INDEX:

SR.NO	NAME OF THE PROGRAM	DATE	SIGNATURE
	PRACTICAL NO. 1		
a.	Write a program to implement depth first search algorithm.		
b.	Write a program to implement breadth first search algorithm		
	PRACTICAL NO. 2		
a.	Write a program to simulate 4-Queen / N-Queen problem.		
b.	Write a program to solve tower of Hanoi problem		
	PRACTICAL NO.3		
a.	Write a program to implement alpha beta search.		
b.	Write a program for Hill climbing problem.		
	Practical no.4 Write a program to implement A* algorithm.		
	Practical no-5 Write a program to solve water jug problem.		

	PRACTICAL No.6 Design an application to simulate number puzzle problem.		
	PRACTICAL No.7 Write a program to shuffle Deck of cards.		
	PRACTICAL No.8 Solve constraint satisfaction problem		

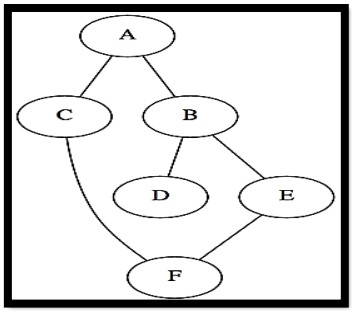
PRACTICAL NO-1

A. **Write a program to implement a depth first search algorithm.**
B. **Write a program to implement breadth first search algorithm**

AIM: -

Write a program to implement a depth first search algorithm.

GRAPH:-



Python code :

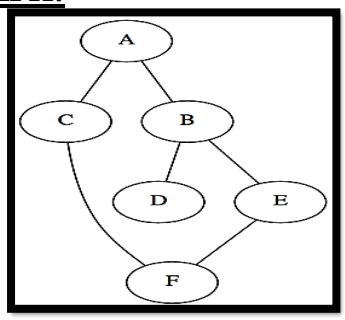
```
graph1 = {
    'A': set(['B', 'C']),
    'B': set(['A', 'D', 'E']),
    'C': set(['A', 'F']),
    'D': set(['B']),
    'E': set(['B', 'F']),
    'F': set(['C', 'E'])
}
def dfs(graph, node, visited):
    if node not in visited:
        visited.append(node)
        for n in graph[node]:
            dfs(graph, n, visited)
    return visited
visited = dfs(graph1, 'A', [])
print(visited)
```

OUTPUT

AIM:-

Write a program to implement breadth first search algorithm.

GRAPH:

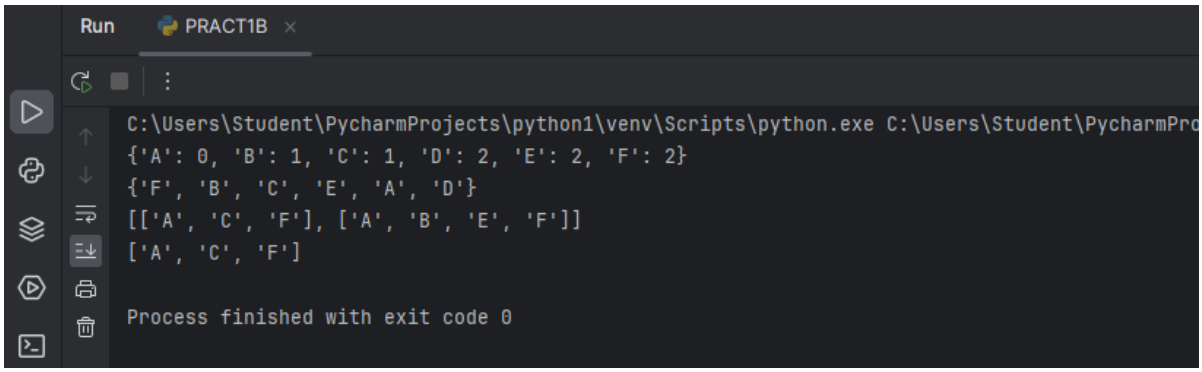


Python code :

```
graph = {'A': set(['B', 'C']),
         'B': set(['A', 'D', 'E']),
         'C': set(['A', 'F']),
         'D': set(['B']),
         'E': set(['B', 'F']),
         'F': set(['C', 'E'])
         }
def bfs(start):
    queue = [start]
    levels = {}
    levels[start] = 0
    visited = set(start)
    while queue:
        node = queue.pop(0)
        neighbours = graph[node]
        for neighbor in neighbours:
            if neighbor not in visited:
                queue.append(neighbor)
                visited.add(neighbor)
                levels[neighbor] = levels[node] + 1
    print(levels) # print graph level
    return visited
print(str(bfs('A'))) # print graph node
def bfs_paths(graph, start, goal):
    queue = [(start, [start])]
    while queue:
          (vertex, path) = queue.pop(0)
          for next in graph[vertex] - set(path):
              if next == goal:
                 yield path + [next]
              else:
                  queue.append((next, path + [next]))
result = list(bfs_paths(graph, 'A', 'F'))
print(result)
def shortest_path(graph, start, goal):
    try:
        return next(bfs_paths(graph, start, goal))
    except StopIteration:
```

```
            return None
result1 = shortest_path(graph, 'A', 'F')
print(result1)  # ['A', 'C', 'F']
```

OUTPUT:



Practical no-2

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

B. Write a program to solve tower of Hanoi problem.

Aim: -

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

DIAGRAM:

