while (!(cellDetails[row, col].parent_i == row && cellDetails[row, col].pa
```

```
            row = temp_row;
            col = temp_col;
        }

        Path.Push(new Pair(row, col));
        while (Path.Count > 0)
        {
            Pair p = Path.Peek();
            Path.Pop();
            Console.Write(" -> ({0},{1}) ", p.first, p.second);
        }
    }

    // Driver method
    public static void Main(string[] args)
    {
        /* Description of the Grid-
            1--> The cell is not blocked
            0--> The cell is blocked */
        int[,] grid =
        {
            {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 = new Pair(8, 0);

        // Destination is the left-most top-most corner
        Pair dest = new Pair(0, 0);

        AStar(grid, src, dest);
    }
}
```

Javascript

```
let COL = 10;

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

// A structure to hold the necessary parameters
class cell {
    // Row and Column index of its parent
    // Note that 0 <= i <= ROW-1 & 0 <= j <= COL-1
    constructor(){
        this.parent_i = 0;
        this.parent_j = 0;
        this.f = 0;
        this.g = 0;
        this.h = 0;
    }
}

// A Utility Function to check whether given cell (row, col)
// is a valid cell or not.
function isValid(row, 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
function isUnBlocked(grid, row, 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
function isDestination(row, col, dest)
{
    if (row == dest[0] && col == dest[1])
        return (true);
    else
        return (false);
}
```

```
{
    // Return using the distance formula
    return (Math.sqrt((row - dest[0]) * (row - dest[0]) + (col - dest[1]) * (col -
}

// A Utility Function to trace the path from the source
// to destination
function tracePath(cellDetails, dest)
{
    console.log("The Path is ");
    let row = dest[0];
    let col = dest[1];

    // stack<Pair> Path;
    let Path = [];

    while (!(cellDetails[row][col].parent_i == row && cellDetails[row][col].parent
        Path.push([row, col]);
        let temp_row = cellDetails[row][col].parent_i;
        let temp_col = cellDetails[row][col].parent_j;
        row = temp_row;
        col = temp_col;
    }

    Path.push([row, col]);
    while (Path.length > 0) {
        let p = Path[0];
        Path.shift();

        if(p[0] == 2 || p[0] == 1){
            console.log("-> (" + p[0] + ", " + (p[1] - 1) + ")");
        }
        else console.log("-> (" + p[0] + ", " + p[1] + ")");
    }

    return;
}

// A Function to find the shortest path between
// a given source cell to a destination cell according
// to A* Search Algorithm
function aStarSearch(grid, src, dest)
{
    // If the source is out of range
    if (isValid(src[0], src[1]) == false) {
```

```
// If the destination is out of range
if (isValid(dest[0], dest[1]) == false) {
    console.log("Destination is invalid\n");
    return;
}

// Either the source or the destination is blocked
if (isUnBlocked(grid, src[0], src[1]) == false
    || isUnBlocked(grid, dest[0], dest[1])
           == false) {
    console.log("Source or the destination is blocked\n");
    return;
}

// If the destination cell is the same as source cell
if (isDestination(src[0], src[1], dest)
    == true) {
    console.log("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
let closedList = new Array(ROW);
for(let i = 0; i < ROW; i++){
    closedList[i] = new Array(COL).fill(false);
}

// Declare a 2D array of structure to hold the details
// of that cell
let cellDetails = new Array(ROW);
for(let i = 0; i < ROW; i++){
    cellDetails[i] = new Array(COL);
}

let i, j;

for (i = 0; i < ROW; i++) {
    for (j = 0; j < COL; j++) {
        cellDetails[i][j] = new cell();
        cellDetails[i][j].f = 2147483647;
        cellDetails[i][j].g = 2147483647;
        cellDetails[i][j].h = 2147483647;
        cellDetails[i][j].parent_i = -1;
```

```
// Initialising the parameters of the starting node
i = src[0], j = src[1];
cellDetails[i][j].f = 0;
cellDetails[i][j].g = 0;
cellDetails[i][j].h = 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 implemented as a set of pair of
 pair.*/
let openList = new Map();

// Put the starting cell on the open list and set its
// 'f' as 0
openList.set(0, [i, j]);

// We set this boolean value as false as initially
// the destination is not reached.
let foundDest = false;

while (openList.size > 0) {
    let p = openList.entries().next().value

    // Remove this vertex from the open list
    openList.delete(p[0]);

    // Add this vertex to the closed list
    i = p[1][0];
    j = p[1][1];
    closedList[i][j] = true;

