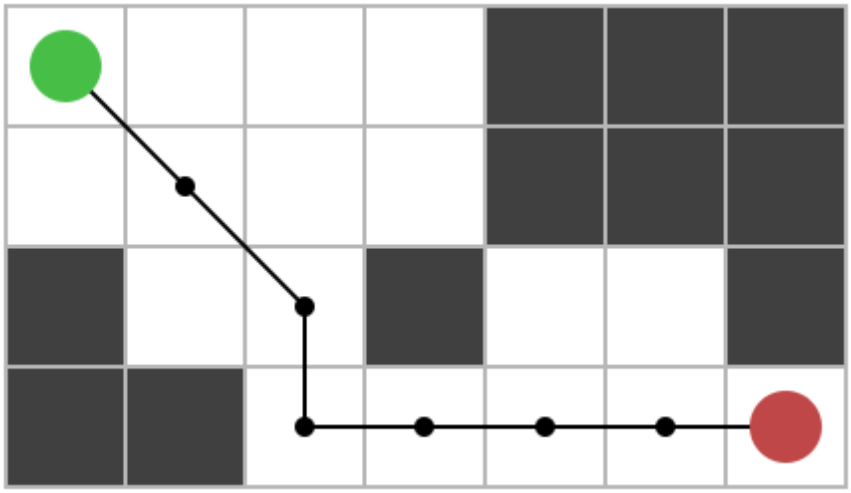
A* Search Algorithm

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 (coloured green below) to reach towards a goal cell (coloured red 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 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.

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)

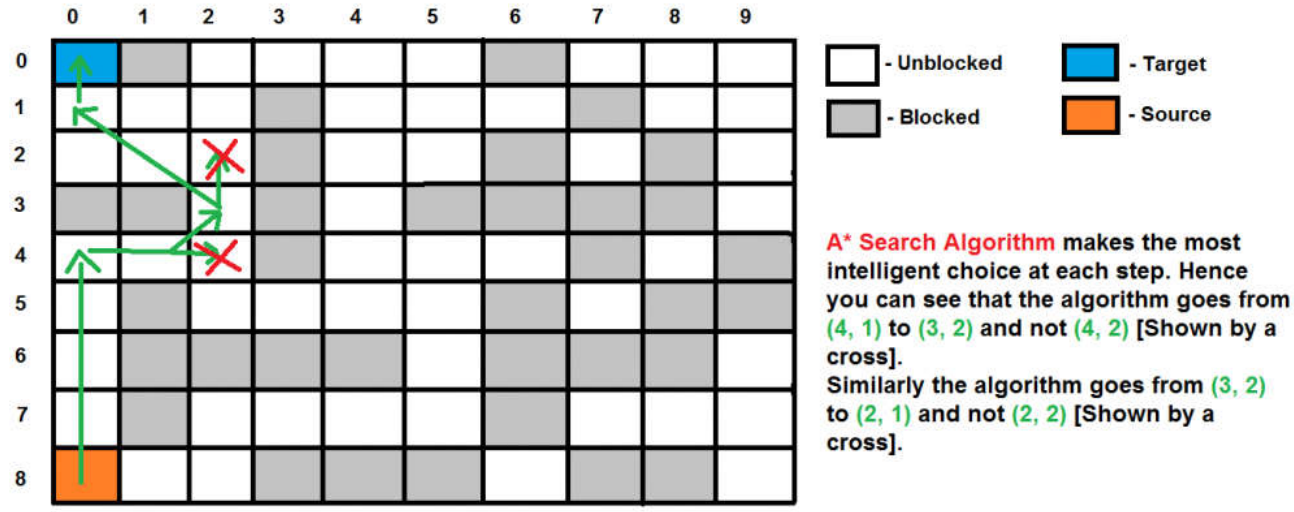
          successor.f = successor.g + successor.h

        ii) if a node with the same position as
            successor is in the OPEN list which has a
           lower f than successor, skip this successor

        iii) 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)

    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).
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 –

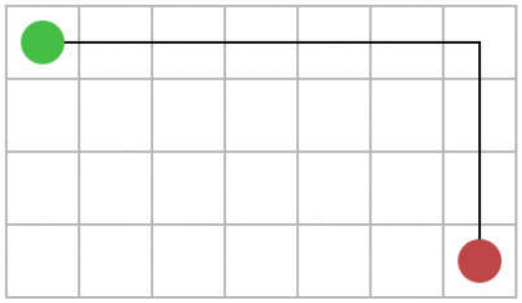
There are generally three approximation heuristics to calculate h –

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 green spot as source cell and red spot as target cell).

