Name: Selas Moro **Class**: BE CMPN **Roll No: 39** **Pid:** 182074

Experiment No. 4: A*Algorithm

Aim: To implement A*algorithm in 8 Puzzle Problem.

Theory:

A-star (also referred as A*) is one of the most successful search algorithms to find the shortest path between nodes of graphs. It is an informed search algorithm , as it uses information about path cost and also heuristics to find the solution.

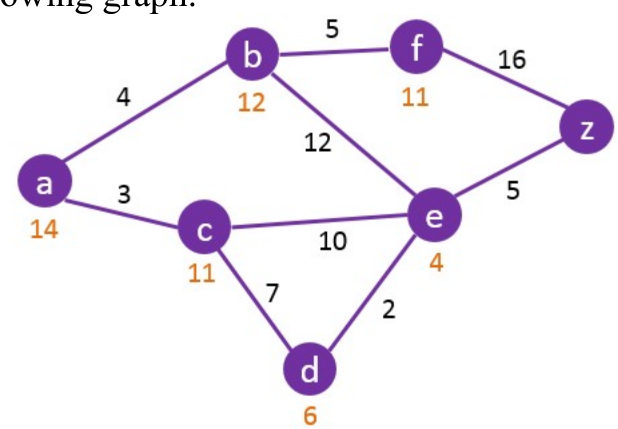
It is best-known form of Best First search. It avoids expanding paths that are already expensive, but expands most promising paths first.

f(n) = g(n) + h(n), where

g(n) the cost (so far) to reach the node
h(n) estimated cost to get from the node to the goal
f(n) estimated total cost of path through n to goal. It is implemented using priority queue by increasing f(n).

Experiment:

The aim of A* algorithm is to traverse the graph from start node A to end node Z.. Stack is used in the implementation of this algorithm. Consider how A* Algorithm reaches the destination node based on the information given with respect to the following graph:

