Custom Search

Write an Article Login

How to write an Interview Experience?

Must Do Coding Questions Company-wise

Must Do Coding Questions Topic-wise

Difficulty Levels

Basic

Easy

Medium

Hard

Expert

Popular Tags

Amazon, Microsoft, Dynamic Programming, Samsung
Click here for more

Interview Preparation

Step by Step Preparation

Company Preparation

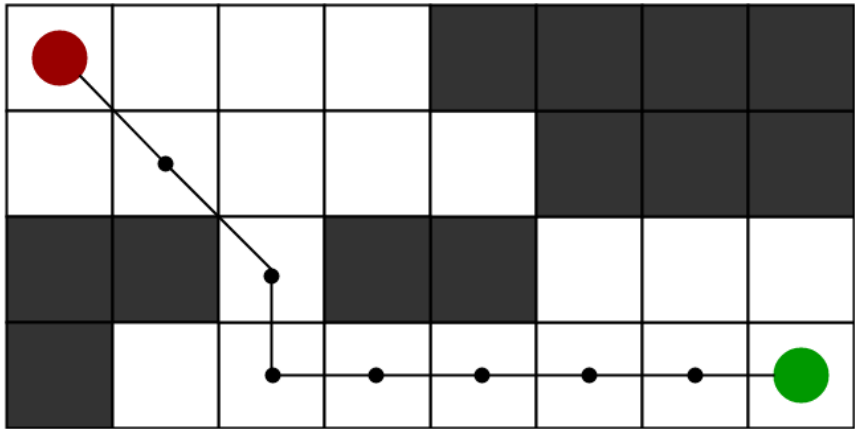
Top Topics

Software Design Patterns
Placements Preparation Course
Interview Corner
Recent Interview Experiences
GQ Home Page
Quiz Corner
LMNs
Practice Platform
What's New ?
Leaderboard !!
Topic-wise Practice
Subjective Problems
Difficulty Level - School
Difficulty Level - Basic
Difficulty Level - Easy
Difficulty Level - Medium
Difficulty Level - Hard
How to pick a difficulty level?
Explore More...

C
C++
Java
Python
SQL
Important Quick Links
School Programming
Operating Systems
DBMS
Computer Networks
Engineering Mathematics
Design Patterns
Common Interview Puzzles
Web Technology
G-Facts
Computer Graphics
Image Processing
Project Ideas

A* Search Algorithm

We can consider a 2D Grid having several obstacles and we start from a source cell (coloured red below) to reach towards a goal cell (coloured 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 detail in below sections.

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 t ther 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 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.

