**CERTIFICATE**

This is to certify that Mr/Miss. \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

with Roll No.:-\_\_\_\_\_\_\_\_\_has successfully completed the necessary course of experiments in the subject of \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ during the academic year 20 -20 complying with the requirements of University of Mumbai, for the course of B.Sc. computer science/Information Technology Semester \_\_\_\_\_\_\_\_.

Date:-\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Lecturer In-charge

College Seal

INDEX

|  |  |  |
| --- | --- | --- |
| Sr No | INDEX | Signature |
| 1 |  |  |
| 2 |  |  |
| 3 |  |  |
| 4 |  |  |
| 5 |  |  |
| 6 |  |  |
| 7 |  |  |
| 8 |  |  |
| 9 |  |  |
| 10 |  |  |

PRACTICAL .1

Aim: - 1.a.Write a program to implement depth first search algorithm.

Procedure:- Click -> Start -> Window->All program->Python IDLE 3.4 -> ENTER

Start-> New Folder->Write a program->Save->RUN.

Code:

class Vertex:

def \_\_init\_\_(self, n):

self.name = n

self.neighbors = list()

self.discovery = 0

self.finish = 0

self.color = 'black'

def add\_neighbor(self, v):

if v not in self.neighbors:

self.neighbors.append(v)

self.neighbors.sort()

class Graph:

vertices = {}

time = 0

def add\_vertex(self, vertex):

if isinstance(vertex, Vertex) and vertex.name not in self.vertices:

self.vertices[vertex.name] = vertex

return True

else:

return False

def add\_edge(self, u, v):

if u in self.vertices and v in self.vertices:

for key, value in self.vertices.items():

if key == u:

value.add\_neighbor(v)

if key == v:

value.add\_neighbor(u)

return True

else:

return False

def print\_graph(self):

for key in sorted(list(self.vertices.keys())):

print(key + str(self.vertices[key].neighbors) + " " + str(self.vertices[key].discovery) + "/" + str(self.vertices[key].finish))

def \_dfs(self, vertex):

global time

vertex.color = 'red'

vertex.discovery = time

time += 1

for v in vertex.neighbors:

if self.vertices[v].color == 'black':

self.\_dfs(self.vertices[v])

vertex.color = 'blue'

vertex.finish = time

time += 1

def dfs(self, vertex):

global time

time = 1

self.\_dfs(vertex)

g = Graph()

# print(str(len(g.vertices)))

a = Vertex('A')

g.add\_vertex(a)

g.add\_vertex(Vertex('B'))

for i in range(ord('A'), ord('K')):

g.add\_vertex(Vertex(chr(i)))

edges = ['AB', 'AE', 'BF', 'CG', 'DE', 'DH', 'EH', 'FG', 'FI', 'FJ', 'GJ', 'HI']

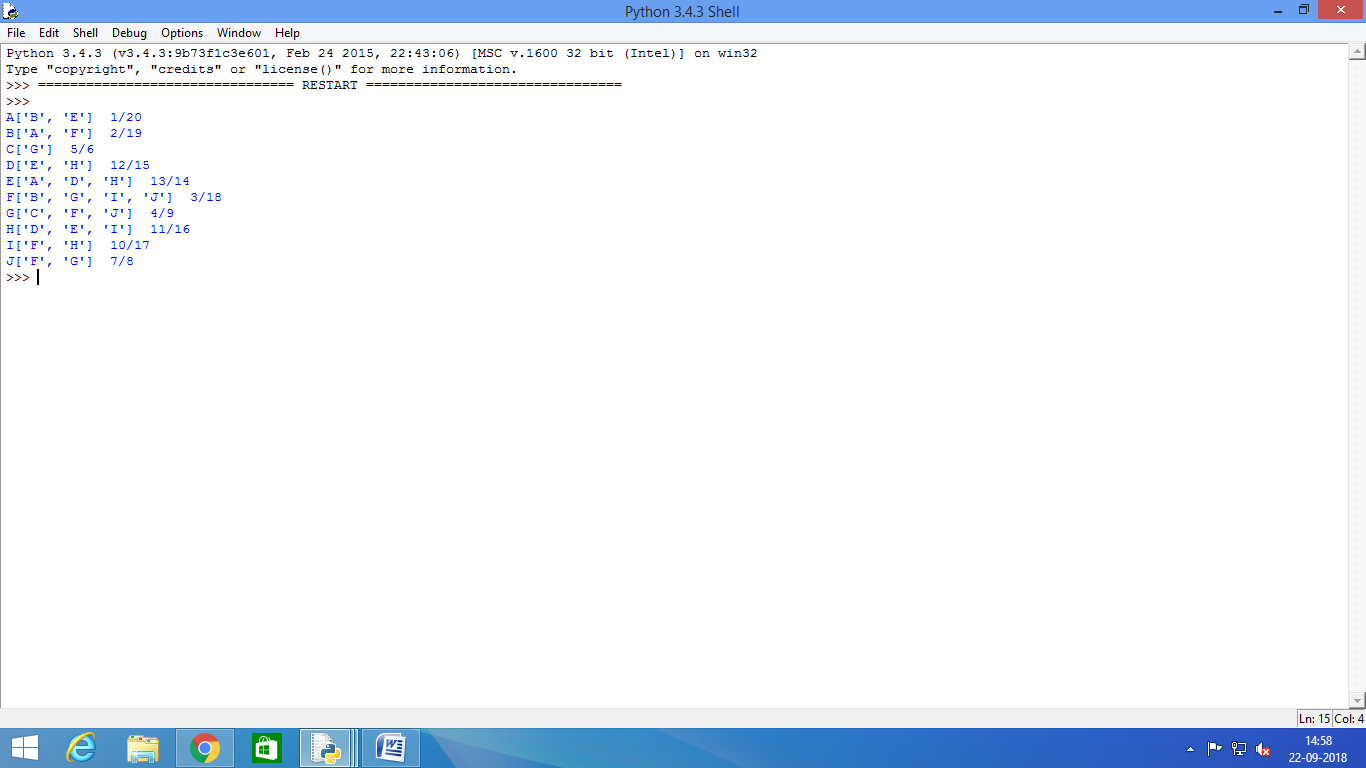
for edge in edges:

g.add\_edge(edge[:1], edge[1:])

g.dfs(a)

g.print\_graph()

output:



Aim: 1.b. write a program to implement breadth first search algorithm.

Procedure:- Click -> Start -> Window->All program->Python IDLE 3.4 -> ENTER

Start-> New Folder->Write a program->Save->RUN.