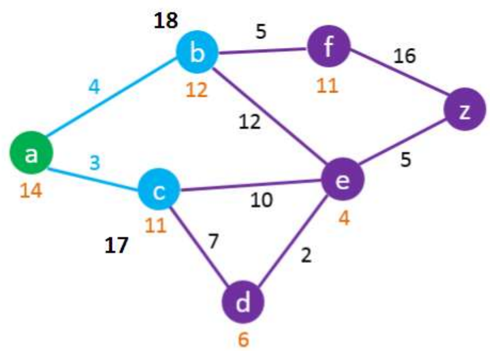


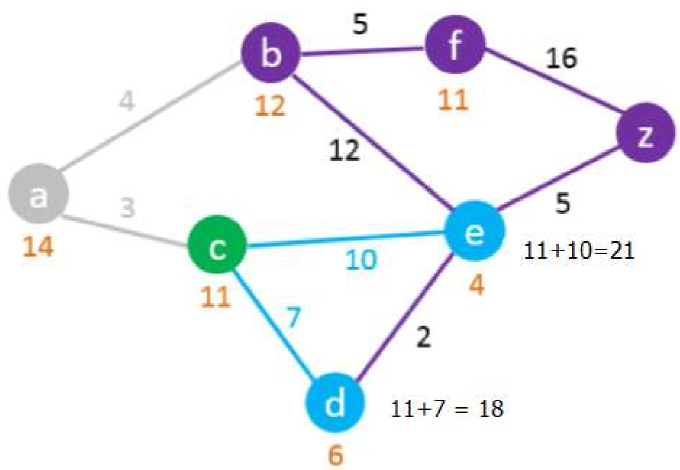
Algorithmic Steps

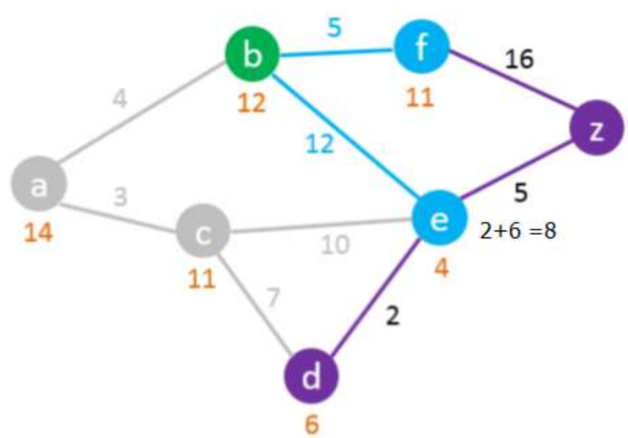
1. **Step 1**: Push the root node in the Stack.
2. **Step 2**: Compute f(n) to the connected node

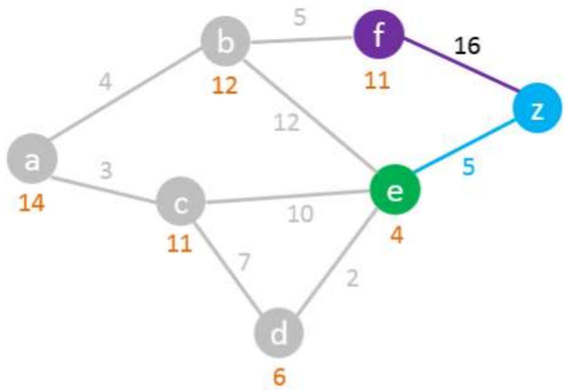
3. **Step 3**: Select the node with the lowest value of f(n) and move it to stack.
4. **Step 4**: If the removed node has unvisited child nodes, mark them as visited and consider the new node and check if it is the end node. If the node is not end node treat it as the root node and follow Step 2, else stop to retrieve path from stack .

Example:









We found the shortest path from A to Z.
Read the path from Z to A using the previous node column:
Z > E > D > C > A
So the Shortest Path is: A – C – D – E – Z with a length of 17
.

Experiment Exercise: Implement A* Algorithm.