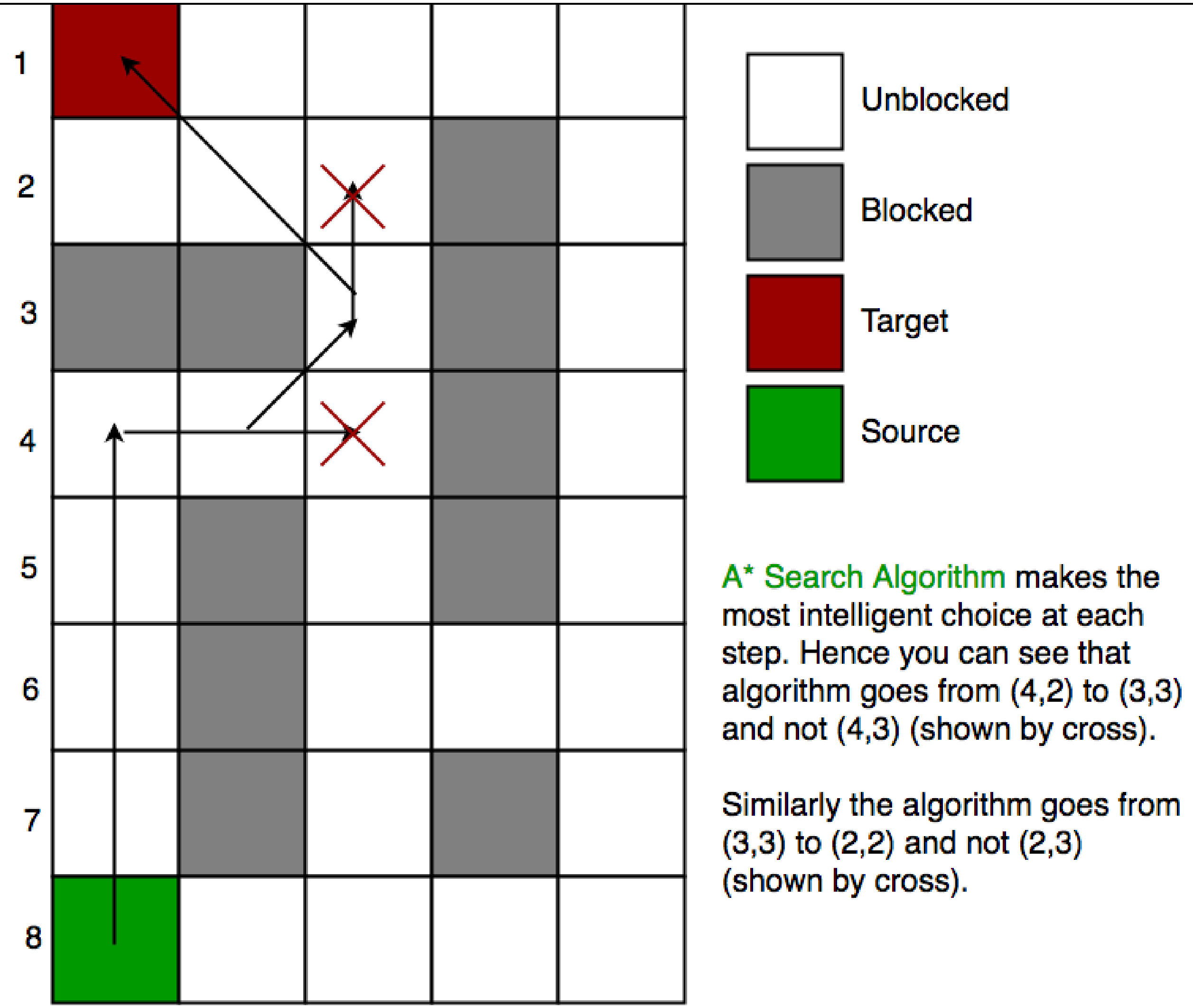
Algorithm

We create two lists – Open List and Closed List (just like Dijkstra Algorithm)

```
// 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
          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

        ii) if a node with the same position as
            successor is in the OPEN list which has a
```

```
            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)

    e) push q on the closed list
