DEPARTMENT OF COMPUTER SCIENCE ENGINEERING

LAB MANUAL

Subject: **Artificial intelligence**

Semester-7th

KCT COLLEGE OF ENGG AND TECHNOLOGY (FATEHGARH)

Punjab Technical University

INDEX

Experiment No: 1

Aim: Write a program to implement Depth First Search

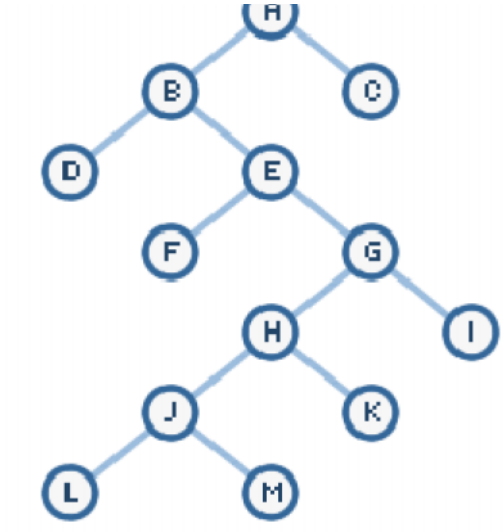
Theory:

If we want to go from Point A to Point B, you are employing some kind of search. For a direction finder, going from Point A to Point B literally means finding a path between where you are now and your intended destination. For a chess game, Point A to Point B might be two points between its current position and its position 5 moves from now. For a genome sequencer, Points A and B could be a link between two DNA sequences

As you can tell, going from Point A to Point B is different for every situation. If there is a vast amount of interconnected data, and you are trying to find a relation between few such pieces of data, you would use search. In this tutorial, you will learn about two forms of searching, depth first and breadth first.

Searching falls under Artificial Intelligence (AI). A major goal of AI is to give computers the ability to think, or in other words, mimic human behavior. The problem is, unfortunately, computers don't function in the same way our minds do. They require a series of *well-reasoned out* steps before finding a solution. Your goal, then, is to take a complicated task and convert it into simpler steps that your computer can handle. That conversion from something complex to something simple is what this tutorial is primarily about. Learning how to use two search algorithms is just a welcome side-effect.

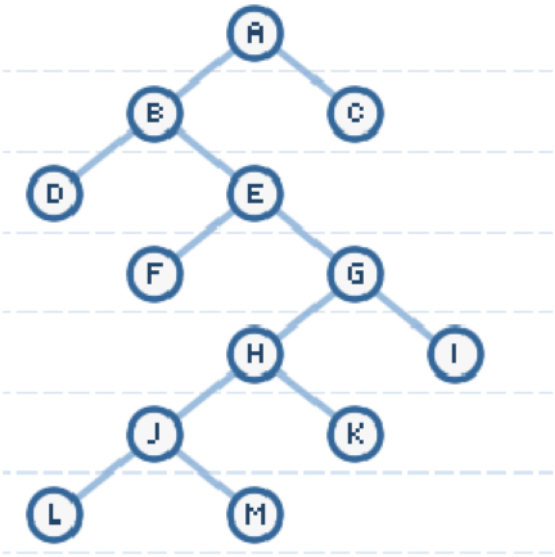
Our Search Representation Let's first learn how we humans would solve a search problem. First, we need a representation of how our search problem will exist. The following is an example of our search tree. It is a series of interconnected nodes that we will be searching through:



In our above graph, the path connections are not two-way. All paths go only from top to bottom. In other words, A has a path to B and C, but B and C do not have a path to A. It is basically like a one-way street.

Each lettered circle in our graph is a node. A node can be connected to other via our edge/path, and those nodes that its connects to are called neighbors. B and C are neighbors of A. E and D are neighbors of B, and B is not a neighbors of D or E because B cannot be reached using either D or E.

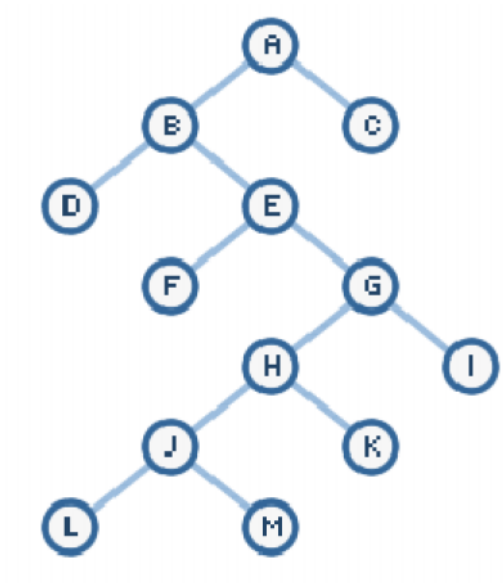
Our search graph also contains depth:



We now have a way of describing location in our graph. We know how the various nodes (the lettered circles) are related to each other (neighbors), and we have a way of characterizing the depth each belongs in. Knowing this information isn't directly relevant in creating our search algorithm, but they do help us to better understand the problem.

Depth First Search Depth first search works by taking a node, checking its neighbors, expanding the first node it finds among the neighbors, checking if that expanded node is our destination, and if not, continue exploring more nodes.

The above explanation is probably confusing if this is your first exposure to depth first search. I hope the following demonstration will help more. Using our same search tree, let's find a path between nodes A and F:



Step 0

let’s start with our root/goal node:

I will be using two lists to keep track of what we are doing - an Open list and a Closed List. An Open list keeps track of what you need to do, and the Closed List keeps track of what you have already done. Right now, we only have our starting point, node A. We haven't done anything to it yet, so let's add it to our Open list

Open List: A
Closed List: <empty>
Step 1

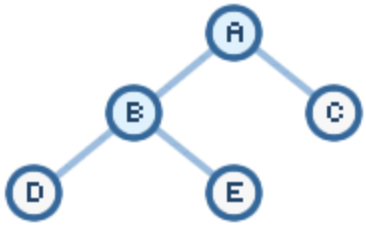
Now, let's explore the neighbors of our A node. To put another way, let's take the first item from our Open list and explore its neighbors:

Node A's neighbors are the B and C nodes. Because we are now done with our A node, we can remove it from our Open list and add it to our Closed List. You aren't done with this step though. You now have two new nodes B and C that need exploring. Add those two nodes to our Open list.