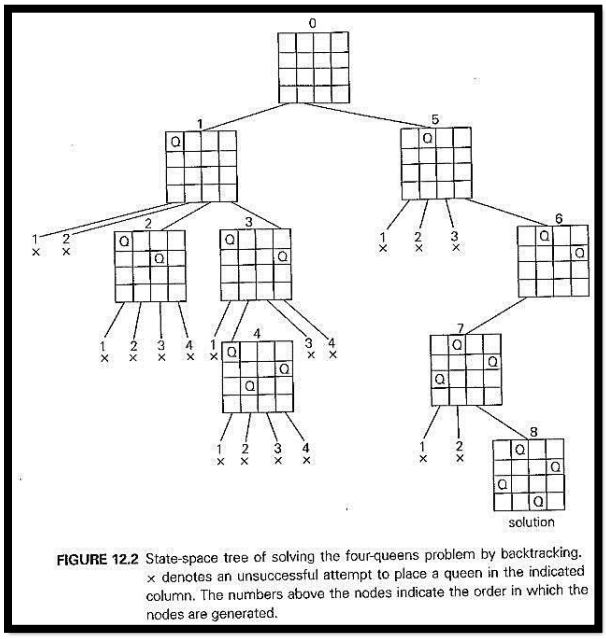
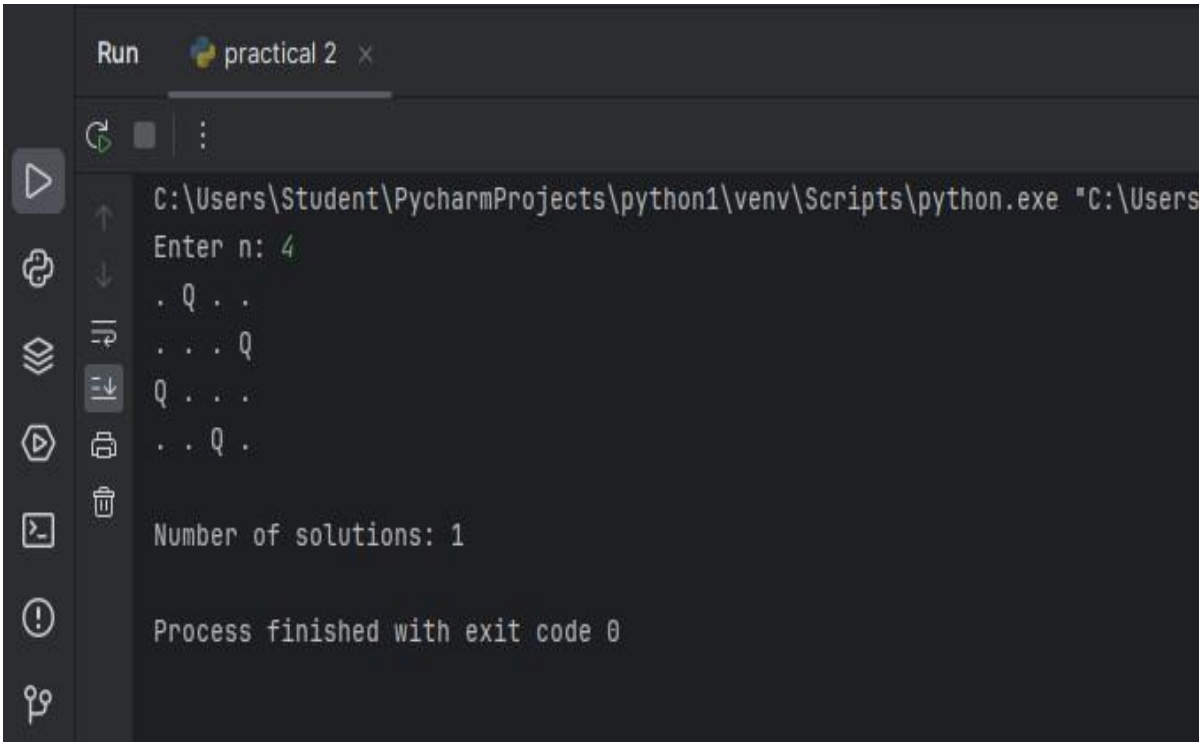


FIGURE 12.2 State-space tree of solving the four-queens problem by backtracking. × denotes an unsuccessful attempt to place a queen in the indicated column. The numbers above the nodes indicate the order in which the nodes are generated.

PYTHON CODE:-

```
class QueenChessBoard:
    def __init__(self, size):
       self.size = size
       self.columns = []
    def place_in_next_row(self, column):
        self.columns.append(column)
    def remove_in_current_row(self):
        return self.columns.pop()
    def is_this_column_safe_in_next_row(self, column):
        row = len(self.columns)
        for queen_column in self.columns:
            if column == queen_column:
               return False
        for queen_row, queen_column in enumerate(self.columns):
            if queen_column - queen_row == column - row:
               return False
        for queen_row, queen_column in enumerate(self.columns):
            if ((self.size - queen_column) - queen_row
               == (self.size - column) - row):
               return False
        return True
    def display(self):
        for row in range(self.size):
            for column in range(self.size):
                if column == self.columns[row]:
                   print('Q', end=' ')
                else:
                    print('.', end=' ')
            print()
def solve_queen(size):
    board = QueenChessBoard(size)
    number_of_solutions = 0
    row = 0
    column = 0
    while True:
        while column < size:
            if board.is_this_column_safe_in_next_row(column):
                board.place_in_next_row(column)
                row += 1
                column = 0
                break
            else:
                column += 1
        if (column == size or row == size):
            if row == size:
               board.display()
               print()
               number_of_solutions += 1
               row -= 1
            try:
               prev_column = board.remove_in_current_row()
            except IndexError:
                break
            row -= 1
            column = 1 + prev_column
    print('Number of solutions:', number_of_solutions)
n = int(input('Enter n: '))
solve_queen(n)
```

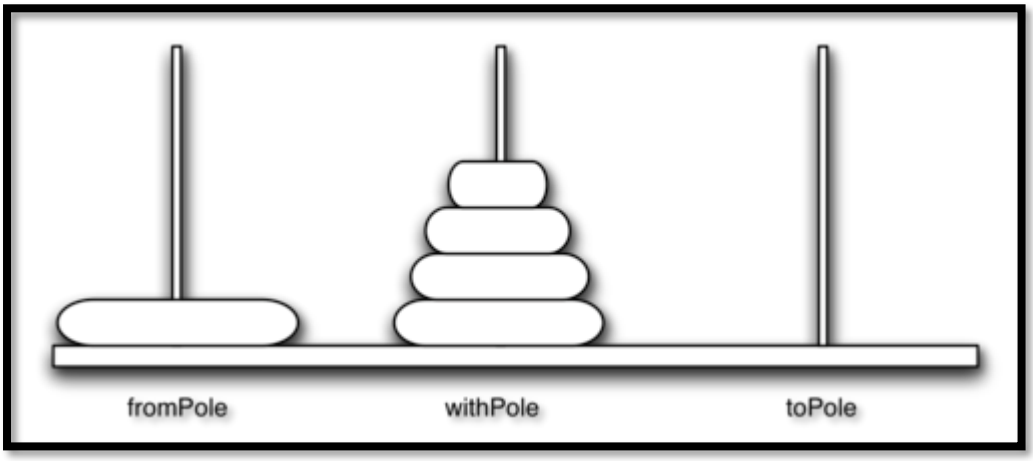
OUTPUT:


Run
practical 2
C:\Users\Student\PycharmProjects\python1\venv\Scripts\python.exe "C:\Users
Enter n: 4
. Q . .
. . . Q
Q . . .
. . Q .
Number of solutions: 1
Process finished with exit code 0

AIM:-

Write a program to solve tower of Hanoi problem.