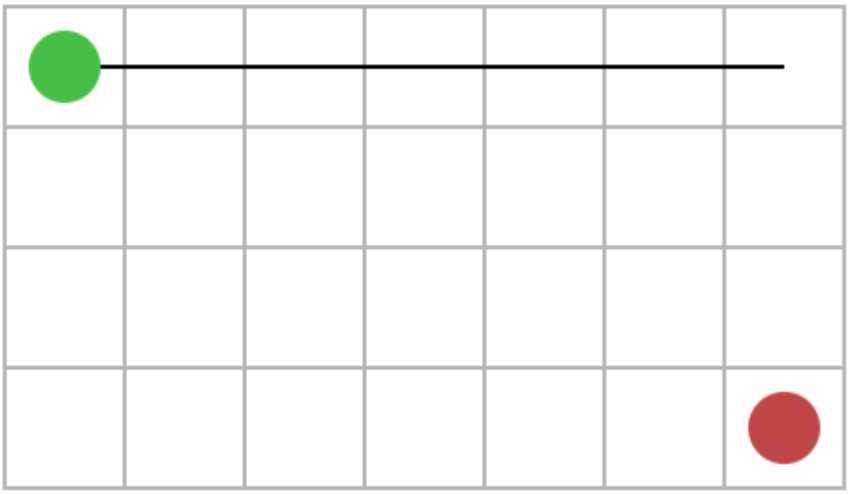


2) Diagonal Distance-

- 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 green spot as source cell and red spot as target cell).
  ```

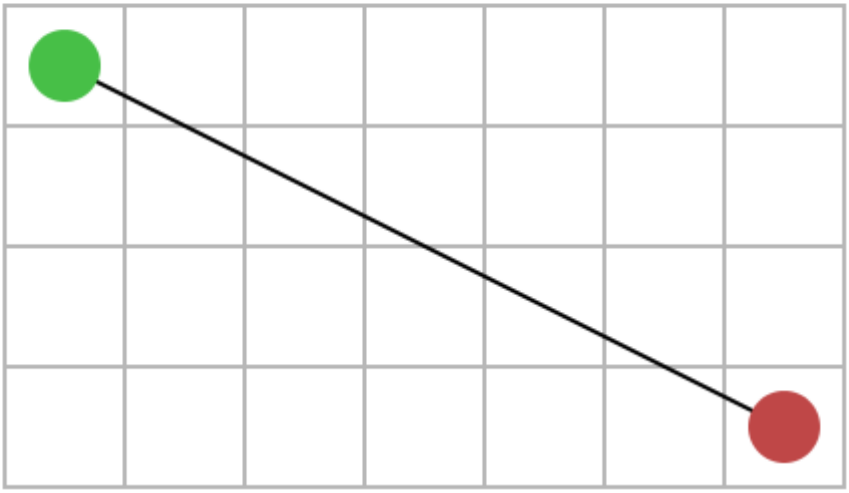


3) Euclidean Distance-

- 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 green spot as source cell and red 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.