    end (while 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).

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 –

There are generally three approximation heuristics to calculate h –

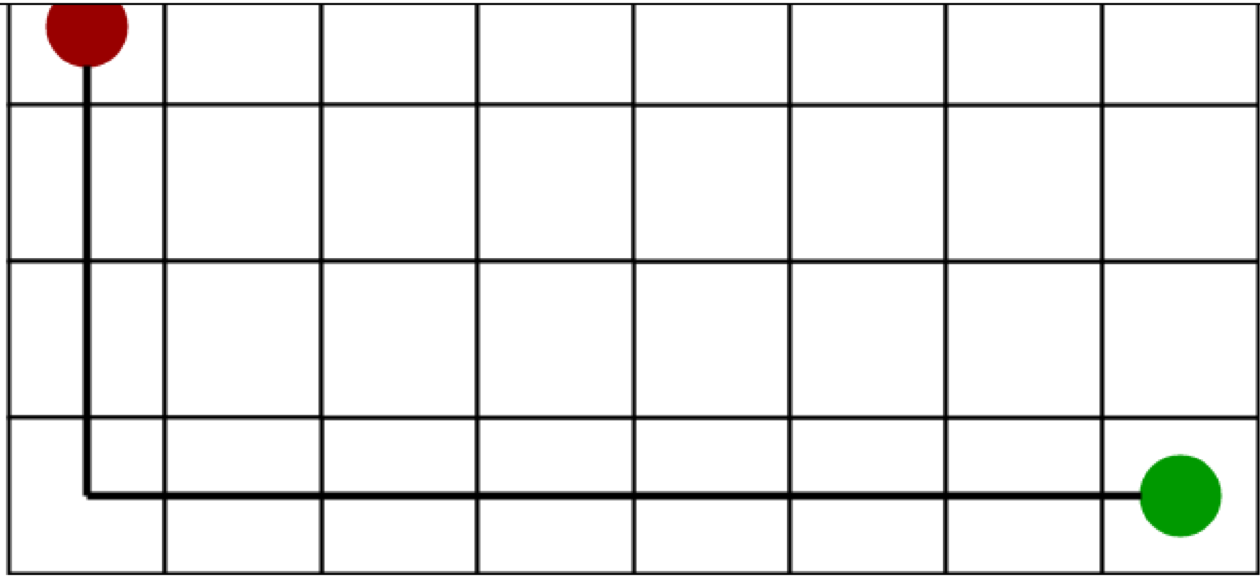
1) Manhattan Distance –

- 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).



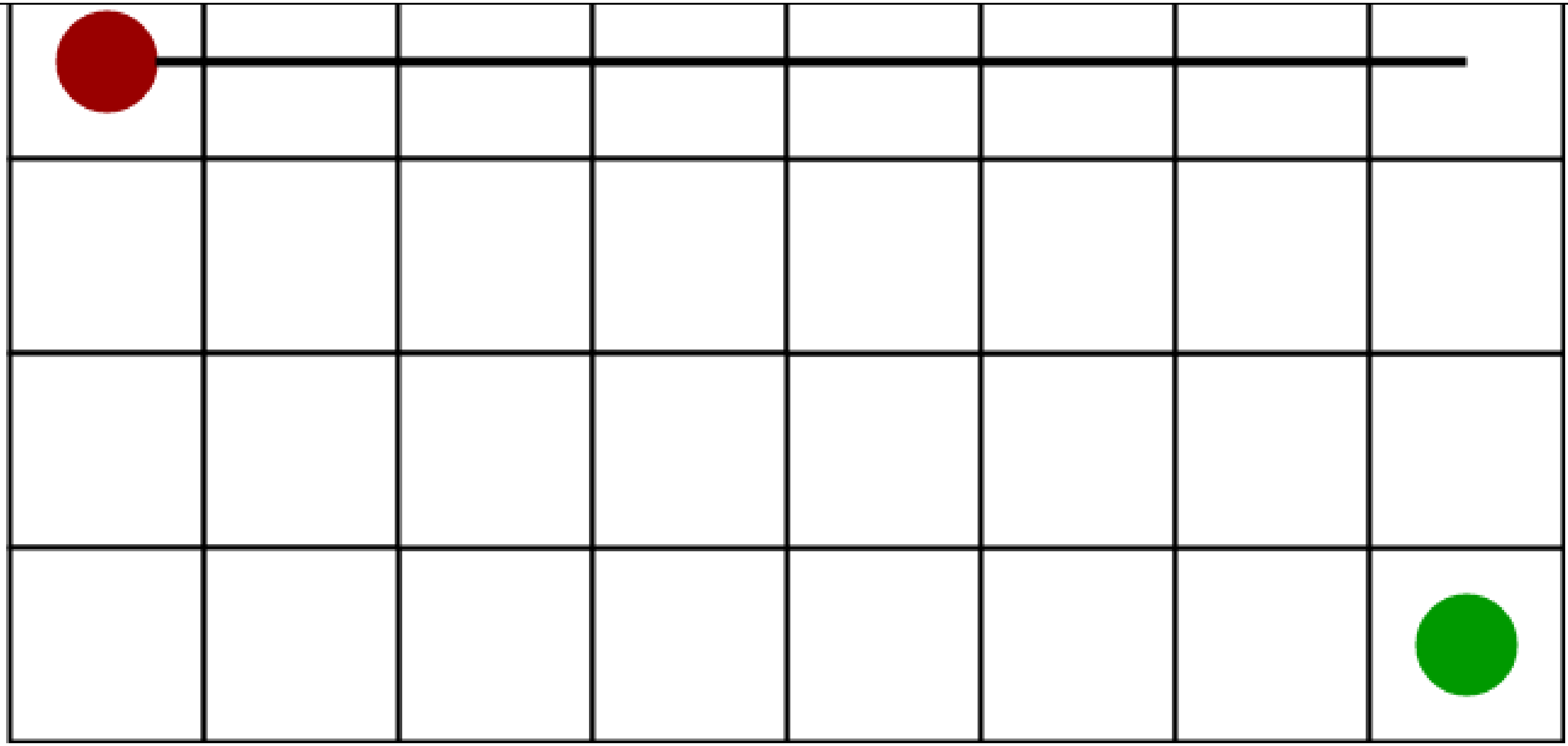
2) Diagonal Distance-

- 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.,