Code:

class Vertex:

def \_\_init\_\_(self, n):

self.name = n

self.neighbors = list()

self.distance = 9999

self.color = 'black'

def add\_neighbor(self, v):

if v not in self.neighbors:

self.neighbors.append(v)

self.neighbors.sort()

class Graph:

vertices = {}

def add\_vertex(self, vertex):

if isinstance(vertex, Vertex) and vertex.name not in self.vertices:

self.vertices[vertex.name] = vertex

return True

else:

return False

def add\_edge(self, u, v):

if u in self.vertices and v in self.vertices:

for key, value in self.vertices.items():

if key == u:

value.add\_neighbor(v)

if key == v:

value.add\_neighbor(u)

return True

else:

return False

def print\_graph(self):

for key in sorted(list(self.vertices.keys())):

print(key + str(self.vertices[key].neighbors) + " " + str(self.vertices[key].distance))

def bfs(self, vert):

q = list()

vert.distance = 0

vert.color = 'red'

for v in vert.neighbors:

self.vertices[v].distance = vert.distance + 1

q.append(v)

while len(q) > 0:

u = q.pop(0)

node\_u = self.vertices[u]

node\_u.color = 'red'

for v in node\_u.neighbors:

node\_v = self.vertices[v]

if node\_v.color == 'black':

q.append(v)

if node\_v.distance>node\_u.distance + 1:

node\_v.distance = node\_u.distance + 1

g = Graph()

a = Vertex('A')

g.add\_vertex(a)

g.add\_vertex(Vertex('B'))

for i in range(ord('A'), ord('K')):

g.add\_vertex(Vertex(chr(i)))

edges = ['AB', 'AE', 'BF', 'CG', 'DE', 'DH', 'EH', 'FG', 'FI', 'FJ', 'GJ', 'HI']

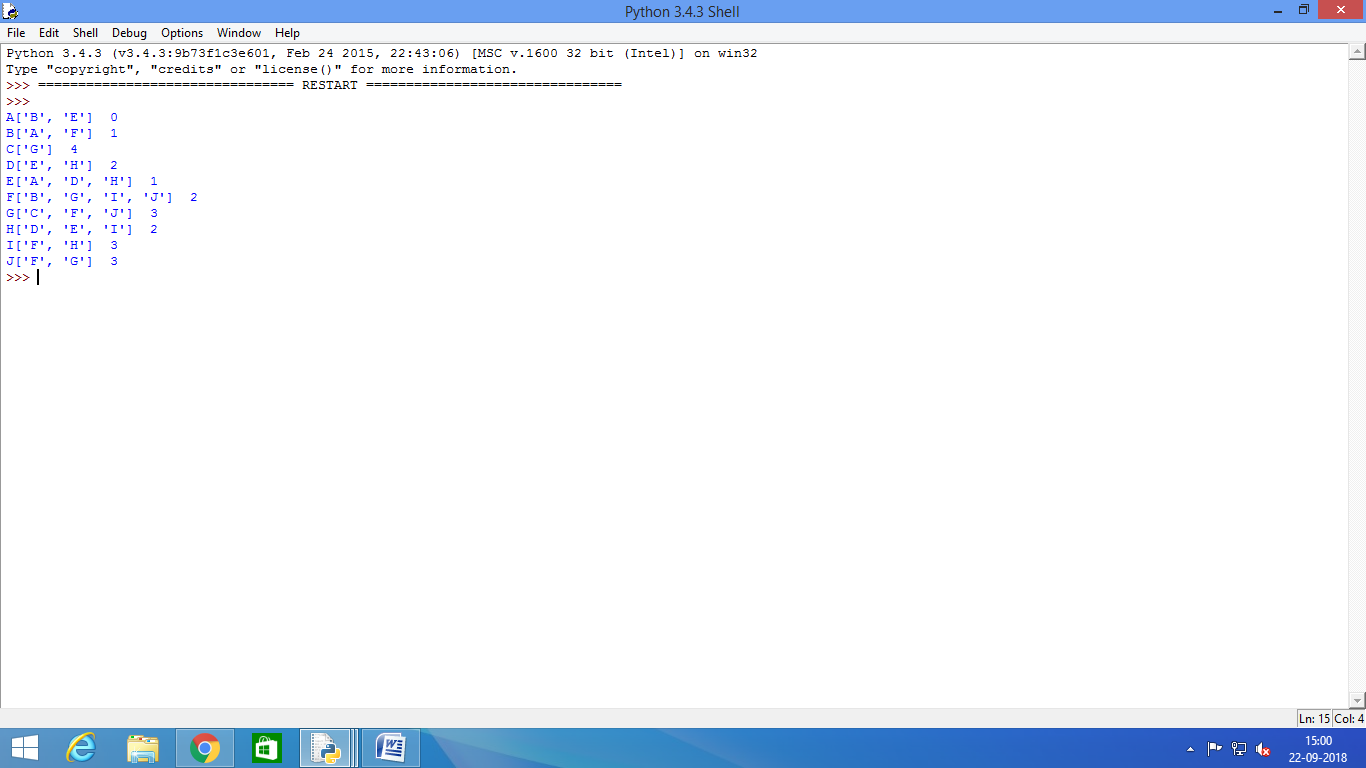
for edge in edges:

g.add\_edge(edge[:1], edge[1:])

g.bfs(a)

g.print\_graph()

output:



PRACTICAL .2

Aim: 2.a.write a program to simulate 4-queen/N-queen problem.

Procedure:- Click -> Start -> Window->All program->Python IDLE 3.4 -> ENTER

Start-> New Folder->Write a program->Save->RUN.

Code:

**import** copy

**deftake\_input**():

"""Accepts the size of the chess board"""

**whileTrue**:

**try**:

size = int(input('What is the size of the chessboard? n = \n'))

**if** size == 1:

print("Trivial solution, choose a board size of atleast 4")

**if** size <= 3:

print("Enter a value such that size>=4")

**continue**

**return** size

**except**ValueError:

print("Invalid value entered. Enter again")

**defget\_board**(size):

"""Returns an n by n board"""

board = [0]\*size

**for** ix **in** range(size):

board[ix] = [0]\*size

**return** board

**defprint\_solutions**(solutions, size):

"""Prints all the solutions in user friendly way"""

**for** sol **in** solutions:

**for** row **in** sol:

print(row)

print()

**defis\_safe**(board, row, col, size):

"""Check if it's safe to place a queen at board[x][y]"""

#check row on left side

**for**iy**in** range(col):

**if** board[row][iy] == 1:

**returnFalse**

ix, iy = row, col

**while** ix >= 0**and**iy>= 0:

**if** board[ix][iy] == 1:

**returnFalse**

ix-=1

iy-=1

jx, jy = row,col

**while**jx< size **and**jy>= 0:

**if** board[jx][jy] == 1:

**returnFalse**

jx+=1

jy-=1

**returnTrue**

**defsolve**(board, col, size):

"""Use backtracking to find all solutions"""

#base case

**if** col >= size:

**return**

**for**i**in** range(size):

**if**is\_safe(board, i, col, size):

board[i][col] = 1

**if** col == size-1:

add\_solution(board)

board[i][col] = 0

**return**

solve(board, col+1, size)

#backtrack

board[i][col] = 0

**defadd\_solution**(board):

"""Saves the board state to the global variable 'solutions'"""

**global** solutions

saved\_board = copy.deepcopy(board)

solutions.append(saved\_board)

size = take\_input()

board = get\_board(size)

solutions = []

solve(board, 0, size)

print\_solutions(solutions, size)

print("Total solutions = {}".format(len(solutions)))

output:



Aim: 2.b.write a program to slove tower of hanoi problem.