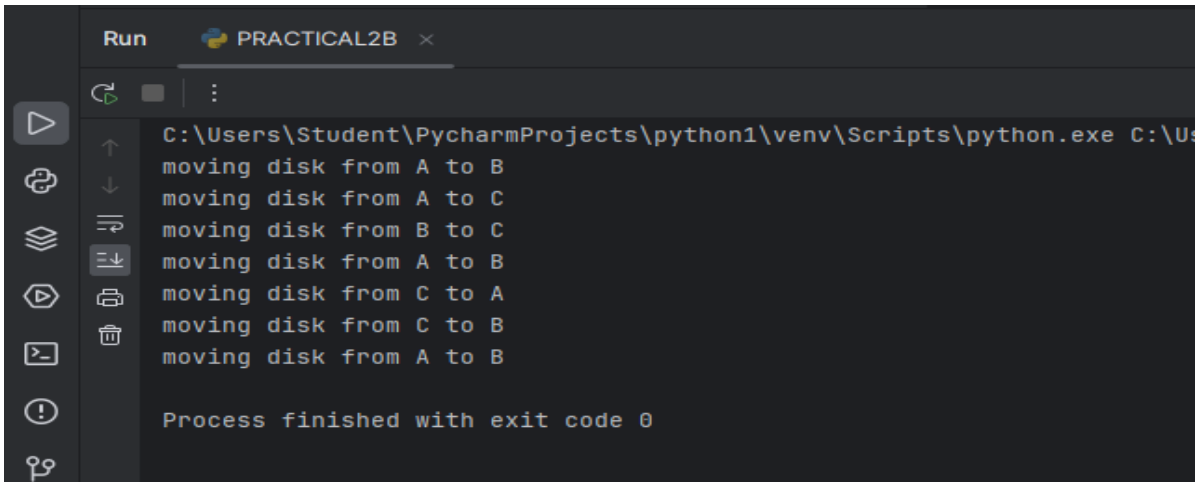
DIAGRAM



Python 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)
def moveDisk(fp,tp):
    print("moving disk from",fp,"to",tp)
moveTower(3,"A","B","C")
```

OUTPUT:



PRACTICAL NO.3

A. Write a program to implement alpha beta search.
B. Write a program for Hill climbing problem.

AIM:-

Write a program to implement alpha beta search.

PYTHON 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)
```

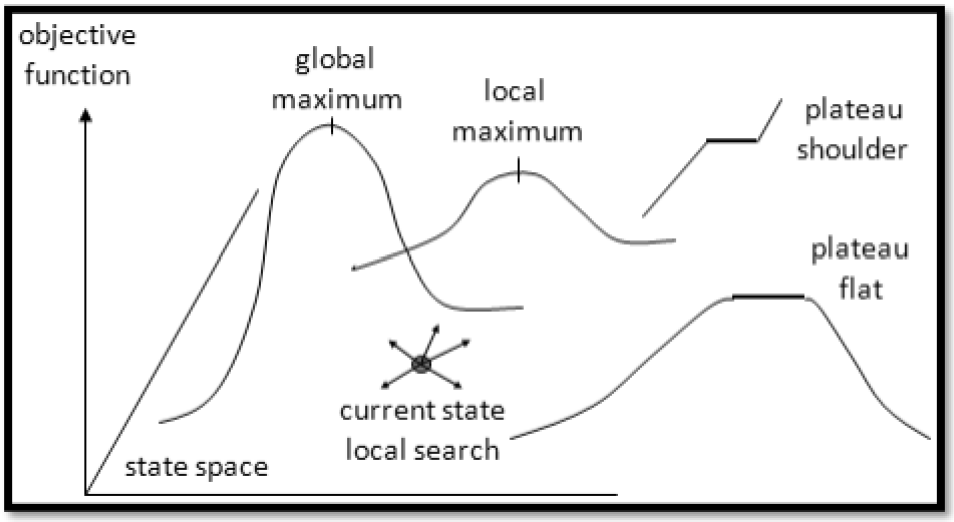
OUTPUT :


C:\Users\Student\PycharmProjects\pythonProject\venv\Scripts\python.exe C:\Users\Student\PycharmProjects\pythonProject\practical3.py
(alpha, beta): 5 15
Result: 5
Times pruned: 1
Process finished with exit code 0

AIM:-

Write a program for Hill climbing problem.

DIAGRAM:-

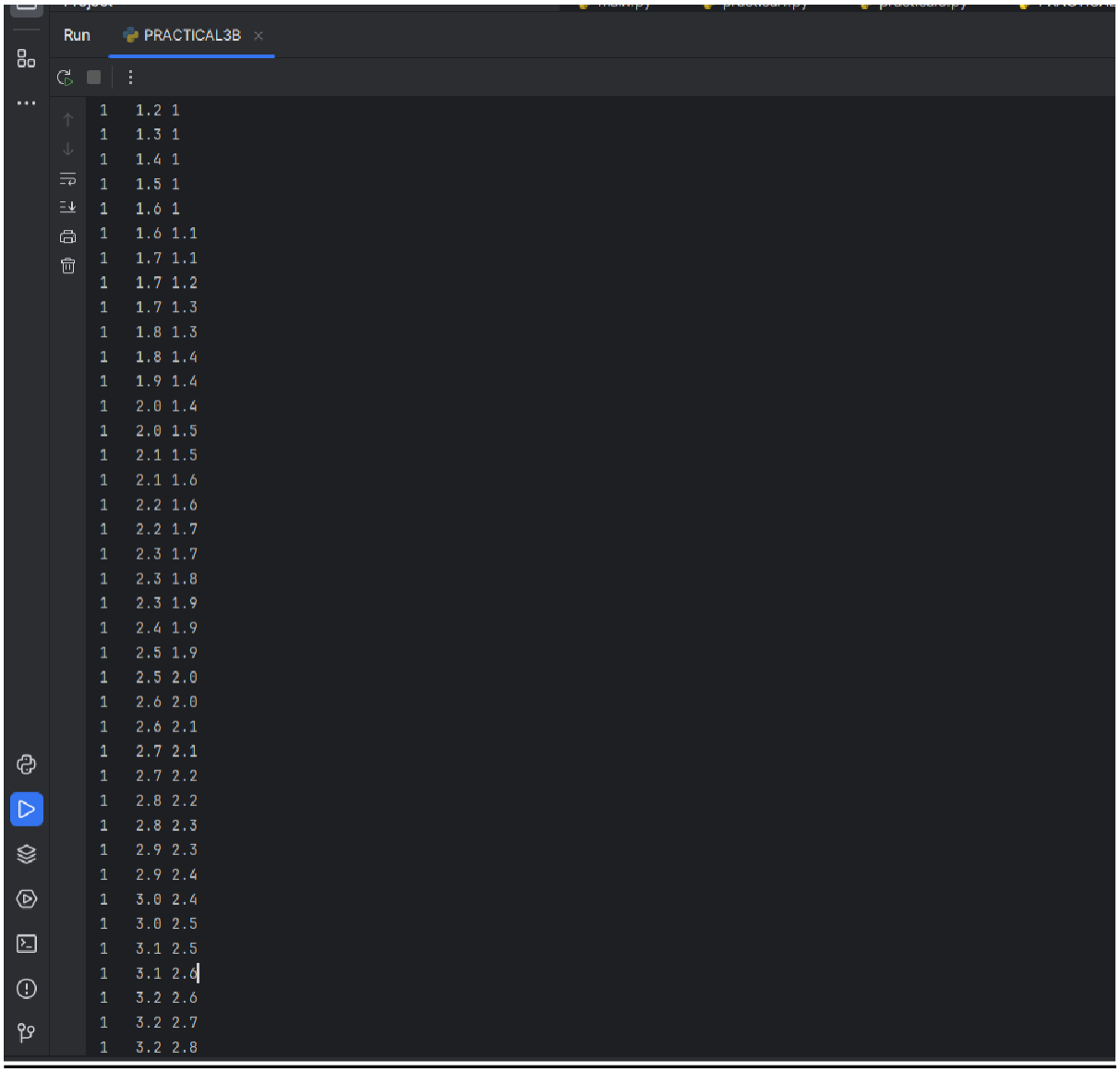


PYTHON CODE:

```
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
    else:
        flag = False
```