Our current Open and Closed Lists contain the following data:

Open List: B, C
Closed List: A
Step 2

Our Open list contains two items. For depth first search and breadth first search, you always explore the first item from our Open list. The first item in our Open list is the B node. B is not our destination, so let's explore its neighbors:



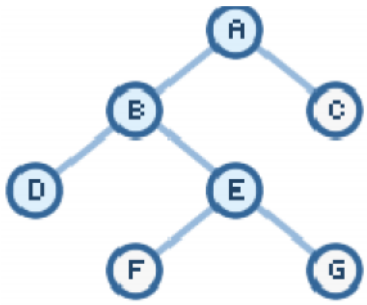
Because I have now expanded B, I am going to remove it from the Open list and add it to the Closed List. Our new nodes are D and E, and we add these nodes to the *beginning* of our Open list:

Open List: D, E, C
Closed List: A, B
Step 3

We should start to see a pattern forming. Because D is at the beginning of our Open List, we expand it. D isn't our destination, and it does not contain any neighbors. All you do in this step is remove D from our Open List and add it to our Closed List

Open List: E, C
Closed List: A, B, D
Step 4

We now expand the E node from our Open list. E is not our destination, so we explore its neighbors and find out that it contains the neighbors F and G. Remember, F is our target, but we don't stop here though. Despite F being on our path, we only end when we are about to *expand* our target Node - F in this case:

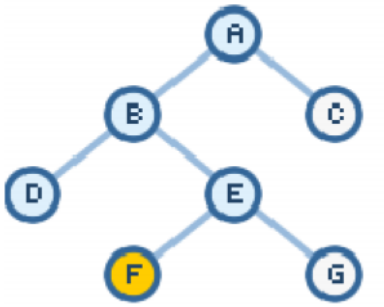


Our Open list will have the E node removed and the F and G nodes added. The removed E node will be added to our Closed List:

Open List: F, G, C
Closed List: A, B, D, E

Step 5

We now expand the F node. Since it is our intended destination, we stop:



We remove F from our Open list and add it to our Closed List. Since we are at our destination, there is no need to expand F in order to find its neighbors. Our final Open and Closed Lists contain the following data:

Open List: G, C
Closed List: A, B, D, E, F

The final path taken by our depth first search method is what the final value of our Closed List is: A, B, D, E, F. Towards the end of this tutorial, I will analyze these results in greater detail so that you have a better understanding of this search method

Pseudocode / Informal Algorithms Now that we have a good idea on how to solve these various search problems, we need to translate them so that a computer can solve it. Before we get into the code, let's solve it generally using pseudocode, or if you prefer a better name, an informal algorithm.

The pseudocode for depth first search is:

i. Declare two empty lists: Open and Closed.
ii. Add Start node to our Open list.
iii. While our Open list is not empty, loop the following:

a. Remove the first node from our Open List.
b. Check to see if the removed node is our destination.

i. If the removed node is our destination, break out of the loop, add the node to our closed list, and return the value of our Closed list.
ii. If the removed node is not our destination, continue the loop (go to Step c).

c. Extract the neighbors of our above removed node.
d. Add the neighbors to the *beginning* of our Open list, and add the removed node to our Closed list.
Continue looping

The following two subsections explain what the grayed out code does in both our search methods

The gray code in a depth first search is replaced with the following
openList.unshift(neighbors[nLength - i - 1]);
Unshift inserts values at the beginning of our array. Therefore, I add the values to the beginning of our array in reverse; hence the awkward syntax: nLength - i - 1. Let me give you an example.

Let's say our neighbors array contains the following three items: **a, b, c**. If we were to unshift each item using neighbors[i], in the first iteration of the for loop, the value of **a** will be added to the first position in our openList. So far so good.
In the second iteration, the value of b will be added to the first position in our openList. So now, openList contains b first, then a: [b, a]. In the third iteration, our openList will look like [c, b, a]. The end result is that the data is reversed
To avoid the reversal, I add the items to our openList in reverse. First c gets inserted, then b, then a. That works in keeping with our algorithm and examples from the previous pages

Result: The depth first search algorithm is implemented.

Experiment No: 2

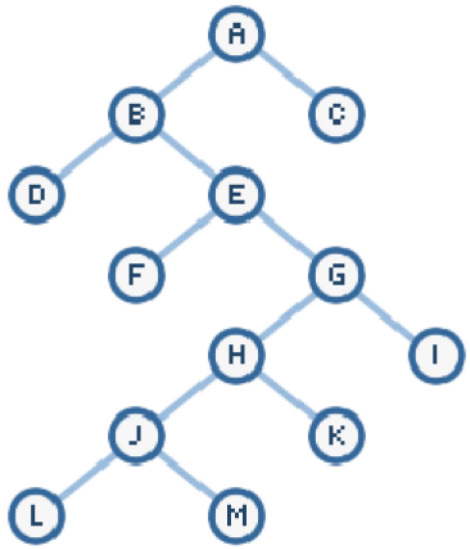
Aim: To implement Breadth First Search Algorithm

Theory:

Breadth First Search

The reason we cover both depth and breadth first search methods in the same tutorial is because they are both similar. In depth first search, newly explored nodes were added to the beginning of your Open list. In breadth first search, newly explored nodes are added to the end of your Open list.

Let's see how that change will affect our results. For reference, here is our original search tree



Let's try to find a path between nodes A and E.