Code:

```
import argparse


class Node:

    #----- Initalize node with pattern gfunction the blanklocation and the move used to reach that state
-----#
    def __init__(self, pattern, gfunc, move='start'):
        self.pattern = pattern
        self.gfunc = gfunc
        self.move = move
        for (row, i) in zip(pattern, range(3)):
            if 0 in row:
                self.blankloc = [i, row.index(0)]
                # print(self.blankloc[0],self.blankloc[1])

    #----- equal magic function to check if states are equal or node element by element-----#
    def __eq__(self, other):
        if other == None:
            return False

        if isinstance(other, Node) != True:
            raise TypeError

        for i in range(3):
            for j in range(3):
                if self.pattern[i][j] != other.pattern[i][j]:
                    return False
        return True

    #----- magic function to retrive an element from the node like an array -----#
    def __getitem__(self, key):
        if isinstance(key, tuple) != True:
            raise TypeError
        if len(key) != 2:
            raise KeyError

        return self.pattern[key[0]][key[1]]

    #----- function to calculate hfunction according to given goal -----#
    def calc_hfunc(self, goal):
        self.hfunc = 0
        for i in range(3):
            for j in range(3):
                # print (i,j)
                if self.pattern[i][j] != goal.pattern[i][j]:
                    self.hfunc += 1
        if self.blankloc != goal.blankloc:
            self.hfunc -= 1  # Remove one counter if the blank location is displaced because it
overestimates the goal
```

```
        self.ffunc = self.hfunc+self.gfunc

        return self.hfunc, self.gfunc, self.ffunc

    #----- Function to move the blank tile left if possible -----#
    def moveleft(self):
        if self.blankloc[1] == 0:
            return None

        left = [[self.pattern[i][j] for j in range(3)]for i in range(3)]
        left[self.blankloc[0]][self.blankloc[1]
                        ] = left[self.blankloc[0]][self.blankloc[1]-1]
        left[self.blankloc[0]][self.blankloc[1]-1] = 0

        return Node(left, self.gfunc+1, 'LEFT')

    #----- Function to move the blank tile right if possible -----#
    def moveright(self):
        if self.blankloc[1] == 2:
            return None

        right = [[self.pattern[i][j] for j in range(3)]for i in range(3)]
        right[self.blankloc[0]][self.blankloc[1]
                        ] = right[self.blankloc[0]][self.blankloc[1]+1]
        right[self.blankloc[0]][self.blankloc[1]+1] = 0

        return Node(right, self.gfunc+1, 'RIGHT')

    #----- Function to move the blank tile up if possible -----#
    def moveup(self):
        if self.blankloc[0] == 0:
            return None

        up = [[self.pattern[i][j] for j in range(3)]for i in range(3)]
        up[self.blankloc[0]][self.blankloc[1]
                        ] = up[self.blankloc[0]-1][self.blankloc[1]]
        up[self.blankloc[0]-1][self.blankloc[1]] = 0

        return Node(up, self.gfunc+1, 'UP')

    #----- Function to move the blank tile down if possible -----#
    def movedown(self):
        if self.blankloc[0] == 2:
            return None

        down = [[self.pattern[i][j] for j in range(3)]for i in range(3)]
        down[self.blankloc[0]][self.blankloc[1]
                        ] = down[self.blankloc[0]+1][self.blankloc[1]]
        down[self.blankloc[0]+1][self.blankloc[1]] = 0

        return Node(down, self.gfunc+1, 'DOWN')

    #----- Function to check and perform all the moves according to possiblity and weather the next
```

```
move is closed or not -----#
    #----- Close this node and all the new nodes to open list -----#
    def moveall(self, game):
        left = self.moveleft()
        left = None if game.isclosed(left) else left
        right = self.moveright()
        right = None if game.isclosed(right) else right
        up = self.moveup()
        up = None if game.isclosed(up) else up
        down = self.movedown()
        down = None if game.isclosed(down) else down

        game.closeNode(self)
        game.openNode(left)
        game.openNode(right)
        game.openNode(up)
        game.openNode(down)

        return left, right, up, down

    #----- Function to print the array in beautifed format -----#

    def print(self, l2):
        l1 = []
        l1.append(self.move)