OUTPUT:



Practical no-4

A. Write a program to implement A* algorithm.

B. Write a program to implement AO* algorithm.

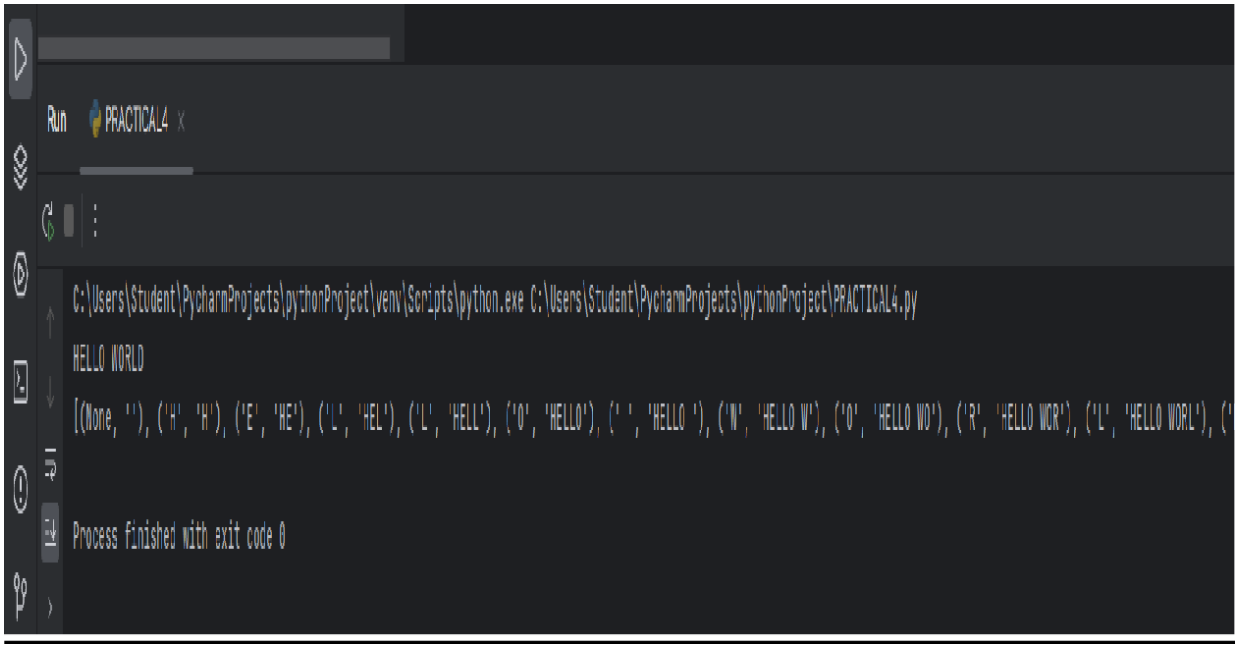
Aim:-

Write a program to implement A* algorithm.

PYTHON 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):
      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:



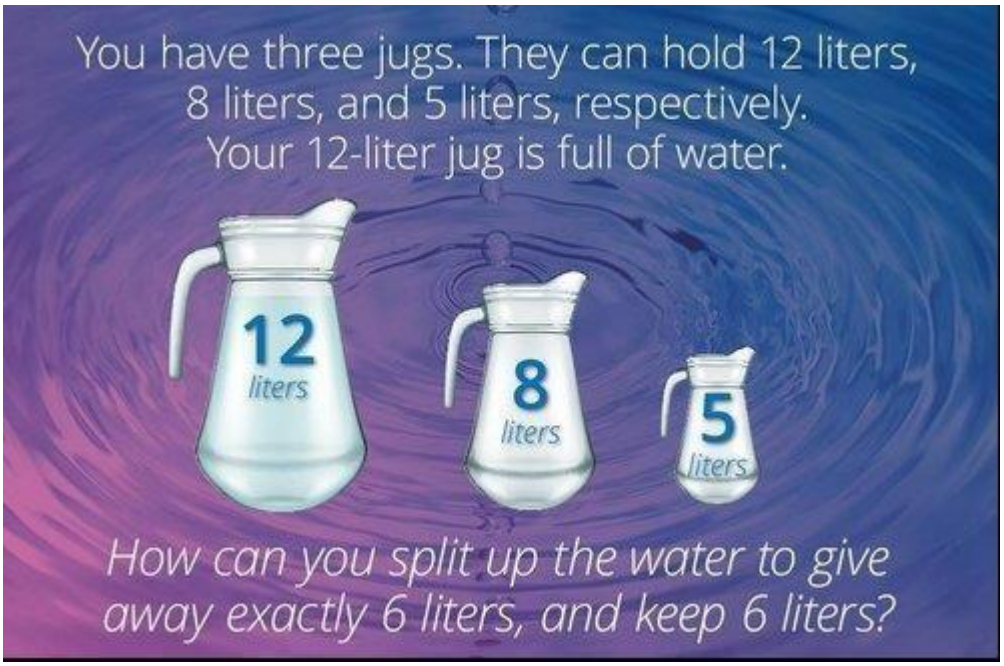
Practical no-5

A. Write a program to solve the water jug problem.

Aim: -

Write a program to solve the water jug problem.