Step 0 Let's start with our root/goal node:

Like before, I will continue to employ the Open and Closed Lists to keep track of what needs to be done:

Open List: A
Closed List: <empty>

Step 1

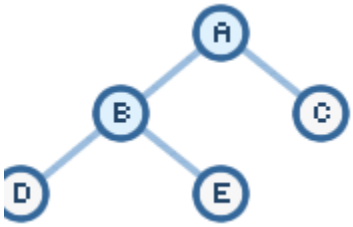
Now, let's explore the neighbors of our A node. So far, we are following in depth first's foot steps:

We remove A from our Open list and add A to our Closed List. A's neighbors, the B and C nodes, are added to our Open list. They are added to the end of our Open list, but since our Open list was empty (after removing A), it's hard to show that in this step

Our current Open and Closed Lists contain the following data:

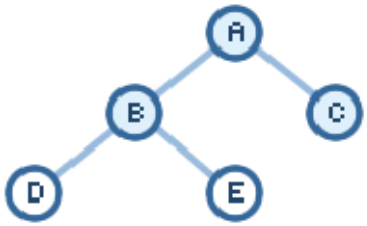
Open List: B, C
Closed List: A
Step 2

Here is where things start to diverge from our depth first search method. We take a look the B node because it appears first in our Open List. Because B isn't our intended destination, we explore its neighbors:



B is now moved to our Closed List, but the neighbors of B, nodes D and E are added to the *end* of our Open list:

Open List: C, D, E
Closed List: A, B
Step 3
we now expand our C node:

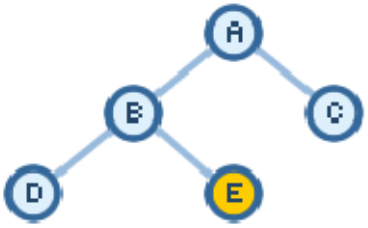


Since C has no neighbors, all we do is remove C from our Closed List and move on:

Open List: D, E
Closed List: A, B, C

Step 4
Similar to Step 3, we expand node D. Since it isn't our destination, and it too does not have any neighbors, we simply remove D from our to Open list, add D to our Closed List, and continue on:
Open List: E
Closed List: A, B, C, D
Step 5

because our Open list only has one item, we have no choice but to take a look at node E. Since node E is our destination, we can stop here:



Our final versions of the Open and Closed Lists contain the following data:

Open List: <empty>
Closed List: A, B, C, D, E

The pseudocode for breadth first first search is:

i. Declare two empty lists: Open and Closed.
ii. Add Start node to our Open list.
iii. While our Open list is not empty, loop the following:

a. Remove the first node from our Open List.
b. Check to see if the removed node is our destination.

i. If the removed node is our destination, break out of the loop, add the node to our Closed list, and return the value of our Closed list.
ii. If the removed node is not our destination, continue the loop (go to Step c).
c. Extract the neighbors of our above removed node.
d. Add the neighbors to the *end* of our Open list, and add the removed node to our Closed list.

Because both depth first and breadth first search methods contain the same data except for one line, I will not describe each method's code separately

var origin:Node = nodeOrigin; var destination:Node = nodeDestination;

In these two lines, you are defining and initializing your origin and destination variables. In order to keep the code example applicable to a wider set of Flash implementations, we are assuming your origin and target nodes are already somehow given to you and stored in the nodeOrigin and nodeDestination variables. Your implementation of it, if you are using the Graph ADT will be different. In the next page, you will see in my implementation, nodeOrigin and nodeDestination are replaced with something else.

// Creating our Open and Closed Lists var closedList:Array = new Array(); var openList:Array = new Array(); // Adding our starting point to Open List openList.push(origin);

We declare our closedList and openList arrays. These arrays store the progress of what we have done so far and a list of what needs to be done. For the initial step from our algorithm, I add the origin node to our openList. With that done, it's time to get into our loop!

// Loop while openList contains some data. while (openList.length != 0) {

As long as our openList contains some data, we continue looping. Ideally, openList should always contain some data such as nodes from recently explored neighbors. IfopenList is empty, besides the loop ending, it also means that you have explored your entire search tree without finding a solution.

// Loop while openList contains some data. var n:Node = Node(openList.shift());

An array's shift method removes the first item from the array and returns that value. Because I specify that the variable n requires an object of type Node, we typecast our openList.shift() operation with the Node(data) operation. If we did not do that, Flash would give me an error. The main thing to remember for the rest of this code explanation is that the variable n refers to the current, active node removed from openList.

// Check if node is Destination if (n == destination) { closedList.push(destination); trace("Done!"); break; }

we first check if n is our destination. If the current node you are examining is the destination, you are done. we add n to our closedList array and trace the string Done!. If you are going to be parsing your data or making any modifications to your final output stored in closedList, you would do so right here before the break command.

// Store n's neighbors in array var neighbors:Array = n.getNeighbors(); var nLength:Number = neighbors.length;

If your program makes it here, it means that n is not our intended destination. We now have to find out who its neighbors are. I use the getNeighbors method on n to return an array of its neighbors. In order to simplify how future lines of code look, I store the length of our recent neighbors data in the nLength variable.

// Add each neighbor to the beginning of our openList for (i=0; i< nLength; i++) { < depends on search method > }

This for loop cycles through each of our neighbors and adds the values to openList. The location at which the neighbors get added to our openList (grayed out line) is where the difference between depth first and breadth first searches occur. The gray code in a breadth first search is replaced with the following:
openList.push(neighbors[i]);

This is very straightforward. Push adds a value to the end of your array, and since that is what we need to do for breadth first search anyway, we are done...well, almost.

// Add current node to closedList closedList.push(n);

Finally we are at the end. Since we have already done everything we can to this node (check if it is our destination and get its neighbors), let's retire it by adding it to our closedList array.

Result: The breadth first search algorithm is implemented.

Experiment No: 3

Aim: Write a Program to implement Traveling Salesman Problem

Introduction:

Problem

A traveling salesman has to travel through a bunch of cities, in such a way that the expenses on traveling are minimized. This is the infamous Traveling Salesman Problem (aka **TSP**) problem it belongs to a family of problems, called NP-complete problem. It is conjectured that all those problems requires exponential time to solve them. In our case, this means that to find the optimal solution you have to go through all possible routes, and the numbers of routes increases exponential with the numbers of cities.

Formal Definition

We are given a complete undirected graph **G** that has a nonnegative integer cost (weight) associated with each edge, and we must find a Hamiltonian cycle (a tour that passes through all the vertices) of G with minimum cost.

Theory:

If you want to get a notion of what numbers we are talking about look at this: the number of routes with 50 cities is (50-2)!, which is

12,413,915,592,536,072,670,862,289,047,373,375,038,521,486,354,677,760,000,000,000

An alternative approach would be to compute a solution which is not optimal, but is guaranteed to be close the optimal solution. We present here an applet that implements such an approximation algorithm for the Euclidean TSP problem. In our case we have points in the plane (i.e. cities) and the cost of the traveling between two points is the distance between them. In other words, we have a map with cities, any two of which are connected by a direct straight road and we want to find a shortest tour for our poor traveling salesman, who "wants" to visit every city.