Procedure:- Click -> Start -> Window->All program->Python IDLE 3.4 -> ENTER

Start-> New Folder->Write a program->Save->RUN.

Code:

def moveTower(height,fromPole, toPole, withPole):

if height >= 1:

moveTower(height-1,fromPole,withPole,toPole)

moveDisk(fromPole,toPole)

moveTower(height-1,withPole,toPole,fromPole)

defssmoveDisk(fp,tp):

print("moving disk from",fp,"to",tp)

moveTower(3,"A","B","C")

output:



PRACTICAL .3

Aim: 3.a.**Write a program to implement alpha betasearch.**

Procedure:- Click -> Start -> Window->All program->Python IDLE 3.4 -> ENTER

Start-> New Folder->Write a program->Save->RUN.

Code:

tree = [[[5, 1, 2], [8, -8, -9]], [[9, 4, 5], [-3, 4, 3]]]

root = 0

pruned = 0

def children(branch, depth, alpha, beta):

global tree

global root

global pruned

i = 0

for child in branch:

if type(child) is list:

(nalpha, nbeta) = children(child, depth + 1, alpha, beta)

if depth % 2 == 1:

beta = nalpha if nalpha < beta else beta

else:

alpha = nbeta if nbeta > alpha else alpha

branch[i] = alpha if depth % 2 == 0 else beta

i += 1

else:

if depth % 2 == 0 and alpha < child:

alpha = child

if depth % 2 == 1 and beta > child:

beta = child

if alpha >= beta:

pruned += 1

break

if depth == root:

tree = alpha if root == 0 else beta

return (alpha, beta)

def alphabeta(in\_tree=tree, start=root, upper=-15, lower=15):

global tree

global pruned

global root

(alpha, beta) = children(tree, start, upper, lower)

if \_\_name\_\_ == "\_\_main\_\_":

print ("(alpha, beta): ", alpha, beta)

print ("Result: ", tree)

print ("Times pruned: ", pruned)

return (alpha, beta, tree, pruned)

if \_\_name\_\_ == "\_\_main\_\_":

alphabeta(None)

output:-



Aim: 3.b.**Write a program for Hill climbing problem.**

Procedure:- Click -> Start -> Window->All program->Python IDLE 3.4 -> ENTER

Start-> New Folder->Write a program->Save->RUN.

Code:

import math

increment = 0.1

startingPoint = [1, 1]

point1 = [1,5]

point2 = [6,4]

point3 = [5,2]

point4 = [2,1]

def distance(x1, y1, x2, y2):

dist = math.pow(x2-x1, 2) + math.pow(y2-y1, 2)

return dist

def sumOfDistances(x1, y1, px1, py1, px2, py2, px3, py3, px4, py4):

d1 = distance(x1, y1, px1, py1)

d2 = distance(x1, y1, px2, py2)

d3 = distance(x1, y1, px3, py3)

d4 = distance(x1, y1, px4, py4)

return d1 + d2 + d3 + d4

def newDistance(x1, y1, point1, point2, point3, point4):

d1 = [x1, y1]

d1temp = sumOfDistances(x1, y1, point1[0],point1[1], point2[0],point2[1], point3[0],point3[1], point4[0],point4[1] )

d1.append(d1temp)

return d1

minDistance = sumOfDistances(startingPoint[0], startingPoint[1],point1[0],point1[1], point2[0],point2[1], point3[0],point3[1], point4[0],point4[1] )

flag = True

def newPoints(minimum, d1, d2, d3, d4):

if d1[2] == minimum:

return [d1[0], d1[1]]

elif d2[2] == minimum:

return [d2[0], d2[1]]

elif d3[2] == minimum:

return [d3[0], d3[1]]

elif d4[2] == minimum:

return [d4[0], d4[1]]

i = 1

while flag:

d1 = newDistance(startingPoint[0]+increment, startingPoint[1], point1, point2,point3, point4)

d2 = newDistance(startingPoint[0]-increment, startingPoint[1], point1, point2,point3, point4)

d3 = newDistance(startingPoint[0], startingPoint[1]+increment, point1, point2,point3, point4)

d4 = newDistance(startingPoint[0], startingPoint[1]-increment, point1, point2,point3, point4)

print (i,' ', round(startingPoint[0], 2), round(startingPoint[1], 2))

minimum = min(d1[2], d2[2], d3[2], d4[2])

if minimum < minDistance:

startingPoint = newPoints(minimum, d1, d2, d3, d4)

minDistance = minimum

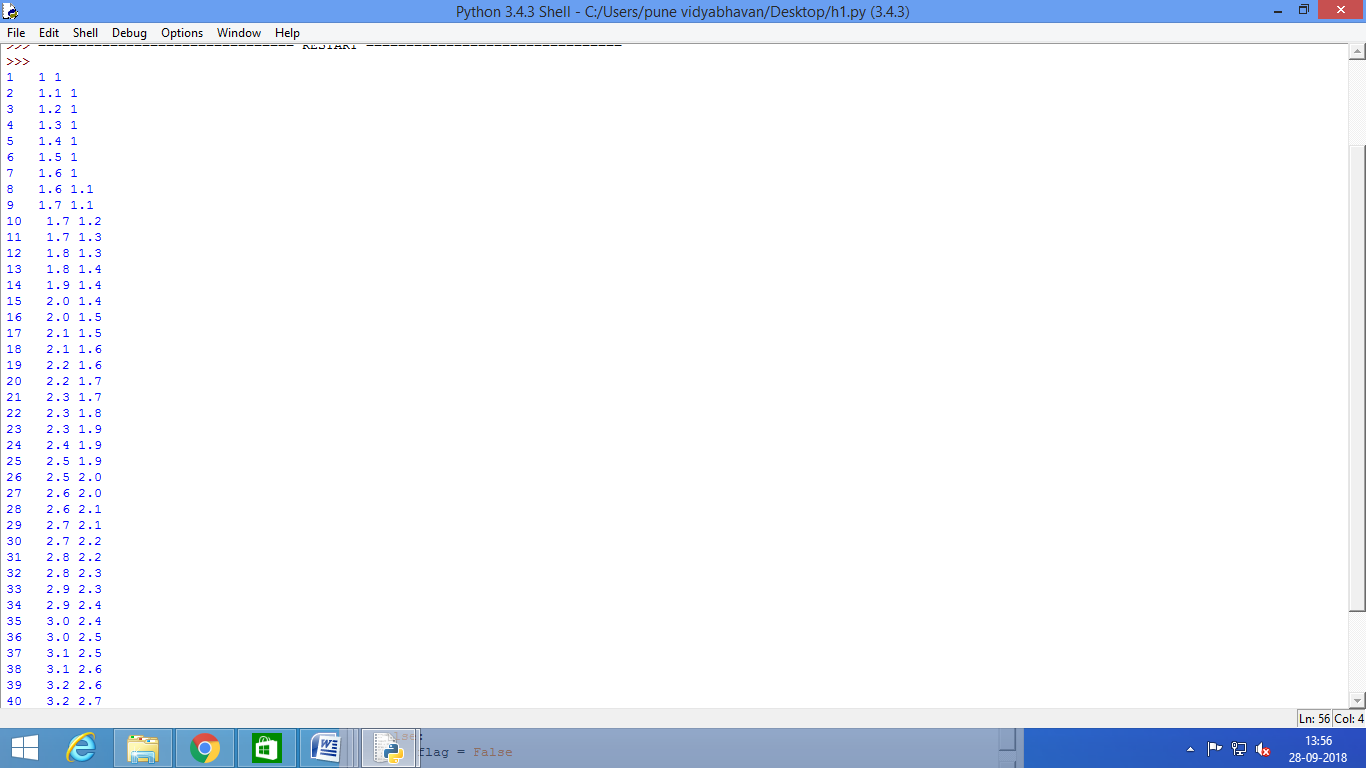
#print i,' ', round(startingPoint[0], 2), round(startingPoint[1], 2)