Diagram: -



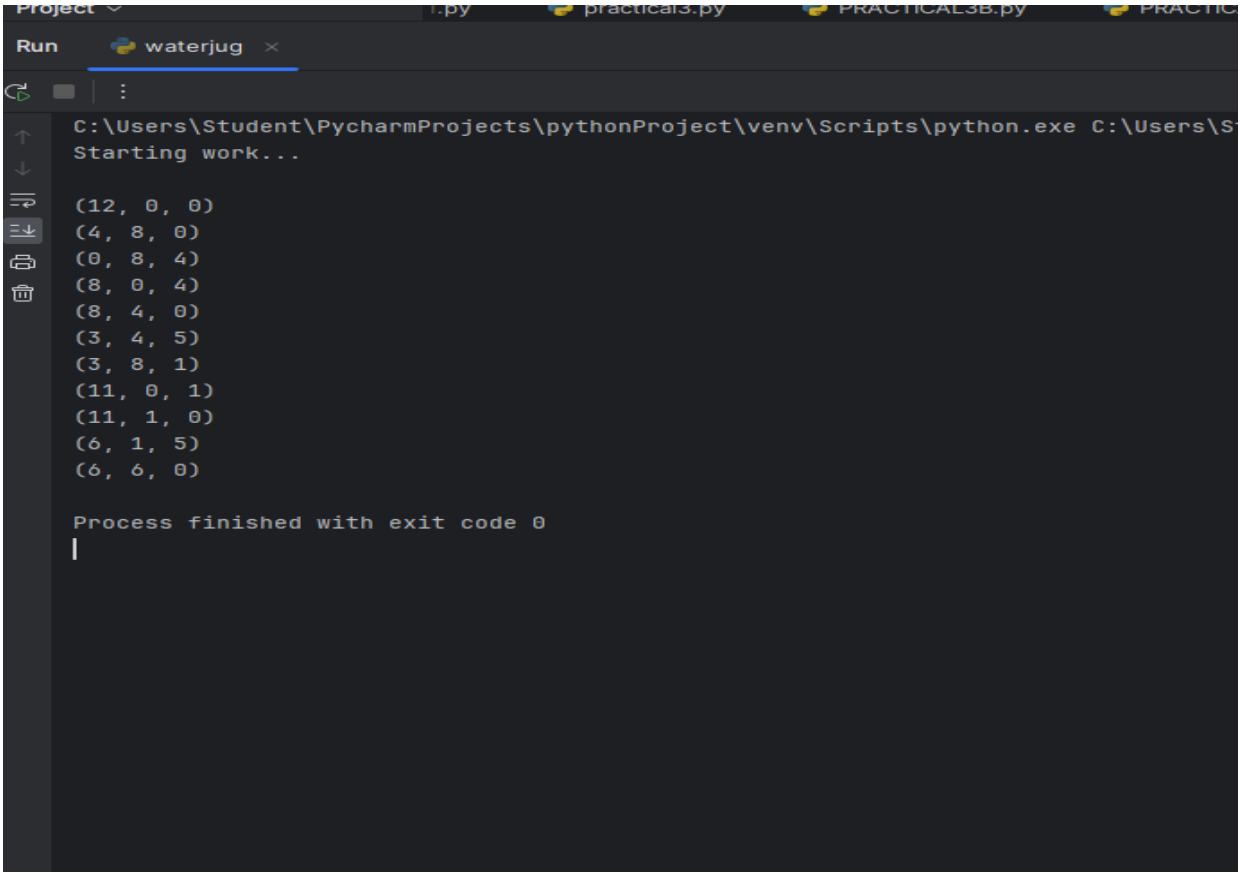
Python Code:

```
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
                # 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


    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

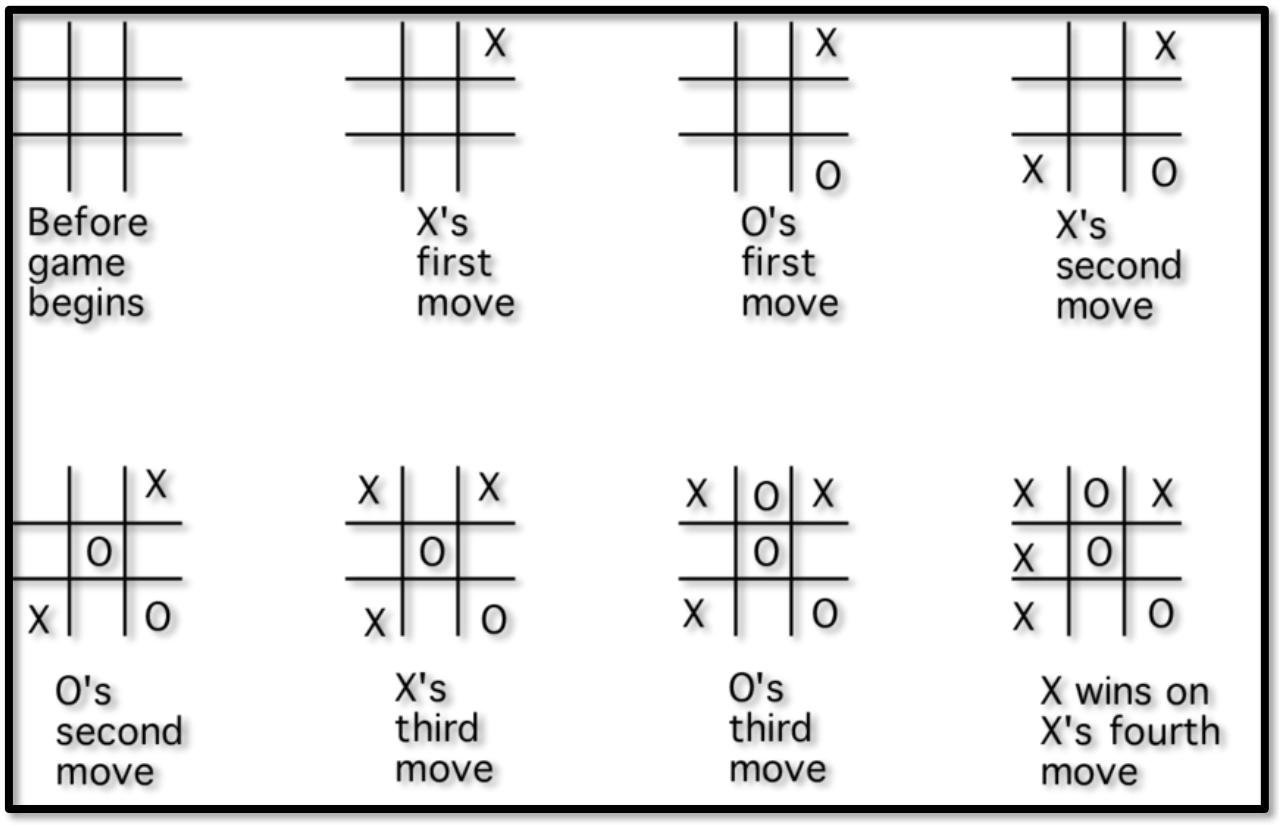