Description of Algorithms

x2 Algorithm:

Find Minimum Spanning Tree (MST).
Perform preoder tree walk on MST, and construct a list of the vertices as we encounter them. (i.e. each vertex will appear only one - corresponding to its first encounter)
The output tour is hamiltonian cycle that visits the vertices in order of this list

The running time of this algorithm is O ($n^2 \log(n)$) time; since the input is a complete graph (n is the number of inserted points). The length of the resulting tour is at most twice of the optimal tour, since its length is at most twice that of the MST, and the optimal tour is longer than the MST.

X1.5 Algorithm:

Find Minimum Spanning Tree (MST).
Find minimum weighted matching between odd vertices of MST.
Find an Euler tour in resulting graph and create list of vertices that represents it.
Find Hamilton cycle (which is in fact TSP tour) by visiting vertices in order of created list when only first appearance of vertex in list is encountered and any other appearance is skipped.

The approximation ratio bound is 1.5, although the argument here is a bit more complicated.

Iterative Algorithm:

We generated an arbitrary initial solution, by visiting all points in order they were inserted. Then in each iteration:
1. Select two random cities
2. Interchange the two cities predecessors
3. Calculate the weight of resulting tour.
4. If new weight is smaller then old one - if so, replace the old tour by the new tour, else undo (2).

Removing Intersections Scheme:

This algorithm is applied on already computed TSP tour. It performs a loop which stops only after all intersections have been removed (which must happen after at most n^2 times). Each loop goes through all the pairs of edges in the tour and checks if they intersect. - If so then two new edges will replace them. New edges created by interchanging the original edge's endpoints. To preserve TSP tour completeness the path between the original edges is reversed

Result: The traveling salesman problem is implemented.

Experiment No: 4

Aim: Write a program to implement 8 puzzle problem.
Introduction:

Write a program to solve the 8-puzzle problem (and its natural generalizations) using the A* search algorithm
The problem
The 8-puzzle problem is a puzzle invented and popularized by Noyes Palmer Chapman in the 1870s. It is played on a 3-by-3 grid with 8 square blocks labeled 1 through 8 and a blank square. Your goal

is to rearrange the blocks so that they are in order. You are permitted to slide blocks horizontally or vertically into the blank square. The following shows a sequence of legal moves from an initial board position (left) to the goal position (right).

1 3 1 3 1 2 3 1 2 3 1 2 3 4 2 5 => 4 2 5 => 4 5 => 4 5 => 4 5 6 7 8 6 7 8 6 7 8 6 7 8 6 7 8 initial goal

Theory:

Best-first search: We now describe an algorithmic solution to the problem that illustrates a general artificial intelligence methodology known as the A* search algorithm. We define a *state* of the game to be the board position, the number of moves made to reach the board position, and the previous state. First, insert the initial state (the initial board, 0 moves, and a null previous state) into a priority queue. Then, delete from the priority queue the state with the minimum priority, and insert onto the priority queue all neighboring states (those that can be reached in one move). Repeat this procedure until the state dequeued is the goal state. The success of this approach hinges on the choice of *priority function* for a state. We consider two priority functions:

Hamming priority function. The number of blocks in the wrong position, plus the number of moves made so far to get to the state. Intuitively, states with a small number of blocks in the wrong position are close to the goal state, and we prefer states that have been reached using a small number of moves.

Manhattan priority function. The sum of the distances (sum of the vertical and horizontal distance) from the blocks to their goal positions, plus the number of moves made so far to get to the state.

For example, the Hamming and Manhattan priorities of the initial state below are 5 and 10, respectively.

8 1 3 1 2 3 1 2 3 4 5 6 7 8 1 2 3 4 5 6 7 8 4 2 4 5 6 ---------- ---------- ----- 7 6 5 7 8 1 1 0 0 1 1 0 1 1 2 0 0 2 2 0 3 initial goal Hamming = 5 + 0 Manhattan = 10 + 0

We make a key oberservation: to solve the puzzle from a given state on the priority queue, the total number of moves we need to make (including those already made) is at least its priority, using either the Hamming or Manhattan priority function. (For Hamming priority, this is true because each block out of place must move at least once to reach its goal position. For Manhattan priority, this is true because each block must move its Manhattan distance from its goal position. Note that we do not count the blank tile when computing the Hamming or Manhattan priorities.)

A critical optimization: After implementing best-first search, you will notice one annoying feature: states corresponding to the same board position are enqueued on the priority queue many times. To prevent unnecessary exploration of useless states, when considering the neighbors of a state, don't enqueue the neighbor if its board position is the same as the previous state.

8 1 3 8 1 3 8 1 3 4 2 4 2 4 2 7 6 5 7 6 5 7 6 5 previous state disallow

The input will consist of the board dimension *N* followed by the *N*-by-*N* initial board position. The input format uses 0 to represent the blank square. As an example, 3 0 1 3 4 2 5 7 8 6 1 3 4 2 5 7 8 6 1 3 4 2 5 7 8 6 1 2 3 4 5 7 8 6 1 2 3 4 5 7 8 6 1 2 3 4 5 6 7 8 Number of states enqueued = 10 Number of moves = 4

Note that your program should work for arbitrary *N*-by-*N* boards (for any *N* greater than 1), even if it is too slow to solve some of them in a reasonable amount of time.

Detecting infeasible puzzles:

Not all initial board positions can lead to the goal state. Modify your program to detect and report such situations. 3 1 2 3 4 5 6 8 7 0

Hint: use the fact that board positions are divided into two equivalence classes with respect to reachability: (i) those that lead to the goal position and (ii) those that lead to the goal position if we modify the initial board by swapping any pair of adjacent (non-blank) blocks. There are two ways to apply the hint:

Run the A* algorithm simultaneously on two puzzle instances - one with the initial board and one with the initial board modified by swapping a pair of adjacent (non-blank) blocks. Exactly one of the two will lead to the goal position.