```
h = max { abs(current_cell.x – goal.x),
          abs(current_cell.y – goal.y) }
```

- 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).



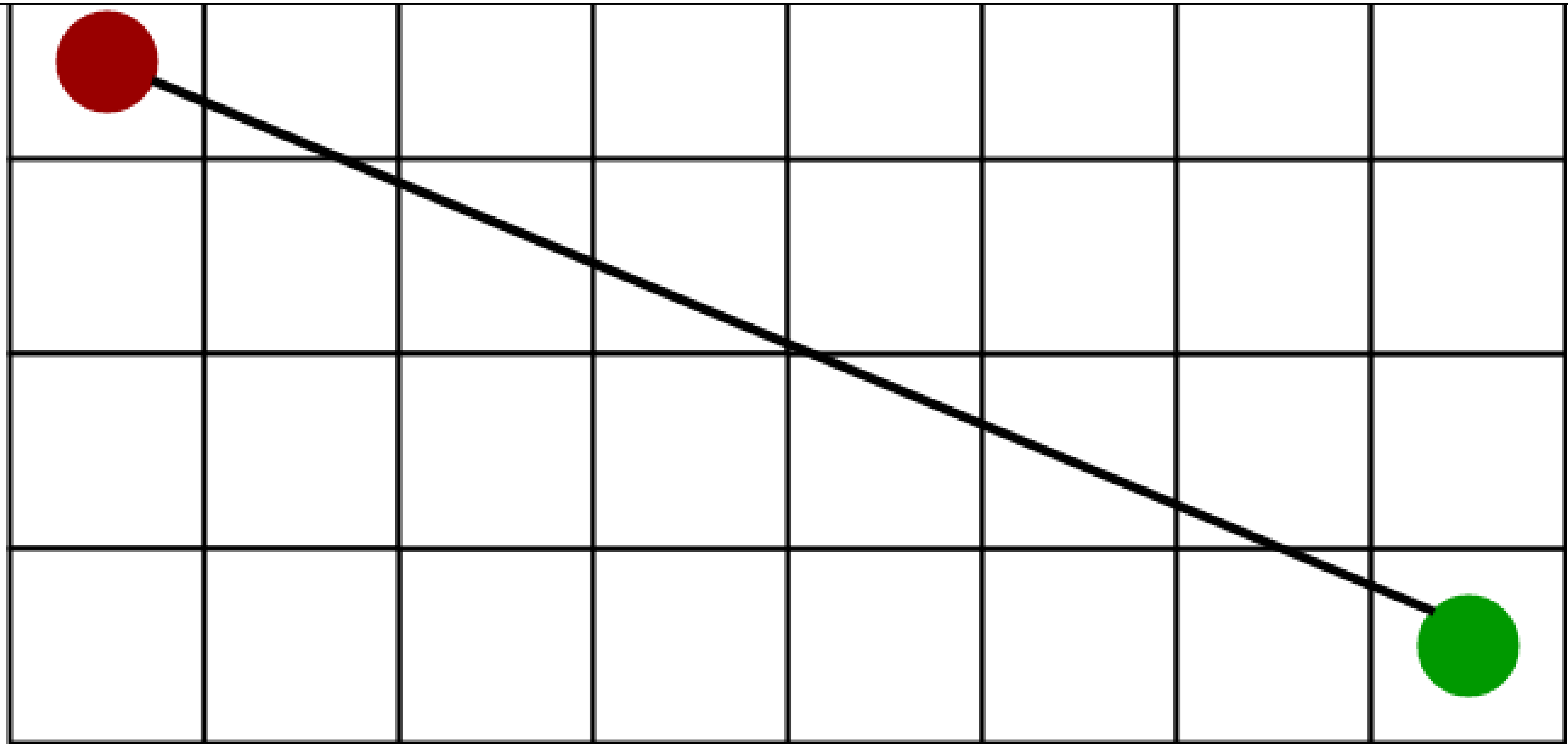
3) Euclidean Distance-

- 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 Dijsktra'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.