B. Design the simulation of tic – tac – toe game using min-max algorithm .

Aim:-

Design the simulation of TIC – TAC –TOE game using min-max algorithm

Diagram:-



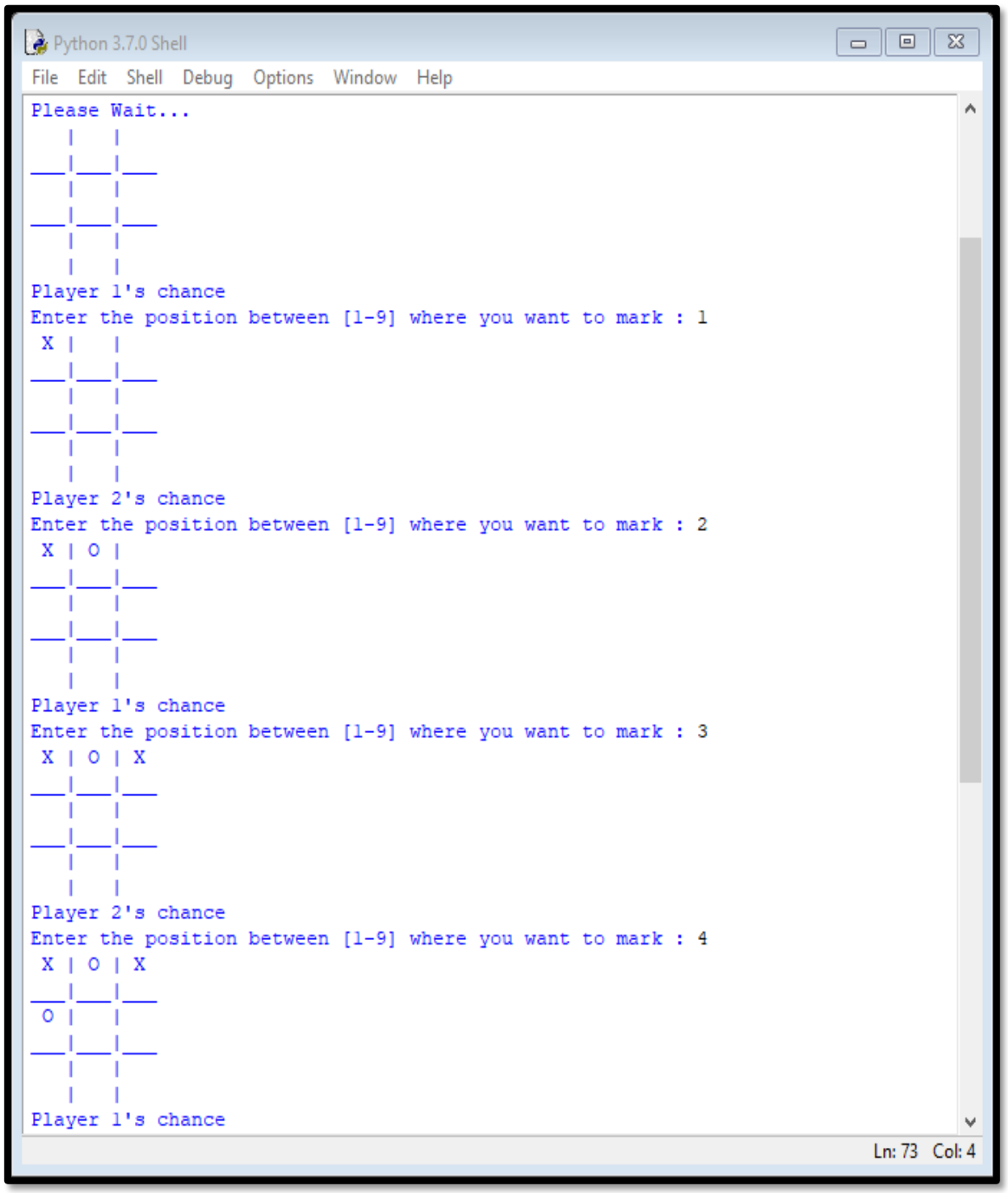
Python Code:

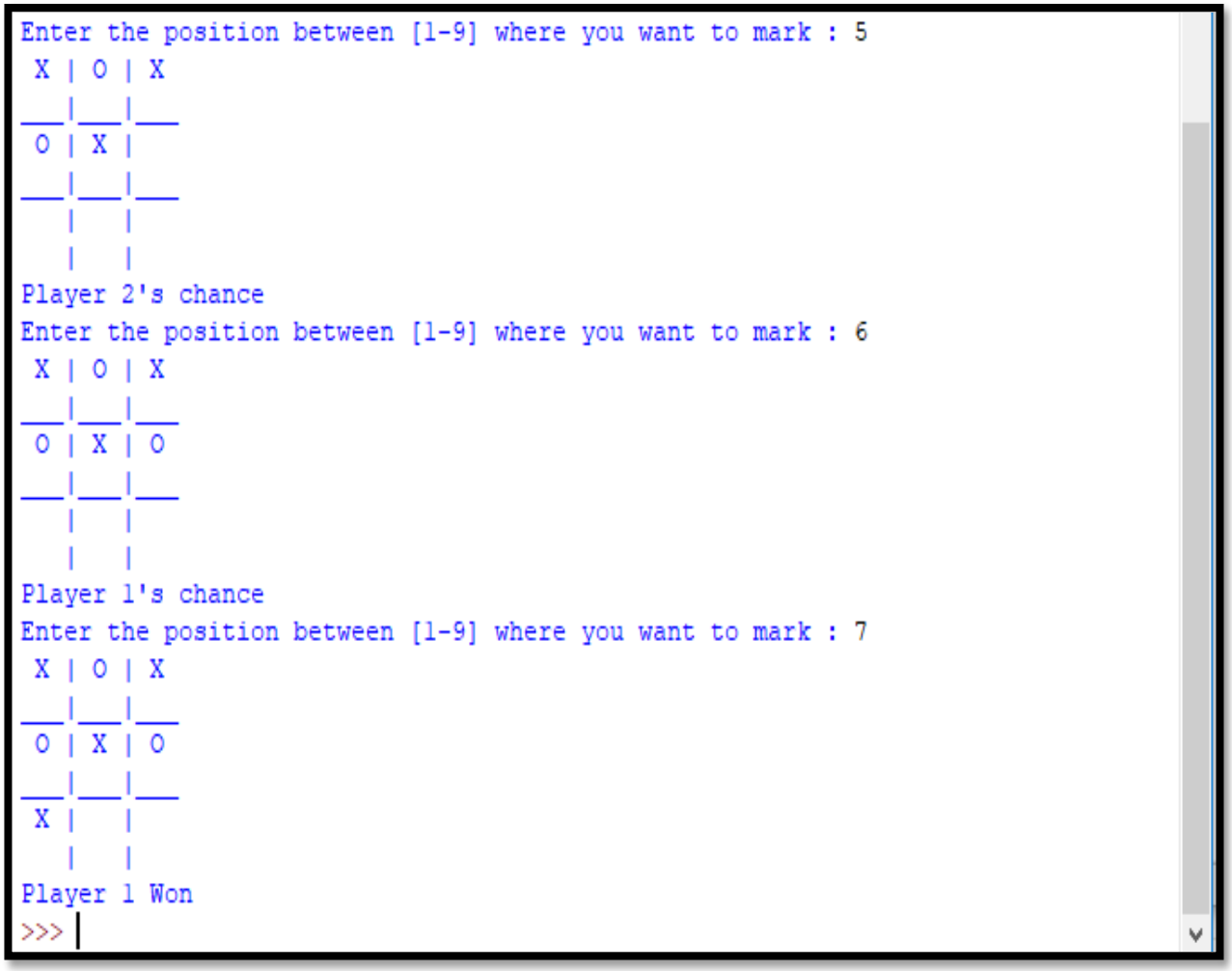
```
import os
import time