i+=1

else:

flag = False

output:-



PRACTICAL .4

Aim: 4.**Write a program to implement A\*algorithm.**

Procedure:- Click -> Start -> Window->All program->Python IDLE 3.4 -> ENTER

Start-> New Folder->Write a program->Save->RUN.

**Note: -**Install 2 package in python scripts directory using pip command.

1. pip install simpleai

2. pip install pydot flask

Code:

from simpleai.search import SearchProblem, astar

GOAL = 'HELLO WORLD'

class HelloProblem(SearchProblem):

def actions(self, state):

if len(state) < len(GOAL):

return list(' ABCDEFGHIJKLMNOPQRSTUVWXYZ')

else:

return []

def result(self, state, action):

return state + action

def is\_goal(self, state):

return state == GOAL

def heuristic(self, state):

# how far are we from the goal?

wrong = sum([1 if state[i] != GOAL[i] else 0 for i in range(len(state))])

missing = len(GOAL) - len(state)

return wrong + missing

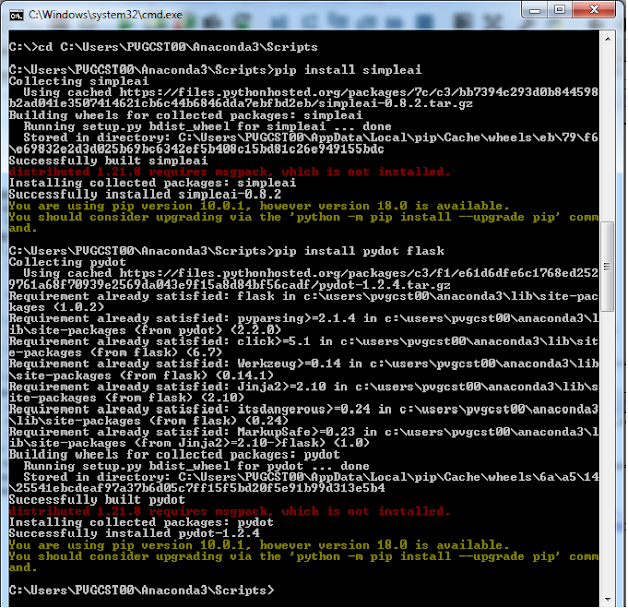
problem = HelloProblem(initial\_state='')

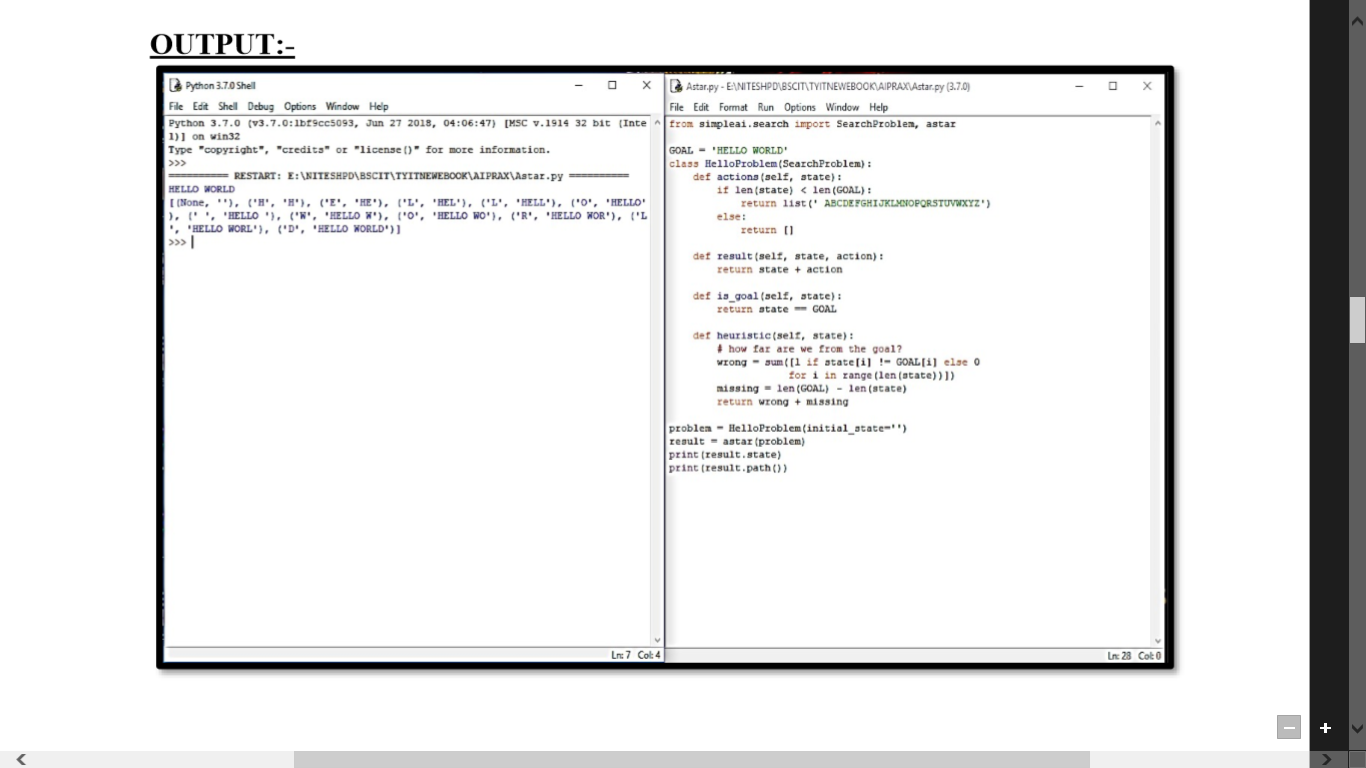
result =astar(problem)

print(result.state)

print(result.path())

Output:-





PRACTICAL .5

Aim:-5.a.**Write a program to solve water jug problem.**

Procedure:- Click -> Start -> Window->All program->Python IDLE 3.4 -> ENTER

Start-> New Folder->Write a program->Save->RUN.

