Name: - Tanishq Parab

Roll No: -A033

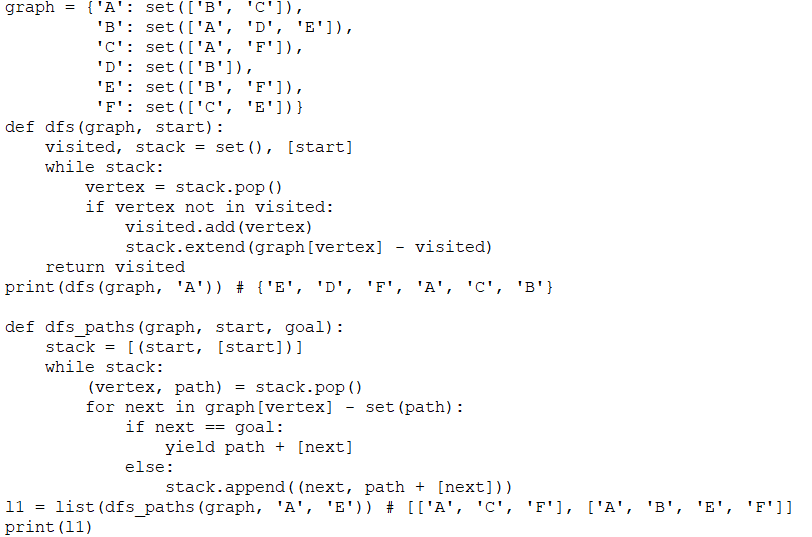
TYBSC IT

Artificial Intelligence Practical

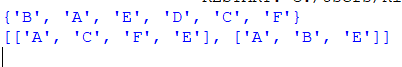
|  |  |  |  |
| --- | --- | --- | --- |
| Serial No | Details | Date | Sign |
| 1 | 1. a. Write a program to implement depth first search algorithm. |  |  |
| 2 | 1.b. Write a program to implement breadth first search algorithm. |  |  |
| 3 | 2.a. Write a program to simulate 4-Queen / N-Queen problem. |  |  |
| 4 | 2.b. Write a program to solve tower of Hanoi problem. |  |  |
| 5 | 5. a. Write a program to solve water jug problem. |  |  |
| 6 | 5.b. Design the simulation of tic – tac – toe game using min-max algorithm |  |  |
| 7 | 6. a. Write a program to solve Missionaries and Cannibals problem |  |  |
| 8 | 6.b Design an application to simulate number puzzle problem. |  |  |
| 9 | 7. a. Write a program to shuffle Deck of cards. |  |  |
| 10 | 8. Solve constraint satisfaction problem  d. Magic Squares |  |  |
|  |  |  |  |

Practical 1

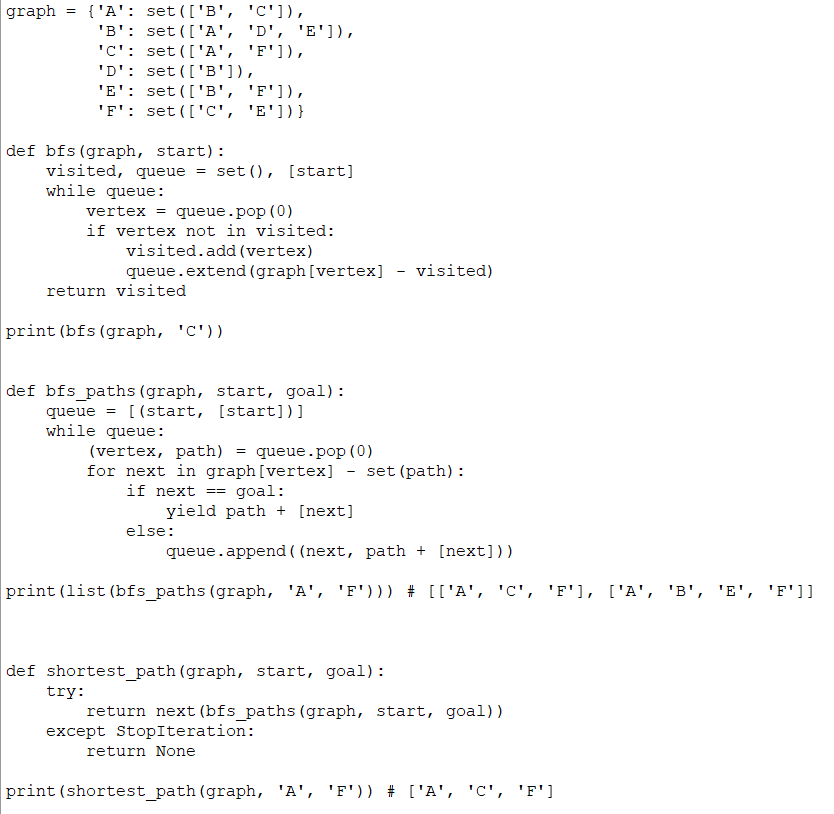
1. a. Write a program to implement depth first search algorithm.