board = [' ',' ',' ',' ',' ',' ',' ',' ',' ',' ']
player = 1
Win = 1
Draw = -1
Running = 0
Stop = 1
Game = Running
Mark = 'X'
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("   |   |   ")
def CheckPosition(x):
   if(board[x] == ' '):
        return True
    else:
        return False
def CheckWin():
    global Game
    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
    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
    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() 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:-




Enter the position between [1-9] where you want to mark : 5
X | O | X
___|___|___
O | X |
___|___|___
| |
| |
Player 2's chance
Enter the position between [1-9] where you want to mark : 6
X | O | X
___|___|___
O | X | O
___|___|___
| |
| |
Player 1's chance
Enter the position between [1-9] where you want to mark : 7
X | O | X
___|___|___
O | X | O
___|___|___
X | |
| |
Player 1 Won
>>>

PRACTICAL No.-6

AIM:-

Design an application to simulate number puzzle problem.

PYHTON 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’]]

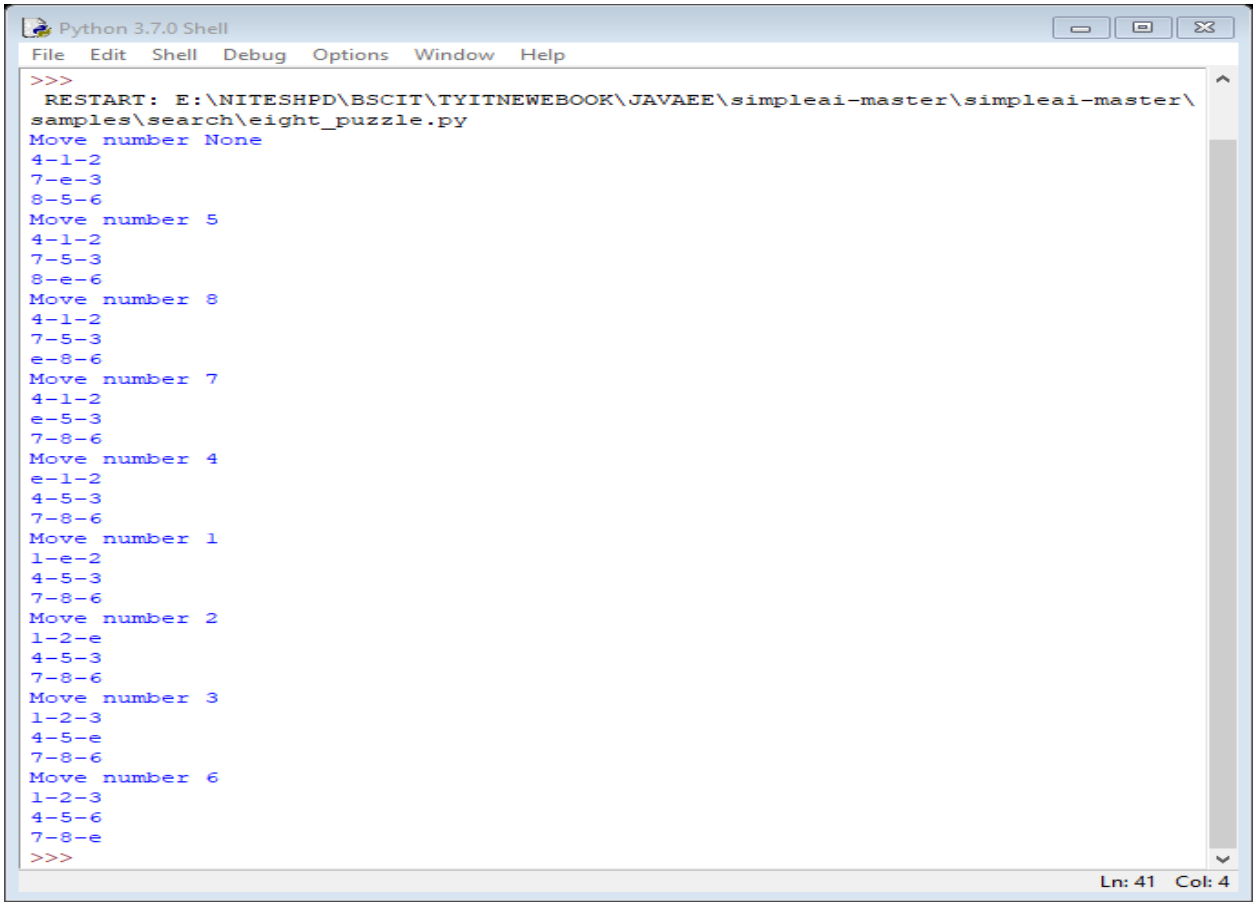
'''

```
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_.spit('\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:-


Python 3.7.0 Shell
File Edit Shell Debug Options Window Help
>>>
RESTART: E:\NITESHPD\BSCIT\TYITNEWEBOOK\JAVAEE\simpleai-master\simpleai-master\samples\search\eight_puzzle.py
Move number None
4-1-2
7-e-3
8-5-6
Move number 5
4-1-2
7-5-3
8-e-6
Move number 8
4-1-2
7-5-3
e-8-6
Move number 7
4-1-2
e-5-3
7-8-6
Move number 4
e-1-2
4-5-3
7-8-6
Move number 1
1-e-2
4-5-3
7-8-6
Move number 2
1-2-e
4-5-3
7-8-6
Move number 3
1-2-3
4-5-e
7-8-6
Move number 6
1-2-3
4-5-6
7-8-e
>>>
Ln: 41 Col: 4

' PRACTICAL No.-7

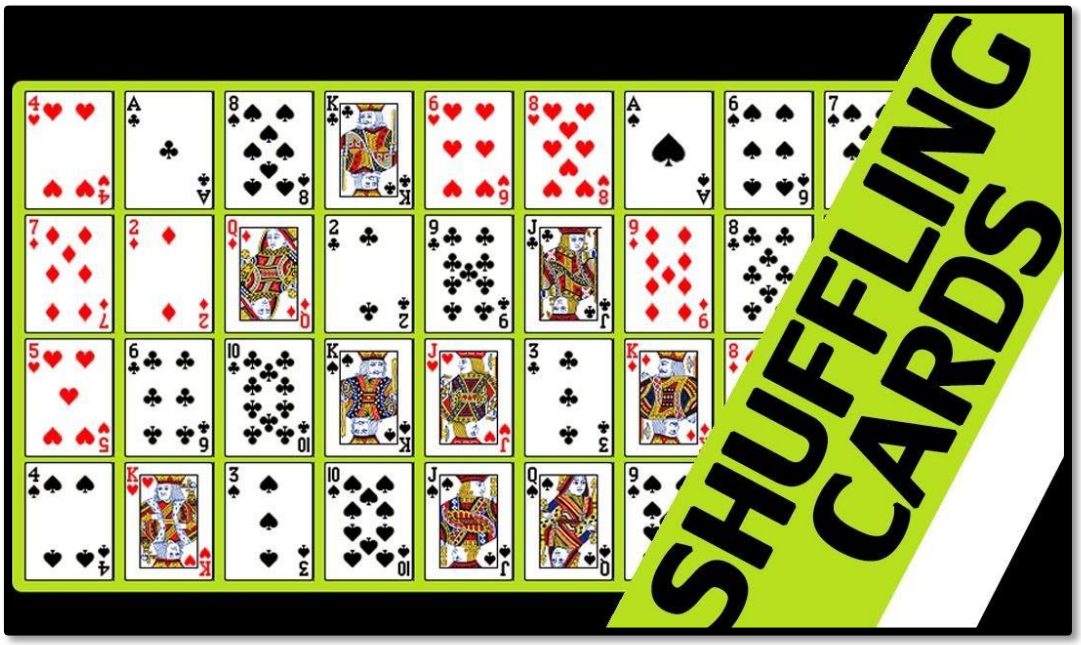
A. Write a program to shuffle Deck of cards.

B. Solve traveling salesman problem using artificial intelligence technique.

Aim:-

Write a program to shuffle Deck of cards.

Diagram:-



Python Code:-

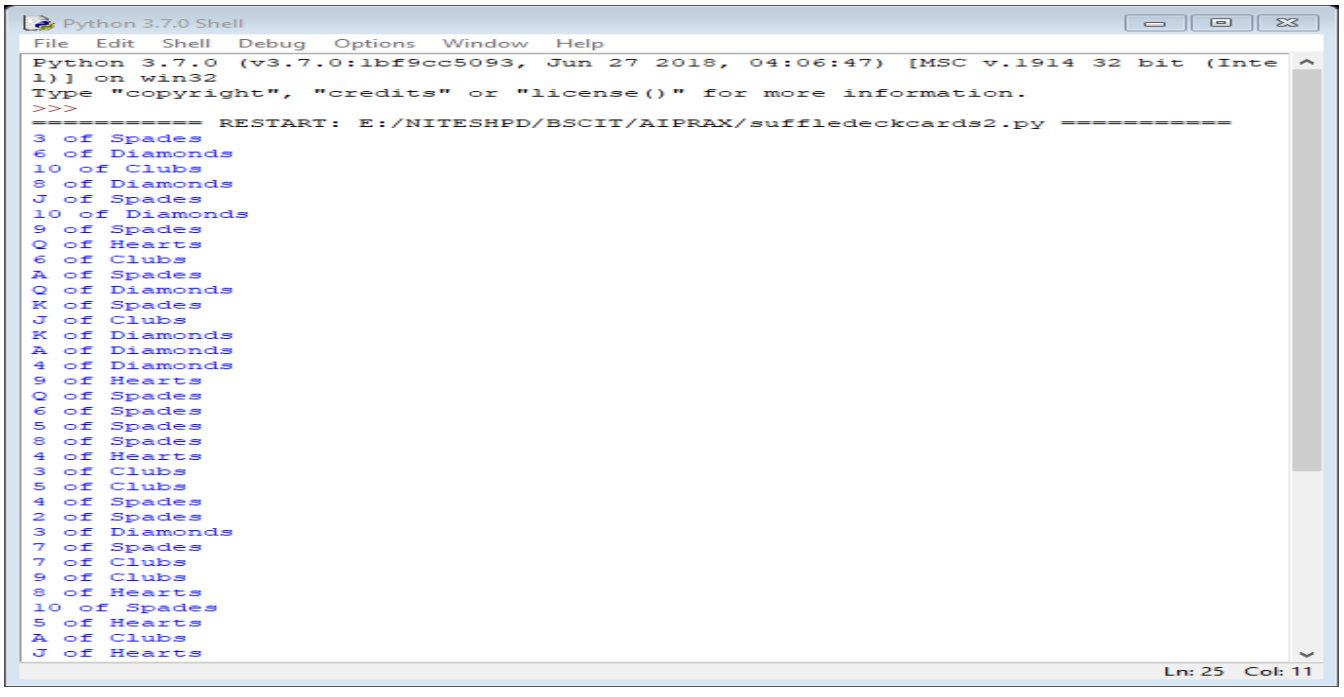
```
#first let's import random procedures since we will be shuffling
 import random