```
// 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<int, int> Pair;

// Creating a shortcut for pair<int, pair<int, int>> type
typedef pair<double, pair<int, int>> pPair;
```

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

// A Utility Function to check whether given cell (row, col)
// is a valid cell or not.
bool isValid(int row, int col)
{
    // Returns true if row number and column number
    // is in range
    return (row >= 0) && (row < ROW) &&
           (col >= 0) && (col < COL);
}

// A Utility Function to check whether the given cell is
// blocked or not
bool isUnBlocked(int grid[][COL], int row, int col)
{
    // Returns true if the cell is not blocked else false
    if (grid[row][col] == 1)
        return (true);
    else
        return (false);
}

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

// A Utility Function to calculate the 'h' heuristics.
double calculateHValue(int row, int col, Pair dest)
{
    // Return using the distance formula
    return ((double)sqrt ((row-dest.first)*(row-dest.first)
                          + (col-dest.second)*(col-dest.second)));
}

// A Utility Function to trace the path from the source
// to destination
void tracePath(cell cellDetails[][COL], Pair dest)
{
    printf ("\nThe Path is ");
    int row = dest.first;
    int col = dest.second;

    stack<Pair> Path;

    while (!(cellDetails[row][col].parent_i == row
             && cellDetails[row][col].parent_j == col ))
    {
```

```
        Path.push (make_pair (row, col));
        int temp_row = cellDetails[row][col].parent_i;
        int temp_col = cellDetails[row][col].parent_j;
        row = temp_row;
        col = temp_col;
    }

    Path.push (make_pair (row, col));
    while (!Path.empty())
    {
        pair<int,int> p = Path.top();
        Path.pop();
        printf("-> (%d,%d) ",p.first,p.second);
    }

    return;
}

// A Function to find the shortest path between
// a given source cell to a destination cell according
// to A* Search Algorithm
void aStarSearch(int grid[][COL], Pair src, Pair dest)
{
    // If the source is out of range
    if (isValid (src.first, src.second) == false)
    {
        printf ("Source is invalid\n");
        return;
    }

    // If the destination is out of range
    if (isValid (dest.first, dest.second) == false)
    {
        printf ("Destination is invalid\n");
        return;
    }

    // Either the source or the destination is blocked
    if (isUnBlocked(grid, src.first, src.second) == false ||
            isUnBlocked(grid, dest.first, dest.second) == false)
    {
        printf ("Source or the destination is blocked\n");
        return;
    }

    // If the destination cell is the same as source cell
    if (isDestination(src.first, src.second, dest) == true)
    {
        printf ("We are already at the destination\n");
        return;
    }

    // Create a closed list and initialise it to false which means
    // that no cell has been included yet
    // This closed list is implemented as a boolean 2D array
    bool closedList[ROW][COL];
    memset(closedList, false, sizeof (closedList));

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

```
int i, j;

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

// Initialising the parameters of the starting node
i = src.first, j = src.second;
cellDetails[i][j].f = 0.0;
cellDetails[i][j].g = 0.0;
cellDetails[i][j].h = 0.0;
cellDetails[i][j].parent_i = i;
cellDetails[i][j].parent_j = j;

/*
 Create an open list having information as-
 <f, <i, j>>
 where f = g + h,
 and i, j are the row and column index of that cell
 Note that 0 <= i <= ROW-1 & 0 <= j <= COL-1
 This open list is implenented as a set of pair of pair.*/
set<pPair> openList;

// Put the starting cell on the open list and set its
// 'f' as 0
openList.insert(make_pair (0.0, make_pair (i, j)));

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

while (!openList.empty())
{
    pPair p = *openList.begin();

    // Remove this vertex from the open list
    openList.erase(openList.begin());

    // Add this vertex to the 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)
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
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].g = gNew;
            cellDetails[i-1][j].h = hNew;
            cellDetails[i-1][j].parent_i = i;
            cellDetails[i-1][j].parent_j = j;
```

```
        }
    }
}

//----------- 2nd Successor (South) ------------

// Only process this cell if this is a valid one
if (isValid(i+1, j) == true)
{
    // If the destination cell is the same as the
    // current successor
    if (isDestination(i+1, j, dest) == true)
    {
        // Set the Parent of the destination cell
        cellDetails[i+1][j].parent_i = i;
        cellDetails[i+1][j].parent_j = j;
        printf("The destination cell is found\n");
        tracePath(cellDetails, dest);
        foundDest = true;
        return;
    }
    // If the successor is already on the closed
    // list or if it is blocked, then ignore it.
    // Else do the following
    else if (closedList[i+1][j] == false &&
             isUnBlocked(grid, i+1, j) == true)
    {
        gNew = cellDetails[i][j].g + 1.0;
        hNew = calculateHValue(i+1, j, dest);
        fNew = gNew + hNew;