Code:

capacity = (12, 8, 5)

x = capacity[0]

y = capacity[1]

z = capacity[2]

memory = {}

ans = []

def get\_all\_states(state):

a = state[0]

b = state[1]

c = state[2]

if(a==6 and b==6):

ans.append(state)

return True

if((a,b,c) in memory):

return False

memory[(a,b,c)]=1

if(a>0):

if(a+b<=y):

if( get\_all\_states((0,a+b,c)) ):

ans.append(state)

return True

else:

if( get\_all\_states((a-(y-b), y, c)) ):

ans.append(state)

return True

if(a+c<=z):

if( get\_all\_states((0,b,a+c)) ):

ans.append(state)

return True

else:

if( get\_all\_states((a-(z-c), b, z)) ):

ans.append(state)

return True

if(b>0):

if(a+b<=x):

if( get\_all\_states((a+b, 0, c)) ):

ans.append(state)

return True

else:

if( get\_all\_states((x, b-(x-a), c)) ):

ans.append(state)

return True

if(b+c<=z):

if( get\_all\_states((a, 0, b+c)) ):

ans.append(state)

return True

else:

if( get\_all\_states((a, b-(z-c), z)) ):

ans.append(state)

return True

if(c>0):

if(a+c<=x):

if( get\_all\_states((a+c, b, 0)) ):

ans.append(state)

return True

else:

if( get\_all\_states((x, b, c-(x-a))) ):

ans.append(state)

return True

if(b+c<=y):

if( get\_all\_states((a, b+c, 0)) ):

ans.append(state)

return True

else:

if( get\_all\_states((a, y, c-(y-b))) ):

ans.append(state)

return True

return False

initial\_state = (12, 0, 0)

print ("starting work...\n")

get\_all\_states(initial\_state)

ans.reverse()

for i in ans:

print(i)

Output:-



Aim:-5.b.**Design the simulation of tic – tac – toe game using min-max algorithm.**

Procedure:- Click -> Start -> Window->All program->Python IDLE 3.4 -> ENTER

Start-> New Folder->Write a program->Save->RUN.

Code:

import os

import time

board = [' ',' ',' ',' ',' ',' ',' ',' ',' ',' ']

player = 1

########win Flags##########

Win = 1

Draw = -1

Running = 0

Stop = 1

###########################

Game = Running

Mark = 'X'

#This Function Draws Game Board

def DrawBoard():

print(" %c | %c | %c " % (board[1],board[2],board[3]))

print("\_\_\_|\_\_\_|\_\_\_")

print(" %c | %c | %c " % (board[4],board[5],board[6]))

print("\_\_\_|\_\_\_|\_\_\_")

print(" %c | %c | %c " % (board[7],board[8],board[9]))

print(" | | ")

#This Function Checks position is empty or not

def CheckPosition(x):

if(board[x] == ' '):

return True

else:

return False

#This Function Checks player has won or not

def CheckWin():

global Game

#Horizontal winning condition

if(board[1] == board[2] and board[2] == board[3] and board[1] != ' '):

Game = Win

elif(board[4] == board[5] and board[5] == board[6] and board[4] != ' '):

Game = Win

elif(board[7] == board[8] and board[8] == board[9] and board[7] != ' '):

Game = Win

#Vertical Winning Condition

elif(board[1] == board[4] and board[4] == board[7] and board[1] != ' '):

Game = Win

elif(board[2] == board[5] and board[5] == board[8] and board[2] != ' '):

Game = Win

elif(board[3] == board[6] and board[6] == board[9] and board[3] != ' '):

Game=Win

#Diagonal Winning Condition

elif(board[1] == board[5] and board[5] == board[9] and board[5] != ' '):

Game = Win

elif(board[3] == board[5] and board[5] == board[7] and board[5] != ' '):

Game=Win

#Match Tie or Draw Condition

elif(board[1]!=' ' and board[2]!=' ' and board[3]!=' ' and board[4]!=' ' and

board[5]!=' ' and board[6]!=' ' and board[7]!=' ' and board[8]!=' ' and board[9]!=' '):

Game=Draw

else:

Game=Running

print("Tic-Tac-Toe Game")

print("Player 1 [X] --- Player 2 [O]\n")

print()

print("Please Wait...")

time.sleep(1)

while(Game == Running):

os.system('cls')

DrawBoard()

if(player % 2 != 0):

print("Player 1's chance")

Mark = 'X'

else:

print("Player 2's chance")

Mark = 'O'

choice = int(input("Enter the position between [1-9] where you want to mark :"))

if(CheckPosition(choice)):

board[choice] = Mark

player+=1

CheckWin()

os.system('cls')

DrawBoard()

if(Game==Draw):

print("Game Draw")

elif(Game==Win):