#next, let's start building list holders so we can place our cards in there:
 cardfaces = []
 suits = ["Hearts", "Diamonds", "Clubs", "Spades"]
 royals = ["J", "Q", "K", "A"]
 deck = []

#now, let's start using loops to add our content:
 for i in range(2,11):
    cardfaces.append(str(i)) #this adds numbers 2-10 and converts them to string
 data

 for j in range(4):
```

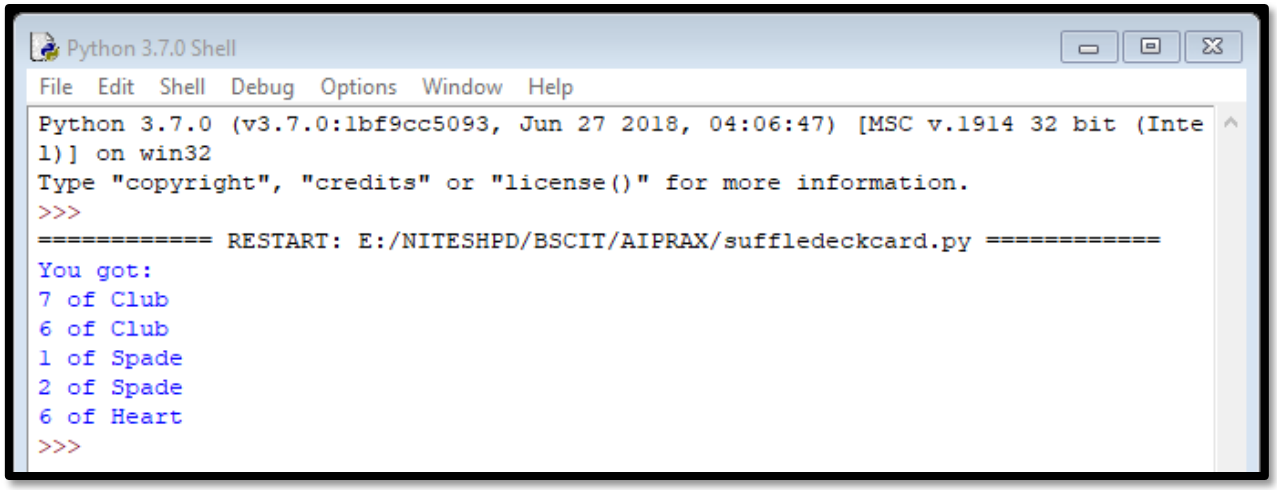
```
    cardfaces.append(royals[j]) #this will add the royal faces to the cardbase

 for k in range(4):  for l in
    range(13):
       card = (cardfaces[l] + " of " + suits[k])
       #this makes each card, cycling through suits, but first through faces
       deck.append(card)
      #this adds the information to the "full deck" we want to make #now let's
shuffle our deck!
 random.shuffle(deck)

#now let's see the cards!
 for m in range(52): print(deck[m])
```

OR

```
# 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:-




Python 3.7.0 Shell
File Edit Shell Debug Options Window Help
Python 3.7.0 (v3.7.0:1bf9cc5093, Jun 27 2018, 04:06:47) [MSC v.1914 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>>
========== RESTART: E:/NITESHPD/BSCIT/AIPRAX/suffledeckcard.py ==========
You got:
7 of Club
6 of Club
1 of Spade
2 of Spade
6 of Heart
>>>

PRACTICAL No.-8

Aim:-

Implementation Of Constraints Satisfactions Problem

PYTHON 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:-