        l1.append(self.gfunc)
        l1.append(self.hfunc)
        l1.append(self.pattern[0])
        l1.append(self.pattern[1])
        l1.append(self.pattern[2])
        l2.insert(0, l1)

    def printl(l2):
        for i in range(len(l2)):
            print("(STATE-{},{},f(n) = {})".format(
                l2[i][1], l2[i][0], l2[i][1]+l2[i][2]))
            print(l2[i][3])
            print(l2[i][4])
            print(l2[i][5])
            print("\n")


class Game:

    #---- Initilaize Node with start, goal, a hashtable of open nodes, a hashtable of closed Node and
add the start to the open node ----#
    #---- Open nodes is a hash table based on 'f function' and Closed nodes is a hash table based on 'h
function' ----#
    def __init__(self, start, goal):
        self.start = start
        self.goal = goal
        self.open = {}
        self.closed = {}
        _, _, ffunc = self.start.calc_hfunc(self.goal)
```

```
    self.open[ffunc] = [start]

#---- Function to check weather a node is in closed node or not ----#
def isclosed(self, node):
    if node == None:  # return True if no node
        return True

    # calculate hfucntion to check in that list of the hash table
    hfunc, _, _ = node.calc_hfunc(self.goal)

    if hfunc in self.closed:
        for x in self.closed[hfunc]:
            if x == node:
                return True

    return False

#---- Function to add a node to the closed list and remove it from the open nodes list ----#
def closeNode(self, node):
    if node == None:  # return back if no node
        return

    hfunc, _, ffunc = node.calc_hfunc(self.goal)
    # remove from the list of the ffunction of the hash table for open nodes
    self.open[ffunc].remove(node)
    if len(self.open[ffunc]) == 0:
        # remove the attribute for a ffunction if its list is empty
        del self.open[ffunc]

    if hfunc in self.closed:
        self.closed[hfunc].append(node)
    else:
        self.closed[hfunc] = [node]

    return

#---- Function to add a node to the open list after its initilaized ----#
def openNode(self, node):
    if node == None:
        return

    # Calculate ffucntion to add the node to the list of that ffucntion in hash table
    _, _, ffunc = node.calc_hfunc(self.goal)
    if ffunc in self.open:
        self.open[ffunc].append(node)
    else:
        self.open[ffunc] = [node]

    return

#---- Function to solve the game using A star algorithm ----#
def solve(self):

    presentNode = None
```

```
        l2 = []
        while(presentNode != self.goal):
            i = 0
            while i not in self.open:
                i += 1  # Check for the list with least 'ffunction' to pick a node from that list
            presentNode = self.open[i][-1]
            # Expand that node for next possible moves
            presentNode.moveall(self)

    #---- Print the solution in reverse direction i.e. from goal to start----#
        while presentNode.move != 'start':
            presentNode.print(l2)
            # do reverse move that what was done to reach the state to backtrack along the solution
            if presentNode.move == 'UP':
                presentNode = presentNode.movedown()
            elif presentNode.move == 'DOWN':
                presentNode = presentNode.moveup()
            elif presentNode.move == 'RIGHT':
                presentNode = presentNode.moveleft()
            elif presentNode.move == 'LEFT':
                presentNode = presentNode.moveright()

            hfunc, _, _ = presentNode.calc_hfunc(self.goal)
            for i in self.closed[hfunc]:
                if i == presentNode:
                    presentNode = i

        Node.printl(l2)
        return


if __name__ == '__main__':
    startrow = [int(i) for i in input(
        "Enter the start state space separated >>>>").split(" ")]
    goalrow = [int(i) for i in input(
        "Enter the goal state space separated >>>>").split(" ")]
    x = [1, 2, 3, 4, 5, 6, 7, 8, 0]

    #----- Assert if Input is correct -----#
    assert set(x) == set(startrow)
    assert set(x) == set(goalrow)

    #----- Reformat Input -----#
    startloc = [startrow[0:3], startrow[3:6], startrow[6:]]
    goalloc = [goalrow[0:3], goalrow[3:6], goalrow[6:]]

    #----- Initalize start and end node -----#
    start = Node(startloc, 0)
    goal = Node(goalloc, 0, 'goal')

    #----- Initilaize Game -----#
    game = Game(start, goal)
    game.solve()  # Solve Game
```