player-=1

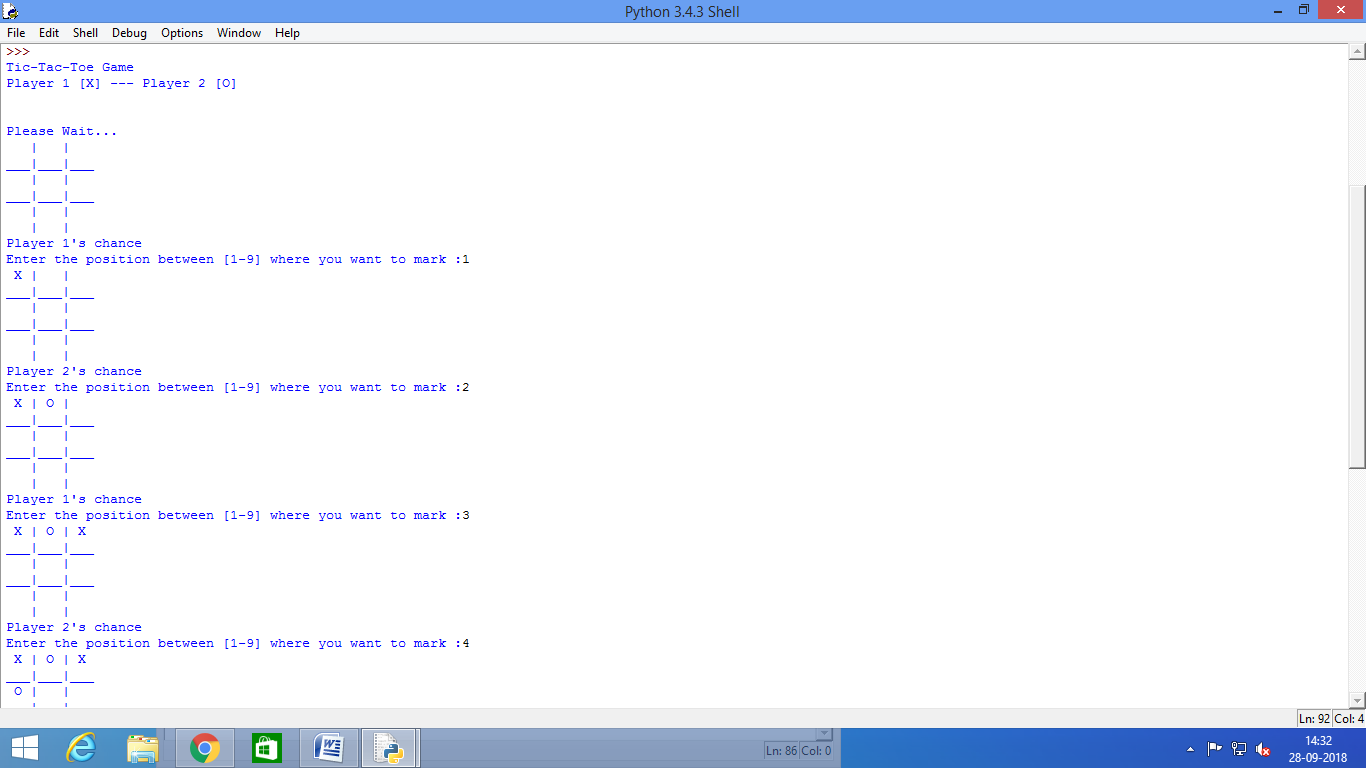
if(player%2!=0):

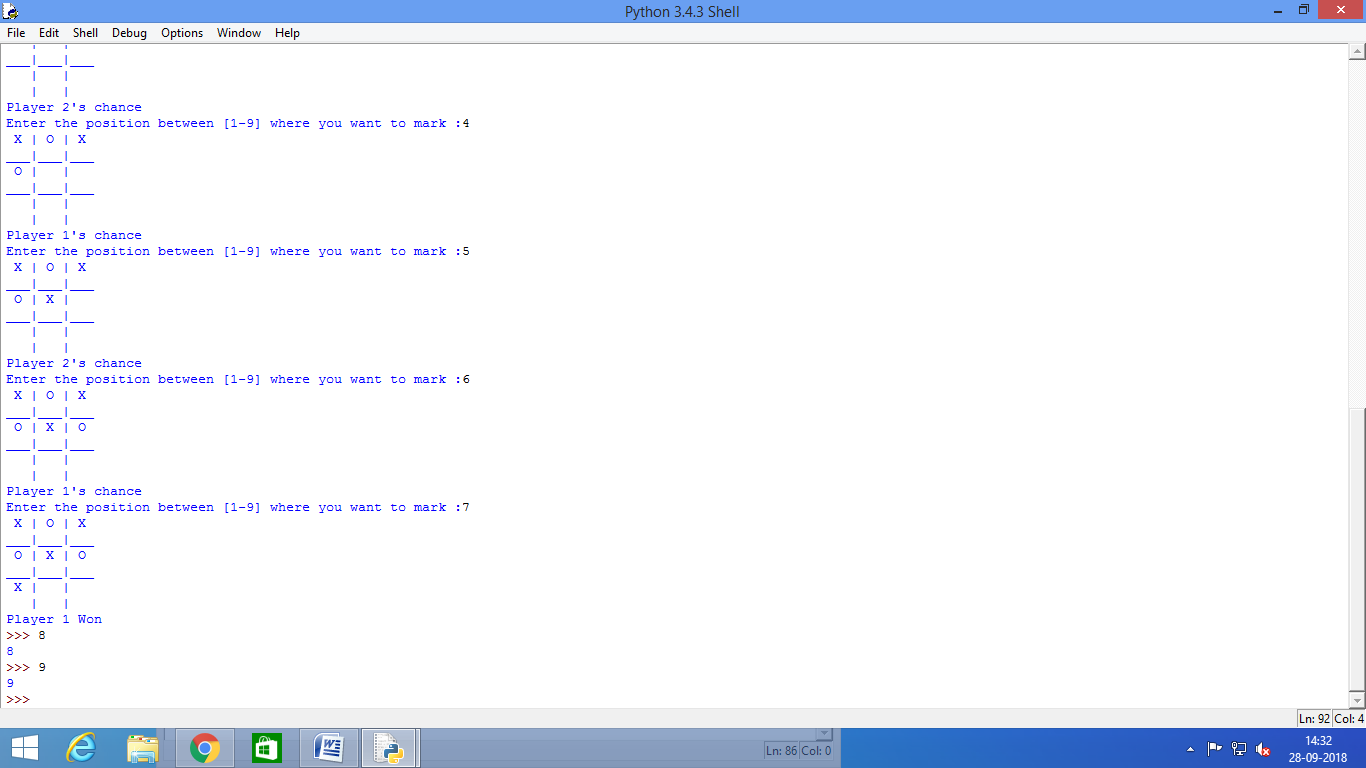
print("Player 1 Won")

else:

print("Player 2 Won")

output:-





PRACTICAL .6

Aim: 6.a Write a program to solve Missionaries and Cannibals problem.

Procedure:- Click -> Start -> Window->All program->Python IDLE 3.4 -> ENTER

Start-> New Folder->Write a program->Save->RUN.

Code:

import math

#Missionaries and Cannibals Problem

class State():

def \_\_init\_\_(self, cannibalLeft, missionaryLeft, boat,cannibalRight, missionaryRight):

self.cannibalLeft = cannibalLeft

self.missionaryLeft = missionaryLeft

self.boat = boat

self.cannibalRight = cannibalRight

self.missionaryRight = missionaryRight

self.parent = None

def is\_goal(self):

if self.cannibalLeft == 0 and self.missionaryLeft == 0:

return True

else:

return False

def is\_valid(self):

if self.missionaryLeft >= 0 and self.missionaryRight >= 0 \

and self.cannibalLeft >= 0 and self.cannibalRight >= 0 \

and (self.missionaryLeft == 0 or self.missionaryLeft >= self.cannibalLeft) \

and (self.missionaryRight == 0 or self.missionaryRight >= self.cannibalRight):

return True

else:

return False

def \_\_eq\_\_(self, other):

return self.cannibalLeft == other.cannibalLeft and self.missionaryLeft == other.missionaryLeft \

and self.boat == other.boat and self.cannibalRight == other.cannibalRight \

and self.missionaryRight == other.missionaryRight

def \_\_hash\_\_(self):

return hash((self.cannibalLeft, self.missionaryLeft, self.boat,self.cannibalRight, self.missionaryRight))

def successors(cur\_state):

children = [];

if cur\_state.boat == 'left':

new\_state = State(cur\_state.cannibalLeft, cur\_state.missionaryLeft - 2, 'right',

cur\_state.cannibalRight, cur\_state.missionaryRight + 2)