Derive a mathematical formula that tells you whether a board is solvable or not.

Deliverables:

Organize your program in an appropriate number of data types. At a minimum, you are required to implement the following APIs. Though, you are free to add additional methods or data types (such as State).

```
public class Board {
public Board(int[][] tiles) // construct a board from an N-by-N array of tiles
public int hamming() // return number of blocks out of place
public int manhattan() // return sum of Manhattan
 distances between blocks and goal
 public boolean equals(Object y) // does this board equal y
public Iterable<Board> neighbors() // return an Iterable of all neighboring board positions
public String toString() // return a string representation of the board
// test client
public static void main(String[] args) }
 public class Solver {
 public Solver(Board initial) // find a solution to the initial board
 public boolean isSolvable() // is the initial board solvable?
 public int moves() // return min number of moves to solve the initial board;
 // -1 if no such solution
 public String toString() // return string representation of solution (as described above)
 // read puzzle instance from stdin and print solution to stdout (in format above)
 public static void main(String[] args)}
```

Result: The program is implemented to solve 8 puzzle problem.

Experiment No: 5

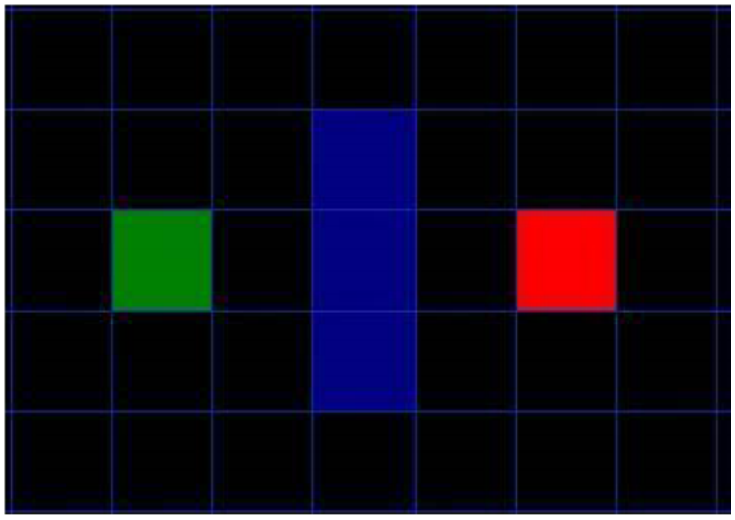
Aim: Write a program to implement A* program.

The A* (pronounced A-star) algorithm can be complicated for beginners. While there are many articles on the web that explain A*, most are written for people who understand the basics already.

Introduction:

The Search

Area Let's assume that we have someone who wants to get from point A to point B. Let's assume that a wall separates the two points. This is illustrated below, with green being the starting point A, and red being the ending point B, and the blue filled squares being the wall in between.



The first thing you should notice is that we have divided our search area into a square grid. Simplifying the search area, as we have done here, is the first step in pathfinding. This particular method reduces our search area to a simple two dimensional array. Each item in the array represents one of the squares on the grid, and its status is recorded as walkable or unwalkable. The path is found by figuring out which squares we should take to get from A to B. Once the path is found, our person moves from the center of one square to the center of the next until the target is reached.

These center points are called ―nodes . When you read about pathfinding elsewhere, you will often see people discussing nodes. Why not just call them squares? Because it is possible to divide up your pathfinding area into something other than squares. They could be rectangles, hexagons, triangles, or any shape, really. And the nodes could be placed anywhere within the shapes – in the center or along the edges, or anywhere else. We are using this system, however, because it is the simplest.

Starting the Search

Once we have simplified our search area into a manageable number of nodes, as we have done with the grid layout above, the next step is to conduct a search to find the shortest path. We do this by starting at point A, checking the adjacent squares, and generally searching outward until we find our target.

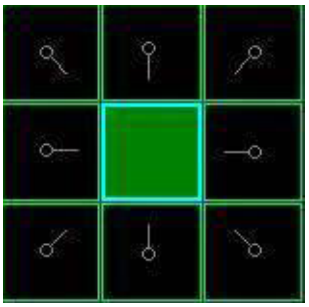
We begin the search by doing the following:

1. Begin at the starting point A and add it to an ―open list of squares to be considered. The open list is kind of like a shopping list. Right now there is just one item on the list, but we will have more later. It contains squares that might fall along the path you want to take, but maybe not. Basically, this is a list of squares that need to be checked out.

2. Look at all the reachable or walkable squares adjacent to the starting point, ignoring squares with walls, water, or other illegal terrain. Add them to the open list, too. For each of these squares, save point A as its ―parent square . This parent square stuff is important when we want to trace our path. It will be explained more later.

3. Drop the starting square A from your open list, and add it to a ―closed list of squares that you don't need to look at again for now.

At this point, you should have something like the following illustration. In this illustration, the dark green square in the center is your starting square. It is outlined in light blue to indicate that the square has been added to the closed list. All of the adjacent squares are now on the open list of squares to be checked, and they are outlined in light green. Each has a gray pointer that points back to its parent, which is the starting square.



Next, we choose one of the adjacent squares on the open list and more or less repeat the earlier process, as described below. But which square do we choose? The one with the lowest F cost.

Path Scoring

The key to determining which squares to use when figuring out the path is the following equation:

$F = G + H$

where

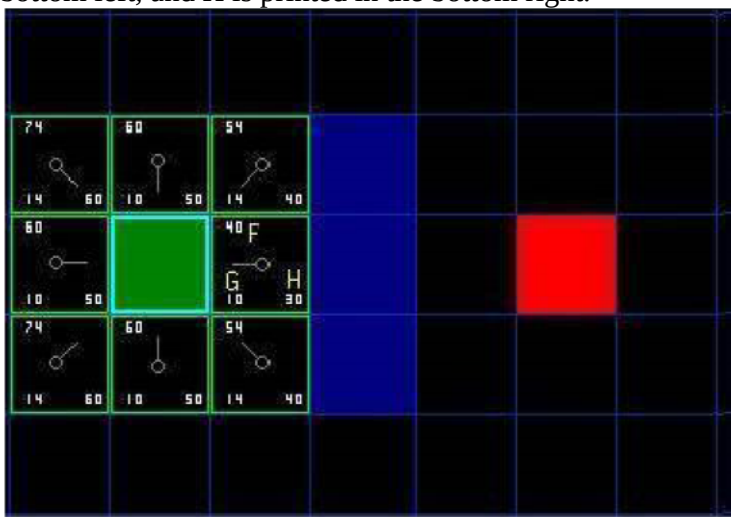
G = the movement cost to move from the starting point A to a given square on the grid, following the path generated to get there.