        // If it isn't on the open list, add it to
        // the open list. Make the current square
        // the parent of this square. Record 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;
        }
    }
}

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

//----------- 4th Successor (West) ------------

// Only process this cell if this is a valid one
if (isValidi, 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;
            cellDetails[i][j-1].h = hNew;
            cellDetails[i][j-1].parent_i = i;
            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].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) ------------

// 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;
        }
    }
}

//----------- 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].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) ------------

    // 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 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);
}
```

Limitations
Although being the best pathfinding 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 (see figure below)

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.

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

Dynamic Programming | Set 16 (Floyd Warshall Algorithm)

The Floyd Warshall Algorithm is for solving the All Pairs Shortest Path problem. The problem is to find shortest distances between every pair of vertices in a given edge weighted directed Graph.

Example:

Input:

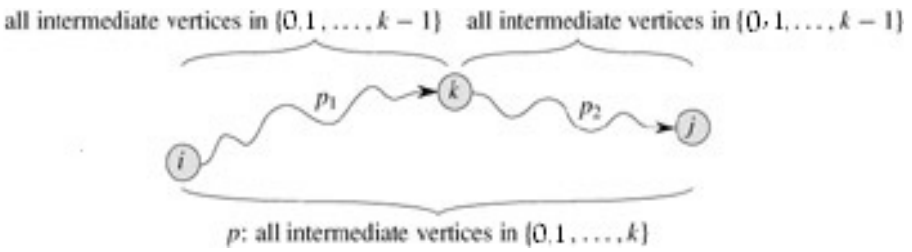
```
       graph[][] = { {0,   5,  INF, 10},
                     {INF,  0,  3,  INF},
                     {INF, INF, 0,   1},
                     {INF, INF, INF, 0} }
which represents the following graph
             10
       (0)------->(3)
        |         /|\
      5 |          |
        |          | 1
       \|/         |
       (1)------->(2)
             3
Note that the value of graph[i][j] is 0 if i is equal to j
And graph[i][j] is INF (infinite) if there is no edge from vertex i to j.
```

Output:

```
Shortest distance matrix
      0      5      8      9
    INF      0      3      4
    INF    INF      0      1
    INF    INF    INF      0
```

Floyd Warshall Algorithm
We initialize the solution matrix same as the input graph matrix as a first step. Then we update the solution matrix by considering all vertices as an intermediate vertex. The idea is to one by one pick all vertices and update all shortest paths which include the picked vertex as an intermediate vertex in the shortest path. When we pick vertex number k as an intermediate vertex, we already have considered vertices {0, 1, 2, .. k-1} as intermediate vertices. For every pair (i, j) of source and destination vertices respectively, there are two possible cases.
1) k is not an intermediate vertex in shortest path from i to j. We keep the value of dist[i][j] as it is.
2) k is an intermediate vertex in shortest path from i to j. We update the value of dist[i][j] as dist[i][k] + dist[k][j].

The following figure is taken from the Cormen book. It shows the above optimal substructure property in the all-pairs shortest path problem.



Following is implementations of the Floyd Warshall algorithm.

```
// C Program for Floyd Warshall Algorithm
#include<stdio.h>
// Number of vertices in the graph
#define V 4
/* Define Infinite as a large enough value. This value will be used
  for vertices not connected to each other */
#define INF 99999

// A function to print the solution matrix
void printSolution(int dist[][V]);

// Solves the all-pairs shortest path problem using Floyd Warshall algorithm
void floydWarshall (int graph[][V])
{
    /* dist[][] will be the output matrix that will finally have the shortest
      distances between every pair of vertices */
    int dist[V][V], i, j, k;

    /* Initialize the solution matrix same as input graph matrix. Or
       we can say the initial values of shortest distances are based
       on shortest paths considering no intermediate vertex. */
    for (i = 0; i < V; i++)
        for (j = 0; j < V; j++)
            dist[i][j] = graph[i][j];