Output:

```
= RESTART: C:\Users\Administrator.MAHESHC\Documents\SEM 7\SEM_7\AISC Lab\exp04.py
Enter the start state space separated >>>>3 7 6 5 1 2 4 0 8
Enter the goal state space separated >>>>5 3 6 7 0 2 4 1 8
(STATE-1,UP,f(n) = 4)
[3, 7, 6]
[5, 0, 2]
[4, 1, 8]


(STATE-2,UP,f(n) = 5)
[3, 0, 6]
[5, 7, 2]
[4, 1, 8]


(STATE-3,LEFT,f(n) = 5)
[0, 3, 6]
[5, 7, 2]
[4, 1, 8]


(STATE-4,DOWN,f(n) = 5)
[5, 3, 6]
[0, 7, 2]
[4, 1, 8]


(STATE-5,RIGHT,f(n) = 5)
[5, 3, 6]
[7, 0, 2]
[4, 1, 8]
```

Post Experiment Exercise:

1. Implement A* Algorithm to solve 8-puzzle problems using any programming language.

Code:

```
class Node:
  def __init__(self,data,level,fval):
    """ Initialize the node with the data, level of the node and the calculated fvalue """
    self.data = data
    self.level = level
    self.fval = fval

  def generate_child(self):
    """ Generate child nodes from the given node by moving the blank space
      either in the four directions {up,down,left,right} """
    x,y = self.find(self.data,'_')
    """ val_list contains position values for moving the blank space in either of
      the 4 directions [up,down,left,right] respectively. """
    val_list = [[x,y-1],[x,y+1],[x-1,y],[x+1,y]]
    children = []
    for i in val_list:
      child = self.shuffle(self.data,x,y,i[0],i[1])
      if child is not None:
        child_node = Node(child,self.level+1,0)
        children.append(child_node)
    return children

  def shuffle(self,puz,x1,y1,x2,y2):
    """ Move the blank space in the given direction and if the position value are out
      of limits the return None """
    if x2 >= 0 and x2 < len(self.data) and y2 >= 0 and y2 < len(self.data):
      temp_puz = []
      temp_puz = self.copy(puz)
      temp = temp_puz[x2][y2]
      temp_puz[x2][y2] = temp_puz[x1][y1]
      temp_puz[x1][y1] = temp
      return temp_puz
    else:
      return None


  def copy(self,root):
    """ Copy function to create a similar matrix of the given node"""
    temp = []
    for i in root:
      t = []
      for j in i:
        t.append(j)
      temp.append(t)
    return temp

  def find(self,puz,x):
    """ Specifically used to find the position of the blank space """
    for i in range(0,len(self.data)):
      for j in range(0,len(self.data)):
```

```
            if puz[i][j] == x:
                return i,j


class Puzzle:
    def __init__(self,size):
        """ Initialize the puzzle size by the specified size,open and closed lists to empty """
        self.n = size
        self.open = []
        self.closed = []

    def accept(self):
        """ Accepts the puzzle from the user """
        puz = []
        for i in range(0,self.n):
            temp = input().split(" ")
            puz.append(temp)
        return puz

    def f(self,start,goal):
        """ Heuristic Function to calculate hueristic value f(x) = h(x) + g(x) """
        return self.h(start.data,goal)+start.level

    def h(self,start,goal):
        """ Calculates the different between the given puzzles """
        temp = 0
        for i in range(0,self.n):
            for j in range(0,self.n):
                if start[i][j] != goal[i][j] and start[i][j] != '_':
                    temp += 1
        return temp


    def process(self):
        """ Accept Start and Goal Puzzle state"""
        print("Enter the start state matrix \n")
        start = self.accept()
        print("Enter the goal state matrix \n")
        goal = self.accept()

        start = Node(start,0,0)
        start.fval = self.f(start,goal)
        """ Put the start node in the open list"""
        self.open.append(start)
        print("\n\n")
        while True:
            cur = self.open[0]
            print("")
            print("  | ")
            print("  | ")
            print(" \\\'/ \n")
            for i in cur.data:
                for j in i:
                    print(j,end=" ")
                print("")
            """ If the difference between current and goal node is 0 we have reached the goal node"""
            if(self.h(cur.data,goal) == 0):
```

```
            break
        for i in cur.generate_child():
            i.fval = self.f(i,goal)
            self.open.append(i)
        self.closed.append(cur)
        del self.open[0]

        """ sort the opne list based on f value """
        self.open.sort(key = lambda x:x.fval,reverse=False)


puz = Puzzle(3)
puz.process()
```

Output:

```
==================== RESTART: C:\Users\Administrator.MAHESHC\Documents\SEM 7\Airtificial intelligence\exp4_post.py ====================
Enter the start state matrix

1 2 3
_ 4 6
7 5 8
Enter the goal state matrix

1 2 3
4 5 6
_ 7 8



  |
  |
 \'/

1 2 3
_ 4 6
7 5 8

  |
  |
 \'/

1 2 3
4 _ 6
7 5 8

  |
  |
 \'/

1 2 3
4 5 6
7 _ 8

  |
  |
 \'/

1 2 3
4 5 6
_ 7 8
```

Conclusion:

A* Algorithm is one of the best and popular techniques used for path finding and graph traversals. It is used in various applications, such as maps. In maps the A* algorithm is used to calculate the shortest distance between the source (initial state) and the destination (final state). In this experiment, we learn about A* algorithm and successfully implemented it using 8 puzzles.