H = the estimated movement cost to move from that given square on the grid to the final destination, point B. This is often referred to as the heuristic, which can be a bit confusing. The reason why it is called that is because it is a 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.). You are given one way to calculate H in this tutorial, but there are many others that you can find in other articles on the web.

Our path is generated by repeatedly going through our open list and choosing the square with the lowest F score. This process will be described in more detail a bit further in the article. First let's look more closely at how we calculate the equation. As described above, G is the movement cost to move from the starting point to the given square using the path generated to get there. In this example, we will assign a cost of 10 to each horizontal or vertical square moved, and a cost of 14 for a diagonal move. We use these numbers because the actual distance to move diagonally is the square root of 2 (don't be scared), or roughly 1.414 times the cost of moving horizontally or vertically. We use 10 and 14 for simplicity's sake. The ratio is about right, and we avoid having to calculate square roots and we avoid decimals. This isn't just because we are dumb and don't like math. Using whole numbers like these is a lot faster for the computer, too. As you will soon find out, pathfinding can be very slow if you don't use short cuts like these.

Since we are calculating the G cost along a specific path to a given square, the way to figure out the G cost of that square is to take the G cost of its parent, and then add 10 or 14 depending on whether it is diagonal or orthogonal (non-diagonal) from that parent square. The need for this method will become apparent a little further on in this example, as we get more than one square away from the starting square. H can be estimated in a variety of ways. The method we use here is called the Manhattan method, where you calculate the total number of squares moved horizontally and vertically to reach the target square from the current square, ignoring diagonal movement, and ignoring any obstacles that may be in the way. We then multiply the total by 10, our cost for moving one square horizontally or vertically. This is (probably) called the Manhattan method because it is like calculating the number of city blocks from one place to another, where you can't cut across the block diagonally.

Reading this description, you might guess that the heuristic is merely a rough estimate of the remaining distance between the current square and the target "as the crow flies." This isn't the case. We are actually trying to estimate the remaining distance along the path (which is usually farther). The closer our estimate is to the actual remaining distance, the faster the algorithm will be. If we overestimate this distance, however, it is not guaranteed to give us the shortest path. In such cases, we have what is called an "inadmissible heuristic." Technically, in this example, the Manhattan method is inadmissible because it slightly overestimates the remaining distance. But we will use it anyway because it is a lot easier to understand for our purposes, and because it is only a slight overestimation. On the rare occasion when the resulting path is not the shortest possible, it will be nearly as short. F is calculated by adding G and H. The results of the first step in our search can be seen in the illustration below. The F, G, and H scores are written in each square. As is indicated in the square to the immediate right of the starting square, F is printed in the top left, G is printed in the bottom left, and H is printed in the bottom right.



So let's look at some of these squares. In the square with the letters in it, G = 10. This is because it is just one square from the starting square in a horizontal direction. The squares immediately above, below, and to the left of the starting square all have the same G score of 10. The diagonal squares have G scores of 14. The H scores are calculated by estimating the Manhattan distance to the red target square, moving only horizontally and vertically and ignoring the wall that is in the way. Using this method, the square to the immediate right of the start is 3 squares from the red square, for a H score of 30. The square just above this square is 4 squares away (remember, only move horizontally and vertically) for an H score of 40. You can probably see how the H scores are calculated for the other squares.

The F score for each square, again, is simply calculated by adding G and H together.

Continuing the Search

To continue the search, we simply choose the lowest F score square from all those that are on the open list. We then do the following with the selected square:

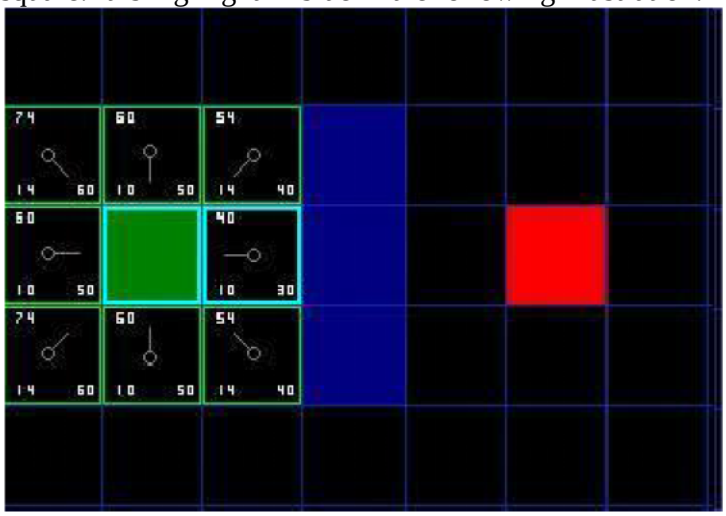
4) Drop it from the open list and add it to the closed list.

5) Check all of the adjacent squares. Ignoring those that are on the closed list or unwalkable (terrain with walls, water, or other illegal terrain), add squares to the open list if they are not on the open list already. Make the selected square the —parent of the new squares

6) If an adjacent square is already on the open list, check to see if this path to that square is a better one. In other words, check to see if the G score for that square is lower if we use the current square to

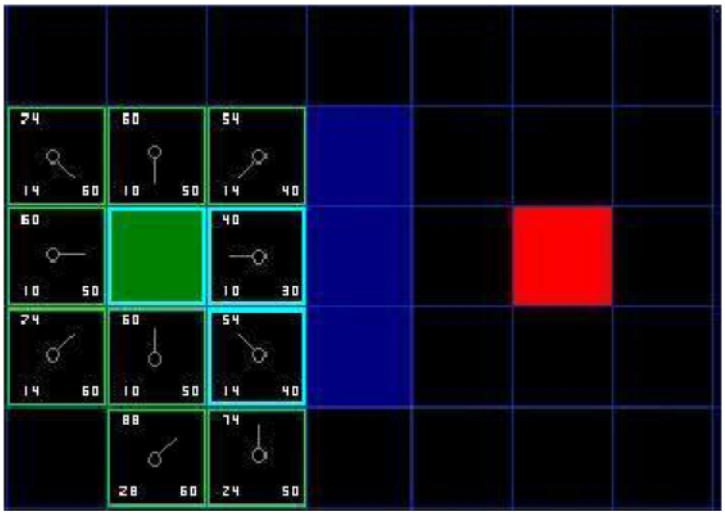
get there. If not, don't do anything. On the other hand, if the G cost of the new path is lower, change the parent of the adjacent square to the selected square (in the diagram above, change the direction of the pointer to point at the selected square). Finally, recalculate both the F and G scores of that square. If this seems confusing, you will see it illustrated below.