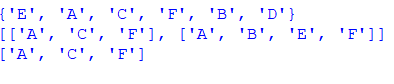
Output: -



1. b. Write a program to implement breadth first search algorithm.



Output: -



Practical 2

1. a. Write a program to simulate 4-Queen / N-Queen problem.

N = 4

def print\_solution(board):

for row in board:

print(" ".join(str(x) for x in row))

print()

def is\_safe(board, row, col):

# Check this row on left side

for i in range(col):

if board[row][i] == 1:

return False

for i, j in zip(range(row, -1, -1), range(col, -1, -1)):

if board[i][j] == 1:

return False

for i, j in zip(range(row, N), range(col, -1, -1)):

if board[i][j] == 1:

return False

return True

def solve\_nq\_util(board, col):

if col >= N:

print\_solution(board)

return True

res = False

for i in range(N):

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

board[i][col] = 1

res = solve\_nq\_util(board, col + 1) or res

board[i][col] = 0 # Backtrack

return res

def solve\_nq():

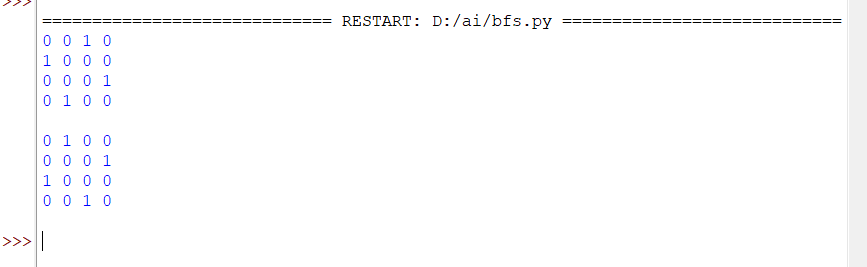
board = [[0] \* N for \_ in range(N)]

if not solve\_nq\_util(board, 0):

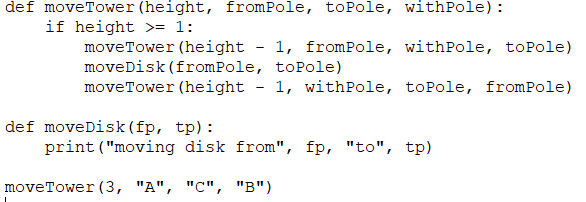
print("Solution does not exist")

solve\_nq()

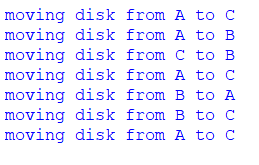
Output: -



2) b. Write a program to solve tower of Hanoi problem.



Output: -



Practical 5

5. a. Write a program to solve water jug problem.

# 3 water jugs capacity -> (x,y,z) where x>y>z

# initial state (12,0,0)

# final state (6,5,1)

capacity = (12,8,5)

# Maximum capacities of 3 jugs -> x,y,z

x = capacity[0]

y = capacity[1]

z = capacity[2]

# to mark visited states memory is a dictionarycontaining key value pair.

memory = {}

# store solution path

ans = []

def get\_all\_states(state):

# Let the 3 jugs be called a,b,c

a = state[0]

b = state[1]

c = state[2]

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

ans.append(state)

return True

# if current state is already visited earlier

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

return False

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

#empty jug a

if(a>0):