    /*
     Generating all the 8 successor of this cell

         N.W   N   N.E
           \   |   /
            \  |  /
         W----Cell----E
              / | \
```

```
 N -->  North       (i-1, j)
 S -->  South       (i+1, j)
 E -->  East        (i, j+1)
 W -->  West           (i, j-1)
 N.E--> North-East  (i-1, j+1)
 N.W--> North-West  (i-1, j-1)
 S.E--> South-East  (i+1, j+1)
 S.W--> South-West  (i+1, j-1)*/

// To store the 'g', 'h' and 'f' of the 8 successors
let 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;
        console.log("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;
        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 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.
```

```
                // 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;
            console.log("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;
            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 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 + 1][j].f == 2147483647
                || cellDetails[i + 1][j].f > fNew) {
```

```
                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;
            console.log("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;
            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 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 == 2147483647
                || cellDetails[i][j + 1].f > fNew) {
                openList.set(fNew, [i, j + 1]);

                // Update the details of this cell
```

```
                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;
            console.log("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;
            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 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 == 2147483647
                || cellDetails[i][j - 1].f > fNew) {
                openList.set(fNew, [i, j - 1]);

                // Update the details of this cell
```

```
                cellDetails[i][j - 1].parent_j = j;
            }
        }
    }

    //----------- 5th Successor (North-East)
    //------------

    // Only process this cell if this is a valid one
    if (isValid(i - 1, j + 1) == true) {
        // If the destination cell is the same as the
        // current successor
        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;
            console.log("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 + 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 it isn’t on the open list, add it to
            // the 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 - 1][j + 1].f == 2147483647
                || cellDetails[i - 1][j + 1].f > fNew) {
                openList.set(fNew, [i - 1, j + 1]);

                // Update the details of this cell
```

```
                cellDetails[i - 1][j + 1].parent_j = j;
            }
        }
    }

    //----------- 6th Successor (North-West)
    //------------

    // Only process this cell if this is a valid one
    if (isValid(i - 1, j - 1) == true) {
        // If the destination cell is the same as the
        // current successor
        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;
            console.log("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 - 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 it isn't on the open list, add it to
            // the 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 - 1][j - 1].f == 2147483647
                || cellDetails[i - 1][j - 1].f > fNew) {
                openList.set(fNew, [i - 1, j - 1]);
                // Update the details of this cell
                cellDetails[i - 1][j - 1].f = fNew;
```

```
            }
        }
    }

    //----------- 7th Successor (South-East)
    //------------

    // Only process this cell if this is a valid one
    if (isValid(i + 1, j + 1) == true) {
        // If the destination cell is the same as the
        // current successor
        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;
            console.log("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 + 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 it isn't on the open list, add it to
            // the 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 + 1][j + 1].f == 2147483647
                || cellDetails[i + 1][j + 1].f > fNew) {
                openList.set(fNew, [i + 1, j + 1]);

                // Update the details of this cell
                cellDetails[i + 1][j + 1].f = fNew;
```

```
            }
        }
    }

    //----------- 8th Successor (South-West)
    //------------

    // Only process this cell if this is a valid one
    if (isValid(i + 1, j - 1) == true) {
        // If the destination cell is the same as the
        // current successor
        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;
            console.log("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 - 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 it isn't on the open list, add it to
            // the 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 + 1][j - 1].f == FLT_MAX
                || cellDetails[i + 1][j - 1].f > fNew) {
                openList.set(fNew, [i + 1, j - 1]);

                // Update the details of this cell
                cellDetails[i + 1][j - 1].f = fNew;
```

```
                }
            }
        }
    }

    // When the destination cell is not found and the open
    // list is empty, then we conclude that we failed to
    // reach the destination cell. This may happen when the
    // there is no way to destination cell (due to
    // blockages)
    if (foundDest == false)
        console.log("Failed to find the Destination Cell\n");

    return;
}

// Driver program to test above function
/* Description of the Grid-
 1--> The cell is not blocked
 0--> The cell is blocked    */
let grid = [[ 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
let src = [8, 0];

// Destination is the left-most top-most corner
let dest = [0, 0];

aStarSearch(grid, src, dest);

// The code is contributed by Nidhi goel.
```

Python3

```
import math
```

```
    def __init__(self):
        self.parent_i = 0  # Parent cell's row index
        self.parent_j = 0  # Parent cell's column index
        self.f = float('inf')  # Total cost of the cell (g + h)
        self.g = float('inf')  # Cost from start to this cell
        self.h = 0  # Heuristic cost from this cell to destination

# Define the size of the grid
ROW = 9
COL = 10

# Check if a cell is valid (within the grid)
def is_valid(row, col):
    return (row >= 0) and (row < ROW) and (col >= 0) and (col < COL)