    /* Add all vertices one by one to the set of intermediate vertices.
      ---> Before start of a iteration, we have shortest distances between all
      pairs of vertices such that the shortest distances consider only the
      vertices in set {0, 1, 2, .. k-1} as intermediate vertices.
      ----> After the end of a iteration, vertex no. k is added to the set of
      intermediate vertices and the set becomes {0, 1, 2, .. k} */
    for (k = 0; k < V; k++)
    {
        // Pick all vertices as source one by one
        for (i = 0; i < V; i++)
        {
            // Pick all vertices as destination for the
            // above picked source
            for (j = 0; j < V; j++)
            {
                // If vertex k is on the shortest path from
                // i to j, then update the value of dist[i][j]
                if (dist[i][k] + dist[k][j] < dist[i][j])
                    dist[i][j] = dist[i][k] + dist[k][j];
            }
        }
    }
    // Print the shortest distance matrix
    printSolution(dist);
}
/* A utility function to print solution */
void printSolution(int dist[][V])
{
    printf ("Following matrix shows the shortest distances"
            " between every pair of vertices \n");
    for (int i = 0; i < V; i++)
    {
        for (int j = 0; j < V; j++)
        {
            if (dist[i][j] == INF)
```

```
                printf("%7s", "INF");
            else
                printf ("%7d", dist[i][j]);
        }
        printf("\n");
    }
}

// driver program to test above function
int main()
{
    /* Let us create the following weighted graph
            10
       (0)------->(3)
        |         /|\
      5 |          |
        |          | 1
       \|/         |
       (1)------->(2)
            3           */
    int graph[V][V] = { {0,   5,  INF, 10},
                        {INF, 0,   3, INF},
                        {INF, INF, 0,   1},
                        {INF, INF, INF, 0}
                      };

    // Print the solution
    floydWarshall(graph);
    return 0;
}
```

Output:

```
Following matrix shows the shortest distances between every pair of vertices
      0      5      8      9
    INF      0      3      4
    INF    INF      0      1
    INF    INF    INF      0
```

Time Complexity: O(V^3)

The above program only prints the shortest distances. We can modify the solution to print the shortest paths also by storing the predecessor information in a separate 2D matrix.
Also, the value of INF can be taken as INT_MAX from limits.h to make sure that we handle maximum possible value. When we take INF as INT_MAX, we need to change the if condition in the above program to avoid arithmatic overflow.