#empty a into b

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

#empty a into c

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

#empty jug b

if(b>0):

#empty b into a

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

#empty b into c

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

#empty jug c

if(c>0):

#empty c into a

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

#empty c into b

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")

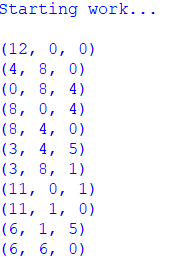
get\_all\_states(initial\_state)

ans.reverse()

for i in ans:

print(i)

Output: -



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

import os

import time

board = [' ' for \_ in range(10)]

player = 1

win = 1

draw = -1

running = 0

stop = 1

game = running

def drawboard():

print(f"{board[1]} |{board[2]} |{board[3]}")

print("---+---+---")

print(f"{board[4]} |{board[5]} |{board[6]}")

print("---+---+---")

print(f"{board[7]} |{board[8]} |{board[9]}")

print("---+---+---")

def checkposition(x):

return board[x] == ' '

def checkwin():

global game

if (board[1] == board[2] == board[3] != ' ' or

board[4] == board[5] == board[6] != ' ' or

board[7] == board[8] == board[9] != ' ' or

board[1] == board[4] == board[7] != ' ' or

board[2] == board[5] == board[8] != ' ' or

board[3] == board[6] == board[9] != ' ' or

board[1] == board[5] == board[9] != ' ' or

board[3] == board[5] == board[7] != ' '):

game = win

elif all(space != ' ' for space in board[1:]):

game = draw

print("Tic-tac-toe game")

print("Please wait...")

time.sleep(1)

while game == running:

os.system('cls' if os.name == 'nt' else 'clear')

drawboard()

if player % 2 != 0:

print("Player 1's turn")

mark = 'X'

else:

print("Player 2's turn")

mark = 'O'

try:

choice = int(input("Enter position [1-9]: "))

if choice < 1 or choice > 9:

print("Invalid position, try again.")

time.sleep(1)

continue

if checkposition(choice):

board[choice] = mark

player += 1

checkwin()

else:

print("Position already taken!")

time.sleep(1)

except ValueError: # fixed typo

print("Please enter a valid position.")

time.sleep(1)

os.system('cls' if os.name == 'nt' else 'clear')

drawboard()

if game == draw:

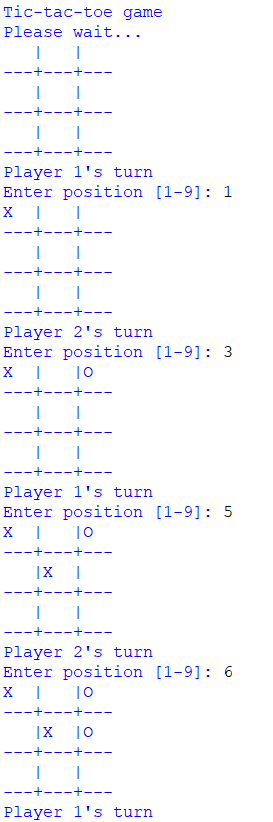
print("Game Draw!")

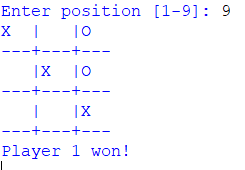
elif game == win:

winner = "Player 1 won!" if player % 2 == 0 else "Player 2 won!"

print(winner)

Output: -





Practical 6

6) a. Write a program to solve Missionaries and Cannibals problem.

import math

# Missionaries and Cannibals Problem

class State:

def \_\_init\_\_(self, cannibal\_left, missionary\_left, boat, cannibal\_right, missionary\_right):

self.cannibal\_left = cannibal\_left

self.missionary\_left = missionary\_left

self.boat = boat

self.cannibal\_right = cannibal\_right

self.missionary\_right = missionary\_right

self.parent = None

def is\_goal(self):

if self.cannibal\_left == 0 and self.missionary\_left == 0:

return True

else:

return False

def is\_valid(self):