# Check if a cell is unblocked
def is_unblocked(grid, row, col):
    return grid[row][col] == 1

# Check if a cell is the destination
def is_destination(row, col, dest):
    return row == dest[0] and col == dest[1]

# Calculate the heuristic value of a cell (Euclidean distance to destination)
def calculate_h_value(row, col, dest):
    return ((row - dest[0]) ** 2 + (col - dest[1]) ** 2) ** 0.5

# Trace the path from source to destination
def trace_path(cell_details, dest):
    print("The Path is ")
    path = []
    row = dest[0]
    col = dest[1]

    # Trace the path from destination to source using parent cells
    while not (cell_details[row][col].parent_i == row and cell_details[row][col].
        path.append((row, col))
        temp_row = cell_details[row][col].parent_i
        temp_col = cell_details[row][col].parent_j
        row = temp_row
        col = temp_col

    # Add the source cell to the path
    path.append((row, col))
    # Reverse the path to get the path from source to destination
```

```
            print("->", i, end=" ")
        print()

    # Implement the A* search algorithm
    def a_star_search(grid, src, dest):
        # Check if the source and destination are valid
        if not is_valid(src[0], src[1]) or not is_valid(dest[0], dest[1]):
            print("Source or destination is invalid")
            return

        # Check if the source and destination are unblocked
        if not is_unblocked(grid, src[0], src[1]) or not is_unblocked(grid, dest[0],
            print("Source or the destination is blocked")
            return

        # Check if we are already at the destination
        if is_destination(src[0], src[1], dest):
            print("We are already at the destination")
            return

        # Initialize the closed list (visited cells)
        closed_list = [[False for _ in range(COL)] for _ in range(ROW)]
        # Initialize the details of each cell
        cell_details = [[Cell() for _ in range(COL)] for _ in range(ROW)]

        # Initialize the start cell details
        i = src[0]
        j = src[1]
        cell_details[i][j].f = 0
        cell_details[i][j].g = 0
        cell_details[i][j].h = 0
        cell_details[i][j].parent_i = i
        cell_details[i][j].parent_j = j

        # Initialize the open list (cells to be visited) with the start cell
        open_list = []
        heapq.heappush(open_list, (0.0, i, j))

        # Initialize the flag for whether destination is found
        found_dest = False

        # Main loop of A* search algorithm
        while len(open_list) > 0:
            # Pop the cell with the smallest f value from the open list
            p = heapq.heappop(open_list)
```

```
            closed_list[i][j] = True

            # For each direction, check the successors
            directions = [(0, 1), (0, -1), (1, 0), (-1, 0), (1, 1), (1, -1), (-1, 1),
            for dir in directions:
                new_i = i + dir[0]
                new_j = j + dir[1]

                # If the successor is valid, unblocked, and not visited
                if is_valid(new_i, new_j) and is_unblocked(grid, new_i, new_j) and no
                    # If the successor is the destination
                    if is_destination(new_i, new_j, dest):
                        # Set the parent of the destination cell
                        cell_details[new_i][new_j].parent_i = i
                        cell_details[new_i][new_j].parent_j = j
                        print("The destination cell is found")
                        # Trace and print the path from source to destination
                        trace_path(cell_details, dest)
                        found_dest = True
                        return
                    else:
                        # Calculate the new f, g, and h values
                        g_new = cell_details[i][j].g + 1.0
                        h_new = calculate_h_value(new_i, new_j, dest)
                        f_new = g_new + h_new

                        # If the cell is not in the open list or the new f value is sm
                        if cell_details[new_i][new_j].f == float('inf') or cell_detai
                            # Add the cell to the open list
                            heapq.heappush(open_list, (f_new, new_i, new_j))
                            # Update the cell details
                            cell_details[new_i][new_j].f = f_new
                            cell_details[new_i][new_j].g = g_new
                            cell_details[new_i][new_j].h = h_new
                            cell_details[new_i][new_j].parent_i = i
                            cell_details[new_i][new_j].parent_j = j

        # If the destination is not found after visiting all cells
        if not found_dest:
            print("Failed to find the destination cell")

    def main():
        # Define the grid (1 for unblocked, 0 for blocked)
        grid = [
            [1, 0, 1, 1, 1, 1, 0, 1, 1, 1],
```

```
        [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]
    ]