Okay, so let's see how this works. Of our initial 9 squares, we have 8 left on the open list after the starting square was switched to the closed list. Of these, the one with the lowest F cost is the one to the immediate right of the starting square, with an F score of 40. So we select this square as our next square. It is highlight in blue in the following illustration.

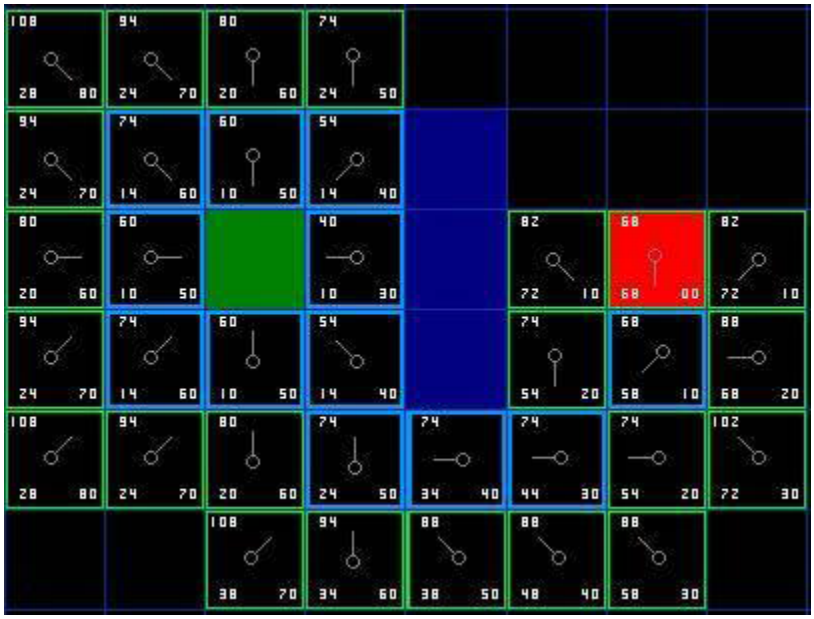


First, we drop it from our open list and add it to our closed list (that's why it's now highlighted in blue). Then we check the adjacent squares. Well, the ones to the immediate right of this square are wall squares, so we ignore those. The one to the immediate left is the starting square. That's on the closed list, so we ignore that, too. The other four squares are already on the open list, so we need to check if the paths to those squares are any better using this square to get there, using G scores as our point of reference. Let's look at the square right above our selected square. Its current G score is 14. If we instead went through the current square to get there, the G score would be equal to 20 (10, which is the G score to get to the current square, plus 10 more to go vertically to the one just above it). A G score of 20 is higher than 14, so this is not a better path. That should make sense if you look at the diagram. It's more direct to get to that square from the starting square by simply moving one square diagonally to get there, rather than moving horizontally one square, and then vertically one square. When we repeat this process for all 4 of the adjacent squares already on the open list, we find that none of the paths are improved by going through the current square, so we don't change anything. So now that we looked at all of the adjacent squares, we are done with this square, and ready to move to the next square.

So we go through the list of squares on our open list, which is now down to 7 squares, and we pick the one with the lowest F cost. Interestingly, in this case, there are two squares with a score of 54. So which do we choose? It doesn't really matter. For the purposes of speed, it can be faster to choose the last one you added to the open list. This biases the search in favor of squares that get found later on in the search, when you have gotten closer to the target. But it doesn't really matter. (Differing treatment of ties is why two versions of A* may find different paths of equal length.) So let's choose the one just below, and to the right of the starting square, as is shown in the following illustration



This time, when we check the adjacent squares we find that the one to the immediate right is a wall square, so we ignore that. The same goes for the one just above that. We also ignore the square just below the wall. Why? Because you can't get to that square directly from the current square without cutting across the corner of the nearby wall. You really need to go down first and then move over to that square, moving around the corner in the process. (Note: This rule on cutting corners is optional. Its use depends on how your nodes are placed.) That leaves five other squares. The other two squares below the current square aren't already on the open list, so we add them and the current square becomes their parent. Of the other three squares, two are already on the closed list (the starting square, and the one just above the current square, both highlighted in blue in the diagram), so we ignore them. And the last square, to the immediate left of the current square, is checked to see if the G score is any lower if you go through the current square to get there. No dice. So we're done and ready to check the next square on our open list. We repeat this process until we add the target square to the closed list, at which point it looks something like the illustration below



So how do we determine the path? Simple, just start at the red target square, and work backwards moving from one square to its parent, following the arrows. This will eventually take you back to the starting square, and that's your path. It should look like the following illustration. Moving from the starting square A to the destination square B is simply a matter of moving from the center of each square (the node) to the center of the next square on the path, until you reach the target.

Summary of the A Method*

Okay, now that you have gone through the explanation, let's lay out the step-by-step method all in one place:

1) Add the starting square (or node) to the open list.

2) Repeat the following:

a) Look for the lowest F cost square on the open list. We refer to this as the current square.

b) Switch it to the closed list.

c) For each of the 8 squares adjacent to this current square …

If it is not walkable or if it is on the closed list, ignore it. Otherwise do the following.

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.