```
#define ROW 9
#define COL 10

// 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
```

```
// 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");
```

```
    // 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].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 open 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)
```

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

        // To store the 'g', 'h' and 'f' of the 8 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 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].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 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;
```

```
            }
        }

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

```
                }
            }
        }

        //----------- 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
                // 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].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;
                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].parent_i = i;
                    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;
                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].parent_j = j;
            }
        }
    }

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

    //----------- 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;
            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)));
```

```
                    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 destiantion 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);
}
```

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 source node and the target node in a graph using this A* Search Algorithm, just like we did for a 2D Grid.

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.

Summary

So when to use DFS 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

Related Article:

Best First Search (Informed Search)

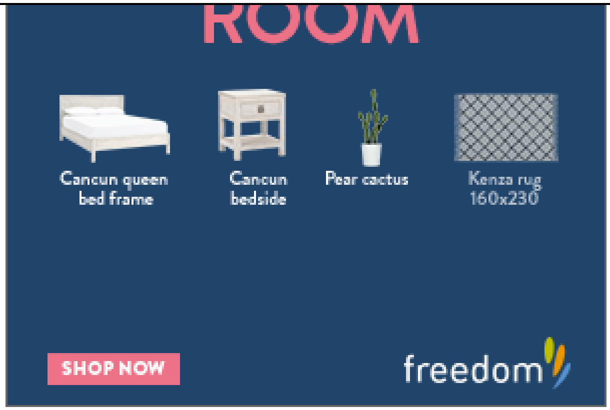
References-

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

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

This article is contributed by **Rachit Belwariar**. If you like GeeksforGeeks and would like to contribute, you can also write an article and mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above



GATE CS Corner **Company Wise Coding Practice**

Algorithm

Login to Improve this Article

Please write to us at contribute@geeksforgeeks.org to report any issue with the above content.

Recommended Posts:

The Two Water Jug Puzzle
Converting Roman Numerals to Decimal lying between 1 to 3999
Exact Cover Problem and Algorithm X | Set 2 (Implementation with DLX)
DDA Line generation Algorithm in Computer Graphics
Recursion
Implementation of K Nearest Neighbors
Program for class interval arithmetic mean
Program to find covariance
Operating System | Dekker's algorithm

(Login to Rate)

3.8 Average Difficulty : **3.8/5.0**
Based on **16** vote(s)

Basic Easy Medium Hard Expert

Add to TODO List

Mark as DONE

Writing code in comment? Please use ide.geeksforgeeks.org, generate link and share the link here.

Load Comments Share this post!

A computer science portal for geeks

710-B, Advant Navis Business Park,
Sector-142, Noida, Uttar Pradesh - 201305
feedback@geeksforgeeks.org