# All counts must be non-negative

if (self.missionary\_left < 0 or self.missionary\_right < 0 or

self.cannibal\_left < 0 or self.cannibal\_right < 0):

return False

# Cannibals cannot outnumber missionaries on either bank

if (self.missionary\_left > 0 and self.missionary\_left < self.cannibal\_left):

return False

if (self.missionary\_right > 0 and self.missionary\_right < self.cannibal\_right):

return False

return True

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

return (self.cannibal\_left == other.cannibal\_left and

self.missionary\_left == other.missionary\_left and

self.boat == other.boat and

self.cannibal\_right == other.cannibal\_right and

self.missionary\_right == other.missionary\_right)

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

return hash((self.cannibal\_left, self.missionary\_left, self.boat,

self.cannibal\_right, self.missionary\_right))

def successors(self):

children = []

# Boat is on the left side

if self.boat == 'left':

# Two missionaries cross left to right

new\_state = State(self.cannibal\_left, self.missionary\_left - 2,

'right', self.cannibal\_right, self.missionary\_right + 2)

if new\_state.is\_valid():

new\_state.parent = self

children.append(new\_state)

# Two cannibals cross left to right

new\_state = State(self.cannibal\_left - 2, self.missionary\_left,

'right', self.cannibal\_right + 2, self.missionary\_right)

if new\_state.is\_valid():

new\_state.parent = self

children.append(new\_state)

# One missionary and one cannibal cross left to right

new\_state = State(self.cannibal\_left - 1, self.missionary\_left - 1,

'right', self.cannibal\_right + 1, self.missionary\_right + 1)

if new\_state.is\_valid():

new\_state.parent = self

children.append(new\_state)

# One missionary crosses left to right

new\_state = State(self.cannibal\_left, self.missionary\_left - 1,

'right', self.cannibal\_right, self.missionary\_right + 1)

if new\_state.is\_valid():

new\_state.parent = self

children.append(new\_state)

# One cannibal crosses left to right

new\_state = State(self.cannibal\_left - 1, self.missionary\_left,

'right', self.cannibal\_right + 1, self.missionary\_right)

if new\_state.is\_valid():

new\_state.parent = self

children.append(new\_state)

# Boat is on the right side

else:

# Two missionaries cross right to left

new\_state = State(self.cannibal\_left, self.missionary\_left + 2,

'left', self.cannibal\_right, self.missionary\_right - 2)

if new\_state.is\_valid():

new\_state.parent = self

children.append(new\_state)

# Two cannibals cross right to left

new\_state = State(self.cannibal\_left + 2, self.missionary\_left,

'left', self.cannibal\_right - 2, self.missionary\_right)

if new\_state.is\_valid():

new\_state.parent = self

children.append(new\_state)

# One missionary and one cannibal cross right to left

new\_state = State(self.cannibal\_left + 1, self.missionary\_left + 1,

'left', self.cannibal\_right - 1, self.missionary\_right - 1)

if new\_state.is\_valid():

new\_state.parent = self

children.append(new\_state)

# One missionary crosses right to left

new\_state = State(self.cannibal\_left, self.missionary\_left + 1,

'left', self.cannibal\_right, self.missionary\_right - 1)

if new\_state.is\_valid():

new\_state.parent = self

children.append(new\_state)

# One cannibal crosses right to left

new\_state = State(self.cannibal\_left + 1, self.missionary\_left,

'left', self.cannibal\_right - 1, self.missionary\_right)

if new\_state.is\_valid():

new\_state.parent = self

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 = [initial\_state]

explored = {initial\_state}

while frontier:

current\_state = frontier.pop(0)

if current\_state.is\_goal():

return current\_state

children = current\_state.successors()

for child in children:

if child not in explored:

explored.add(child)

frontier.append(child)

return None

def print\_solution(solution):

path = []

parent = solution

while parent:

path.append(parent)

parent = parent.parent

# Print the path from start to finish

for i in range(len(path) - 1, -1, -1):

state = path[i]

print(f"CannibalLeft: {state.cannibal\_left}, MissionaryLeft: {state.missionary\_left}, "

f"Boat: {state.boat}, CannibalRight: {state.cannibal\_right}, "

f"MissionaryRight: {state.missionary\_right}")

def main():

solution = breadth\_first\_search()

if solution:

print("Missionaries and Cannibal Solution Found!")

print\_solution(solution)