If it is on the open list already, check to see if this path to that square is better, using G cost as the measure. A lower G cost means that this is a better path. If so, change the parent of the square to the current square, and recalculate the G and F scores of the square. If you are keeping your open list sorted by F score, you may need to resort the list to account for the change.

d) Stop when you:

Add the target square to the closed list, in which case the path has been found (see note below), or

Fail to find the target square, and the open list is empty. In this case, there is no path.

3) Save the path. Working backwards from the target square, go from each square to its parent square until you reach the starting square. That is your path.

Pseudo code for A* Algorithm:

```
Create a node containing the goal state
node_goal
Create a node containing the start state
node_start
Put node_start on the open list
while the OPEN list is not empty
{
Get the node off the open list with the
lowest f and call it node_current
if node_current is the same state as
node_goal we have found the solution;
break from the while loop
Generate each state node_successor that
can come after node_current
for each node_successor of node_current
{
Set the cost of node_successor to be the
```

cost of node_current plus the cost to get to node_successor from node_current
find node_successor on the OPEN list
if node_successor is on the OPEN list but the existing one is as good or better then discard this successor and continue
if node_successor is on the CLOSED list but the existing one is as good or better then discard this successor and continue

Remove occurrences of node successor from OPEN and CLOSED
Set the parent of node_successor to node_current
Set h to be the estimated distance to node_goal (Using the heuristic function)
Add node_successor to the OPEN list
}
Add node_current to the CLOSED list

Result: The A* algorithm is implemented for finding the least cost path from a given initial node to one goal node.

Experiment No: 6

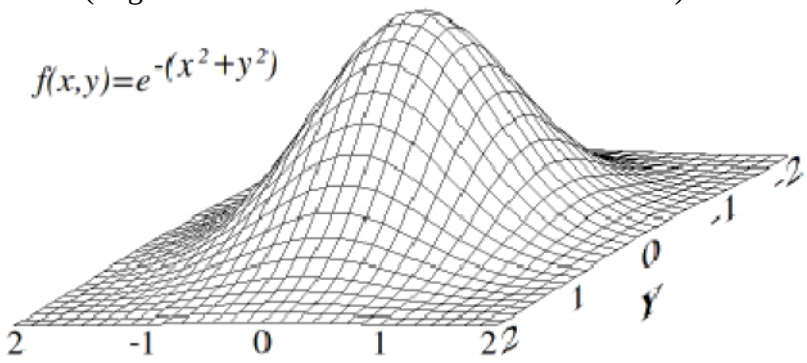
Aim: Write a program to implement Hill climbing Algorithm

In computer science, *hill climbing* is a mathematical optimization technique which belongs to the family of local search. It is relatively simple to implement, making it a popular first choice. Although more advanced algorithms may give better results, in some situations hill climbing works just as well. Hill climbing can be used to solve problems that have many solutions, some of which are better than others. It starts with a random (potentially poor) solution, and iteratively makes small changes to the solution, each time improving it a little. When the algorithm cannot see any improvement anymore, it terminates. Ideally, at that point the current solution is close to optimal, but it is not guaranteed that hill climbing will ever come close to the optimal solution.

For example, hill climbing can be applied to the traveling salesman problem. It is easy to find a solution that visits all the cities but will be very poor compared to the optimal solution. The algorithm starts with such a solution and makes small improvements to it, such as switching the order in which two cities are visited. Eventually, a much better route is obtained. Hill climbing is used widely in artificial intelligence, for reaching a goal state from a starting node. Choice of next node and starting node can be varied to give a list of related algorithms.

Mathematical description

Hill climbing attempts to maximize (or minimize) a function $f(x)$, where x are discrete states. These states are typically represented by vertices in a graph, where edges in the graph encode nearness or similarity of a graph. Hill climbing will follow the graph from vertex to vertex, always locally increasing (or decreasing) the value of f, until a local maximum (or local minimum) x_m is reached. Hill climbing can also operate on a continuous space: in that case, the algorithm is called gradient ascent (or gradient descent if the function is minimized).*.



Variants

In *simple hill climbing*, the first closer node is chosen, whereas in *steepest ascent hill climbing* all successors are compared and the closest to the solution is chosen. Both forms fail if there is no closer node, which may happen if there are local maxima in the search space which are not solutions. Steepest ascent hill climbing is similar to best-first search, which tries all possible extensions of the current path instead of only one. *Stochastic hill climbing* does not examine all neighbors before deciding how to move. Rather, it selects a neighbour at random, and decides (based on the amount of improvement in that neighbour) whether to move to that neighbour or to examine another.

Random-restart hill climbing is a meta-algorithm built on top of the hill climbing algorithm. It is also known as *Shotgun hill climbing*. It iteratively does hill-climbing, each time with a random initial condition x_0. The best x_m is kept: if a new run of hill climbing produces a better x_m than the stored state, it replaces the stored state. Random-restart hill climbing is a surprisingly effective algorithm in many cases. It turns out that it is often better to spend CPU time exploring the space, than carefully optimizing from an initial condition.

Problems
Local maxima

A problem with hill climbing is that it will find only local maxima. Unless the heuristic is convex, it may not reach a global maximum. Other local search algorithms try to overcome this problem such as stochastic hill climbing, random walks and simulated annealing. This problem of hill climbing can be solved by using random hill climbing search technique.

Ridges

A ridge is a curve in the search place that leads to a maximum, but the orientation of the ridge compared to the available moves that are used to climb is such that each move will lead to a smaller point. In other words, each point on a ridge looks to the algorithm like a local maximum, even though the point is part of a curve leading to a better optimum.

Plateau

Another problem with hill climbing is that of a plateau, which occurs when we get to a "flat" part of the search space, i.e. we have a path where the heuristics are all very close together. This kind of flatness can cause the algorithm to cease progress and wander aimlessly.

Pseudocode

Hill Climbing Algorithm

```
currentNode = startNode;
 loop do
 L = NEIGHBORS(currentNode);
 nextEval = -INF;
 nextNode = NULL;
for all x in L if (EVAL(x) > nextEval) nextNode = x;
 nextEval = EVAL(x);
if nextEval <= EVAL(currentNode) //Return current node since no better neighbors exist return currentNode;
currentNode = nextNode;
```

Result: The Hill Climbing algorithm is implemented. The algorithm terminates when goal state is reached from the starting node.