```
#include <limits.h>

#define INF INT_MAX
.........................
if ( dist[i][k] != INF &&
     dist[k][j] != INF &&
     dist[i][k] + dist[k][j] < dist[i][j]
    )
 dist[i][j] = dist[i][k] + dist[k][j];
.........................
```

Johnson's algorithm for All-pairs shortest paths

The problem is to find shortest paths between every pair of vertices in a given weighted directed Graph and weights may be negative. We have discussed Floyd Warshall Algorithm for this problem. Time complexity of Floyd Warshall Algorithm is $\Theta(V^3)$. *Using Johnson's algorithm, we can find all pair shortest paths in $O(V^2 log\ V + VE)$ time.* Johnson's algorithm uses both Dijkstra and Bellman-Ford as subroutines.

If we apply Dijkstra's Single Source shortest path algorithm for every vertex, considering every vertex as source, we can find all pair shortest paths in O(V*VLogV) time. So using Dijkstra's single source shortest path seems to be a better option than Floyd Warshell, but the problem with Dijkstra's algorithm is, it doesn't work for negative weight edge.
The idea of Johnson's algorithm is to re-weight all edges and make them all positive, then apply Dijkstra's algorithm for every vertex.

How to transform a given graph to a graph with all non-negative weight edges?
One may think of a simple approach of finding the minimum weight edge and adding this weight to all edges. Unfortunately, this doesn't work as there may be different number of edges in different paths (See this for an example). If there are multiple paths from a vertex u to v, then all paths must be increased by same amount, so that the shortest path remains the shortest in the transformed graph.
The idea of Johnson's algorithm is to assign a weight to every vertex. Let the weight assigned to vertex u be h[u]. We reweight edges using vertex weights. For example, for an edge (u, v) of weight w(u, v), the new weight becomes w(u, v) + h[u] – h[v]. The great thing about this reweighting is, all set of paths between any two vertices are increased by same amount and all negative weights become non-negative. Consider any path between two vertices s and t, weight of every path is increased by h[s] – h[t], all h[] values of vertices on path from s to t cancel each other.

How do we calculate h[] values? Bellman-Ford algorithm is used for this purpose. Following is the complete algorithm. A new vertex is added to the graph and connected to all existing vertices. The shortest distance values from new vertex to all existing vertices are h[] values.

Algorithm:
1) Let the given graph be G. Add a new vertex s to the graph, add edges from new vertex to all vertices of G. Let the modified graph be G'.

2) Run Bellman-Ford algorithm on G' with s as source. Let the distances calculated by Bellman-Ford be h[0], h[1], .. h[V-1]. If we find a negative weight cycle, then return. Note that the negative weight cycle cannot be created by new vertex s as there is no edge to s. All edges are from s.

3) Reweight the edges of original graph. For each edge (u, v), assign the new weight as "original weight + h[u] – h[v]".

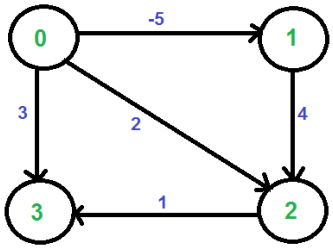
4) Remove the added vertex s and run Dijkstra's algorithm for every vertex.

How does the transformation ensure nonnegative weight edges?
The following property is always true about h[] values as they are shortest distances.

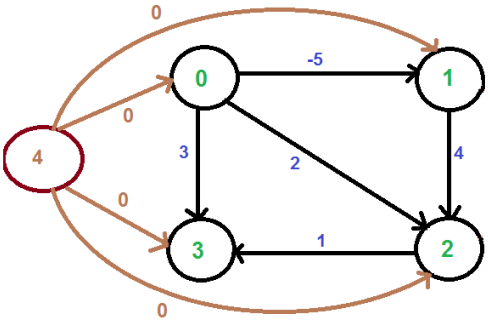
```
h[v] <= h[u] + w(u, v)
```

The property simply means, shortest distance from s to v must be smaller than or equal to shortest distance from s to u plus weight of edge (u, v). The new weights are w(u, v) + h[u] - h[v]. The value of the new weights must

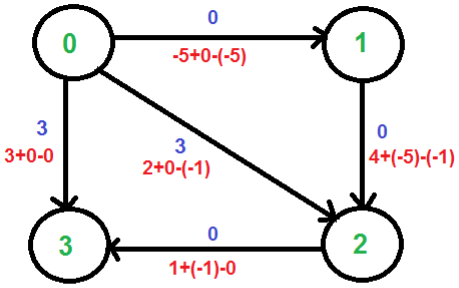
be greater than or equal to zero because of the inequality "h[v] <= h[u] + w(u, v)". **Example:**
Let us consider the following graph.



We add a source s and add edges from s to all vertices of the original graph. In the following diagram s is 4.



We calculate the shortest distances from 4 to all other vertices using Bellman-Ford algorithm. The shortest distances from 4 to 0, 1, 2 and 3 are 0, -5, -1 and 0 respectively, i.e., h[] = {0, -5, -1, 0}. Once we get these distances, we remove the source vertex 4 and reweight the edges using following formula. w(u, v) = w(u, v) + h[u] - h[v].



Distances from 4 to 0, 1, 2 and 3 are 0, -5, -1 and 0 respectievely.

Since all weights are positive now, we can run Dijkstra's shortest path algorithm for every vertex as source.

Time Complexity: The main steps in algorithm are Bellman Ford Algorithm called once and Dijkstra called V times. Time complexity of Bellman Ford is O(VE) and time complexity of Dijkstra is O(VLogV). So overall time complexity is $O(V^2 \log V + VE)$.
The time complexity of Johnson's algorithm becomes same as Floyd Warshell when the graphs is complete (For a complete graph $E = O(V^2)$. But for sparse graphs, the algorithm performs much better than Floyd Warshell.