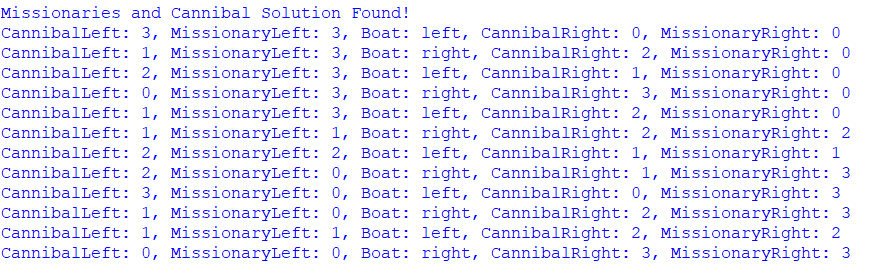
else:

print("No solution found.")

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

main()

Output: -



6) b. Design an application to simulate number puzzle problem.

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

from simpleai.search import astar, SearchProblem

GOAL = '''1-2-3

4-5-6

7-8-e'''

INITIAL = '''4-1-2

7-3-e

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 (row, col) of a piece in the puzzle."""

for ir, row in enumerate(rows):

for ic, element in enumerate(row):

if element == element\_to\_find:

return ir, ic

# Precompute goal positions

goal\_positions = {}

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

for number in '12345678e':

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

# Problem Definition

class EightPuzzleProblem(SearchProblem):

def actions(self, state):

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):

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):

return state == GOAL

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

return 1 # Each move has same cost

def heuristic(self, state):

""" Manhattan distance heuristic """

rows = string\_to\_list(state)

distance = 0

for number in '12345678e':

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

row\_g, col\_g = goal\_positions[number]

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

return distance

result = astar(EightPuzzleProblem(INITIAL))

# Print solution path

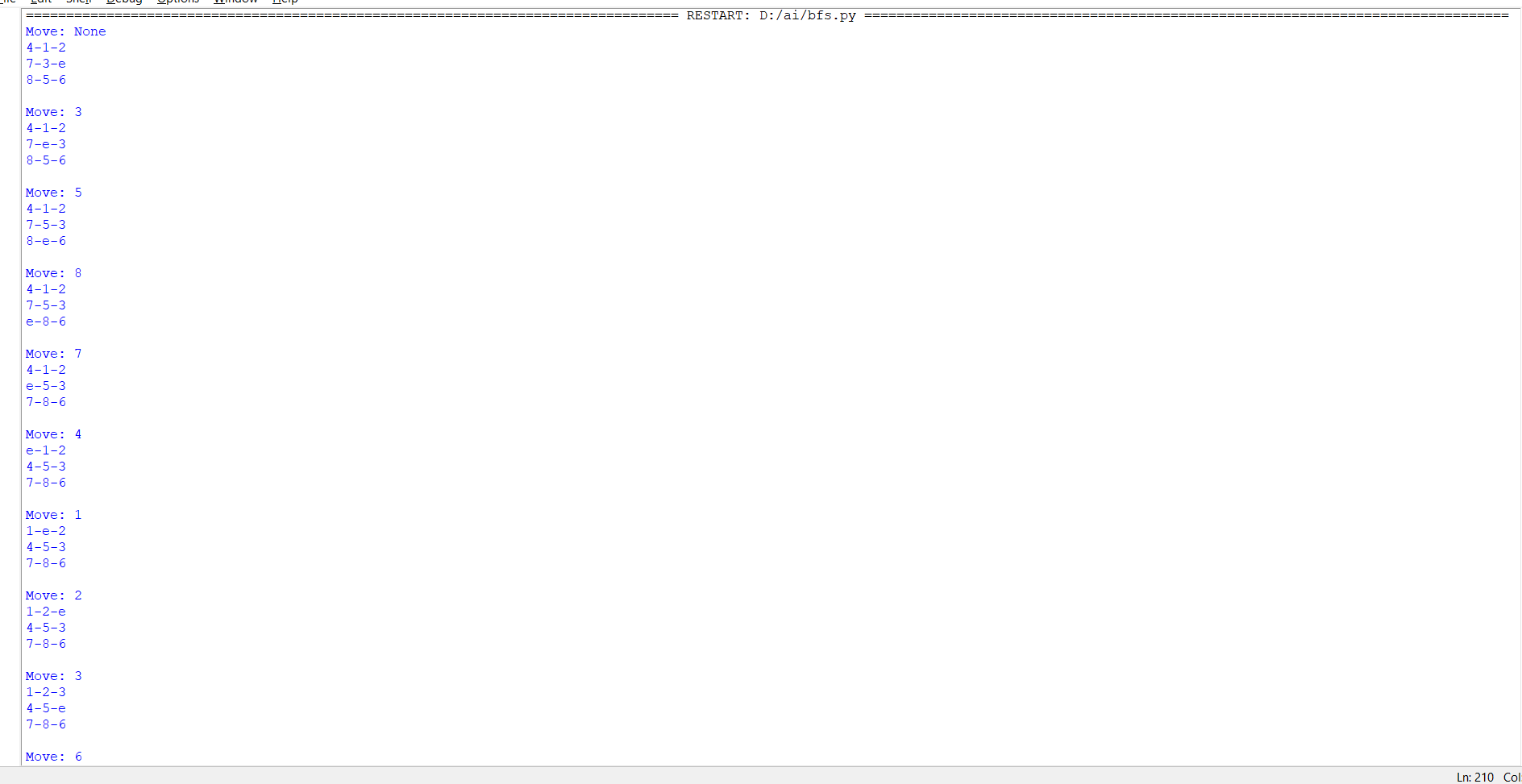
for action, state in result.path():