    # Define the source and destination
    src = [8, 0]
    dest = [0, 0]

    # Run the A* search algorithm
    a_star_search(grid, src, dest)

if __name__ == "__main__":
    main()
```

Limitations

Although being the best path finding algorithm around, A* Search Algorithm doesn't produce the shortest path always, as it relies heavily on heuristics / approximations to calculate – h

Applications

This is the most interesting part of A* Search Algorithm. They are used in games! But how?

Ever played Tower Defense Games ?

Tower defense is a type of strategy video game where the goal is to defend a player's territories or possessions by obstructing enemy attackers, usually achieved by placing defensive structures on or along their path of attack.

A* Search Algorithm is often used to find the shortest path from one point to another point. You can use this for each enemy to find a path to the goal.

One example of this is the very popular game- Warcraft III

What if the search space is not a grid and is a graph ?

The same rules applies there also. The example of grid is taken for the simplicity of understanding. So we can find the shortest path between the

Time Complexity

Considering a graph, it may take us to travel all the edge to reach the destination cell from the source cell [For example, consider a graph where source and destination nodes are connected by a series of edges, like – 0(source) –>1 –> 2 –> 3 (target)

So the worse case time complexity is O(E), where E is the number of edges in the graph

Auxiliary Space In the worse case we can have all the edges inside the open list, so required auxiliary space in worst case is O(V), where V is the total number of vertices.

Exercise to the Readers-

Ever wondered how to make a game like- Pacman where there are many such obstacles. Can we use A* Search Algorithm to find the correct way ?

Think about it as a fun exercise.

Articles for interested readers

In our program, the obstacles are fixed. What if the obstacles are moving ?

Interested readers may see here an excellent discussion on this topic.

Summary

So when to use BFS over A*, when to use Dijkstra over A* to find the shortest paths ?

We can summarise this as below-

1) One source and One Destination-

? Use A* Search Algorithm (For Unweighted as well as Weighted Graphs)

2) One Source, All Destination –

? Use BFS (For Unweighted Graphs)

? Use Dijkstra (For Weighted Graphs without negative weights)

? Use Bellman Ford (For Weighted Graphs with negative weights)

3) Between every pair of nodes-

? Floyd-Warshall

? Johnson's Algorithm

References-

http://theory.stanford.edu/~amitp/GameProgramming/

https://en.wikipedia.org/wiki/A*_search_algorithm

Last Updated : 07 Mar, 2024

Previous

Interval Tree using GNU Tree-based container

Next

Find paths from corner cell to middle cell in maze

Similar Reads

Iterative Deepening Search(IDS) or Iterative Deepening Depth First Search(IDDFS)

Meta Binary Search | One-Sided Binary Search

Anagram Substring Search (Or Search for all permutations) | Set 2

Is Sentinel Linear Search better than normal Linear Search?

Search N elements in an unbalanced Binary Search Tree in O(N * logM) time

Binary Search Tree vs Ternary Search Tree

Interpolation search vs Binary search

Linear Search vs Binary Search

Anagram Substring Search (Or Search for all permutations)

Why is Binary Search preferred over Ternary Search?

R Rachit Belwariar

Article Tags : Algorithms-Graph Traversals , Algorithms-InsertionSort , Intellipaat , Perl-Scalars , Python numpy-Random , QA - Placement Quizzes-Data Interpretation , TCS-coding-questions , Volkswagen IT Services , Algorithms , DSA

Practice Tags : Algorithms

Web Scraping

OpenCV Tutorial

Python Interview Question

DevOps

Git

AWS

Docker

Kubernetes

Azure

GCP

DevOps Roadmap

System Design

High Level Design

Low Level Design

UML Diagrams

Interview Guide

Design Patterns

OOAD

System Design Bootcamp

Interview Questions

Preparation Corner

Company-Wise Recruitment Process

Resume Templates

Aptitude Preparation

Puzzles

Company-Wise Preparation

Software Engineering

Digital Logic Design

Engineering Maths

Competitive Programming

Top DS or Algo for CP

Top 50 Tree

Top 50 Graph

Top 50 Array

Top 50 String

Top 50 DP

Top 15 Websites for CP

JavaScript

JavaScript Examples

TypeScript

ReactJS

NextJS

AngularJS

NodeJS

Lodash

Web Browser

School Subjects

Mathematics

Physics

Chemistry

Biology

Social Science

English Grammar

World GK

Management & Finance

Free Online Tools

Finance

Income Tax

Organisational Behaviour

Marketing

Code Formatters

Code Converters

Currency Converter

Random Number Generator

Random Password Generator

More Tutorials

Software Development

Software Testing

Product Management

SAP

SEO - Search Engine Optimization

Linux

Excel

GeeksforGeeks Videos

DSA

Python

Java

C++

Data Science

CS Subjects