## Two missionaries cross left to right.

if new\_state.is\_valid():

new\_state.parent = cur\_state

children.append(new\_state)

new\_state = State(cur\_state.cannibalLeft - 2, cur\_state.missionaryLeft, 'right',

cur\_state.cannibalRight + 2, cur\_state.missionaryRight)

## Two cannibals cross left to right.

if new\_state.is\_valid():

new\_state.parent = cur\_state

children.append(new\_state)

new\_state = State(cur\_state.cannibalLeft - 1, cur\_state.missionaryLeft - 1, 'right',

cur\_state.cannibalRight + 1, cur\_state.missionaryRight + 1)

## One missionary and one cannibal cross left to right.

if new\_state.is\_valid():

new\_state.parent = cur\_state

children.append(new\_state)

new\_state = State(cur\_state.cannibalLeft, cur\_state.missionaryLeft - 1, 'right',

cur\_state.cannibalRight, cur\_state.missionaryRight + 1)

## One missionary crosses left to right.

if new\_state.is\_valid():

new\_state.parent = cur\_state

children.append(new\_state)

new\_state = State(cur\_state.cannibalLeft - 1, cur\_state.missionaryLeft, 'right',

cur\_state.cannibalRight + 1, cur\_state.missionaryRight)

## One cannibal crosses left to right.

if new\_state.is\_valid():

new\_state.parent = cur\_state

children.append(new\_state)

else:

new\_state = State(cur\_state.cannibalLeft, cur\_state.missionaryLeft + 2, 'left',

cur\_state.cannibalRight, cur\_state.missionaryRight - 2)

## Two missionaries cross right to left.

if new\_state.is\_valid():

new\_state.parent = cur\_state

children.append(new\_state)

new\_state = State(cur\_state.cannibalLeft + 2, cur\_state.missionaryLeft, 'left',

cur\_state.cannibalRight - 2, cur\_state.missionaryRight)

## Two cannibals cross right to left.

if new\_state.is\_valid():

new\_state.parent = cur\_state

children.append(new\_state)

new\_state = State(cur\_state.cannibalLeft + 1, cur\_state.missionaryLeft + 1, 'left',

cur\_state.cannibalRight - 1, cur\_state.missionaryRight - 1)

## One missionary and one cannibal cross right to left.

if new\_state.is\_valid():

new\_state.parent = cur\_state

children.append(new\_state)

new\_state = State(cur\_state.cannibalLeft, cur\_state.missionaryLeft + 1, 'left',

cur\_state.cannibalRight, cur\_state.missionaryRight - 1)

## One missionary crosses right to left.

if new\_state.is\_valid():

new\_state.parent = cur\_state

children.append(new\_state)

new\_state = State(cur\_state.cannibalLeft + 1, cur\_state.missionaryLeft, 'left',

cur\_state.cannibalRight - 1, cur\_state.missionaryRight)

## One cannibal crosses right to left.

if new\_state.is\_valid():

new\_state.parent = cur\_state

children.append(new\_state)

return children

def breadth\_first\_search():

initial\_state = State(3,3,'left',0,0)

if initial\_state.is\_goal():

return initial\_state

frontier = list()

explored = set()

frontier.append(initial\_state)

while frontier:

state = frontier.pop(0)

if state.is\_goal():

return state

explored.add(state)

children = successors(state)

for child in children:

if (child not in explored) or (child not in frontier):

frontier.append(child)

return None

def print\_solution(solution):

path = []

path.append(solution)

parent = solution.parent

while parent:

path.append(parent)

parent = parent.parent

for t in range(len(path)):

state = path[len(path) - t - 1]

print ("(" + str(state.cannibalLeft) + "," + str(state.missionaryLeft) \

+ "," + state.boat + "," + str(state.cannibalRight) + "," + \

str(state.missionaryRight) + ")")

def main():

solution = breadth\_first\_search()

print ("Missionaries and Cannibals solution:")

print ("(cannibalLeft,missionaryLeft,boat,cannibalRight,missionaryRight)")

print\_solution(solution)

# if called from the command line, call main()

if \_\_name\_\_ == "\_\_main\_\_":

main()

output:-



Aim: 6.bDesign an application to simulate number puzzle problem.

Procedure:- Click -> Start -> Window->All program->Python IDLE 3.4 -> ENTER

Start-> New Folder->Write a program->Save->RUN.

Code:

'''

8 puzzle problem, a smaller version of the fifteen puzzle:

States are defined as string representations of the pieces on the puzzle.

Actions denote what piece will be moved to the empty space.

States must allways be inmutable. We will use strings, but internally most of

the time we will convert those strings to lists, which are easier to handle.

For example, the state (string):

'1-2-3

4-5-6

7-8-e'

will become (in lists):

[['1', '2', '3'],

['4', '5', '6'],

['7', '8', 'e']]

'''

from \_\_future\_\_ import print\_function

from simpleai.search import astar, SearchProblem

from simpleai.search.viewers import WebViewer

GOAL = '''1-2-3

4-5-6

7-8-e'''

INITIAL = '''4-1-2

7-e-3

8-5-6'''

def list\_to\_string(list\_):

return '\n'.join(['-'.join(row) for row in list\_])

def string\_to\_list(string\_):

return [row.split('-') for row in string\_.split('\n')]

def find\_location(rows, element\_to\_find):

'''Find the location of a piece in the puzzle.

Returns a tuple: row, column'''

for ir, row in enumerate(rows):

for ic, element in enumerate(row):

if element == element\_to\_find:

return ir, ic

# we create a cache for the goal position of each piece, so we don't have to

# recalculate them every time

goal\_positions = {}

rows\_goal = string\_to\_list(GOAL)

for number in '12345678e':

goal\_positions[number] = find\_location(rows\_goal, number)

class EigthPuzzleProblem(SearchProblem):

def actions(self, state):

'''Returns a list of the pieces we can move to the empty space.'''

rows = string\_to\_list(state)

row\_e, col\_e = find\_location(rows, 'e')

actions = []

if row\_e > 0:

actions.append(rows[row\_e - 1][col\_e])

if row\_e < 2:

actions.append(rows[row\_e + 1][col\_e])

if col\_e > 0:

actions.append(rows[row\_e][col\_e - 1])

if col\_e < 2:

actions.append(rows[row\_e][col\_e + 1])

return actions

def result(self, state, action):

'''Return the resulting state after moving a piece to the empty space.