print("Move:", action)

print(state)

print()

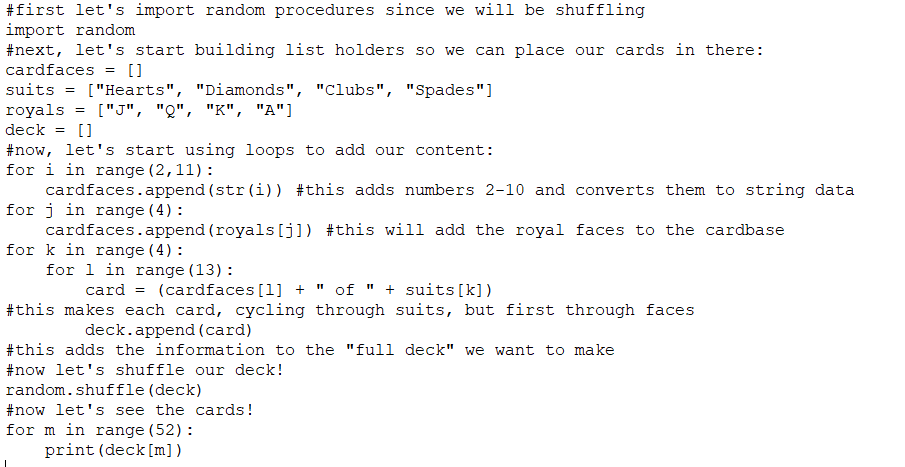
Output: -





Practical 7

7) a. Write a program to shuffle Deck of cards.



Output: -





Practical 8

8. Solve constraint satisfaction problem

a. Sudoku Solving using CSP

b. Map Coloring

c. Zebra Puzzle

d. Magic Squares

# A function to generate odd sized magic squares

def generate\_square(n):

# initialize magic square

mat = [[0] \* n for \_ in range(n)]

# Initialize position for 1

i = n // 2

j = n - 1

# One by one put all values in magic square

for num in range(1, n \* n + 1):

# if row is -1 and column becomes n,

# set row = 0, col = n -2

if i == -1 and j == n:

j = n - 2

i = 0

else:

# If next number goes to out of

# square's right side

if j == n:

j = 0

# If next number goes to out of

# square's upper side

if i < 0:

i = n - 1

# If number is already present decrement

# column by 2, and increment row by 1

if mat[i][j]:

j -= 2

i += 1

continue

else:

# set number

mat[i][j] = num

# increment and decrement

# column and row by 1 respectively

j += 1

i -= 1

return mat

n = 5

magic\_square = generate\_square(n)

for row in magic\_square:

print(" ".join(map(str, row)))

Output: -