(the "action" parameter contains the piece to move)

'''

rows = string\_to\_list(state)

row\_e, col\_e = find\_location(rows, 'e')

row\_n,col\_n = find\_location(rows, action)

rows[row\_e][col\_e], rows[row\_n][col\_n] = rows[row\_n][col\_n], rows[row\_e][col\_e]

return list\_to\_string(rows)

def is\_goal(self, state):

'''Returns true if a state is the goal state.'''

return state == GOAL

def cost(self, state1, action, state2):

'''Returns the cost of performing an action. No useful on this problem, i

but needed.

'''

return 1

def heuristic(self, state):

'''Returns an \*estimation\* of the distance from a state to the goal.

We are using the manhattan distance.

'''

rows = string\_to\_list(state)

distance = 0

for number in '12345678e':

row\_n, col\_n = find\_location(rows, number)

row\_n\_goal, col\_n\_goal = goal\_positions[number]

distance += abs(row\_n - row\_n\_goal) + abs(col\_n - col\_n\_goal)

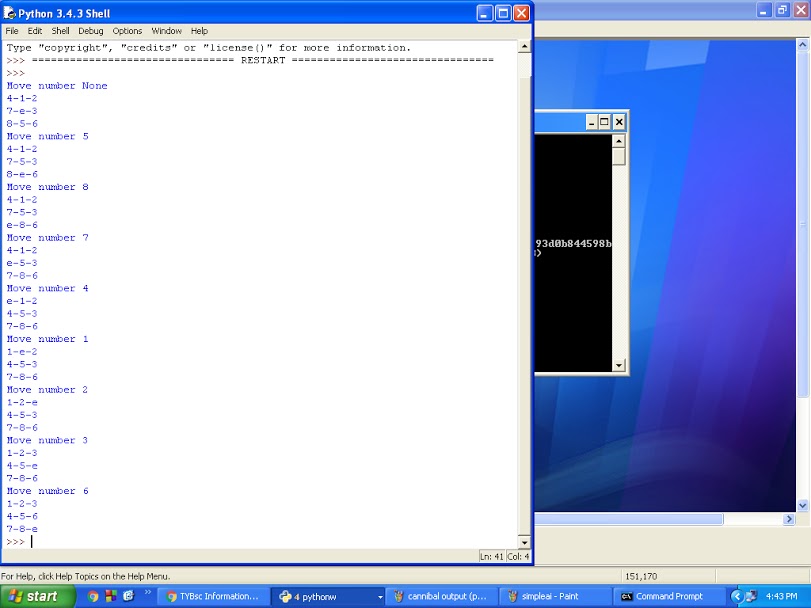
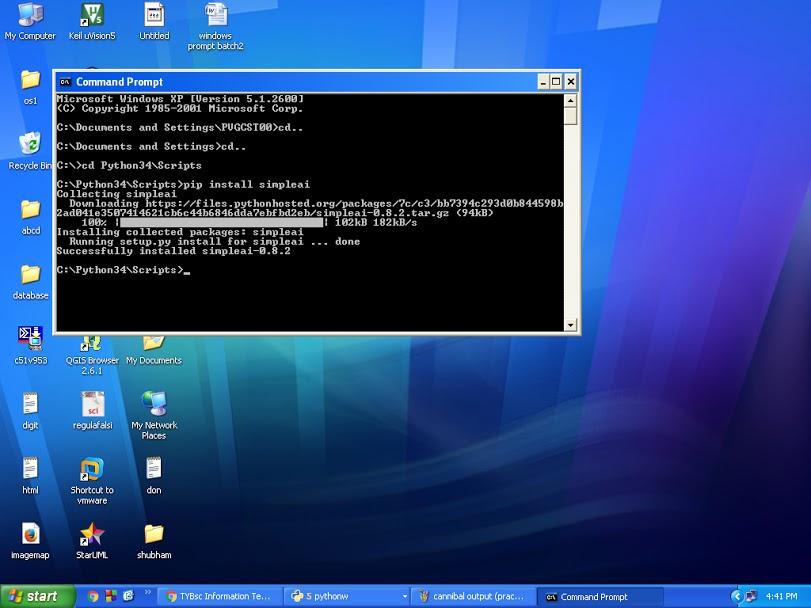
return distance

result = astar(EigthPuzzleProblem(INITIAL))

for action, state in result.path():

print('Move number', action)

print(state)

output:

PRACTICAL .7

Aim: 7.a Write a program to shuffle Deck of cards.

Procedure:- Click -> Start -> Window->All program->Python IDLE 3.4 -> ENTER

Start-> New Folder->Write a program->Save->RUN.

Code:

# Python program to shuffle a deck of card using the module random and draw 5 cards

# import modules

import itertools, random

# make a deck of cards

deck = list(itertools.product(range(1,14),['Spade','Heart','Diamond','Club']))

# shuffle the cards

random.shuffle(deck)

# draw five cards

print("You got:")

for i in range(5):

print(deck[i][0], "of", deck[i][1])

output:-



PRACTICAL .8

8.solve constraint satisfaction problem.

Procedure:- Click -> Start -> Window->All program->Python IDLE 3.4 -> ENTER

Start-> New Folder->Write a program->Save->RUN.

Code:

from \_\_future\_\_ import print\_function

from simpleai.search import CspProblem, backtrack, min\_conflicts,MOST\_CONSTRAINED\_VARIABLE, HIGHEST\_DEGREE\_VARIABLE,LEAST\_CONSTRAINING\_VALUE

variables = ('WA', 'NT', 'SA', 'Q', 'NSW', 'V', 'T')

domains = dict((v, ['red', 'green', 'blue']) for v in variables)

def const\_different(variables, values):

return values[0] != values[1]# expect the value of the neighbors to be different

constraints = [

(('WA', 'NT'), const\_different),

(('WA', 'SA'), const\_different),

(('SA', 'NT'), const\_different),

(('SA', 'Q'), const\_different),

(('NT', 'Q'), const\_different),

(('SA', 'NSW'), const\_different),

(('Q', 'NSW'), const\_different),

(('SA', 'V'), const\_different),

(('NSW', 'V'), const\_different),

]

my\_problem = CspProblem(variables, domains, constraints)

print(backtrack(my\_problem))

print(backtrack(my\_problem,variable\_heuristic=MOST\_CONSTRAINED\_VARIABLE))

print(backtrack(my\_problem,variable\_heuristic=HIGHEST\_DEGREE\_VARIABLE))

print(backtrack(my\_problem,value\_heuristic=LEAST\_CONSTRAINING\_VALUE))

print(backtrack(my\_problem,variable\_heuristic=MOST\_CONSTRAINED\_VARIABLE,value\_heuristic=LEAST\_CONSTRAINING\_VALUE))

print(backtrack(my\_problem,variable\_heuristic=HIGHEST\_DEGREE\_VARIABLE, value\_heuristic=LEAST\_CONSTRAINING\_VALUE))

print(min\_conflicts(my\_problem))

**OUTPUT:-**

