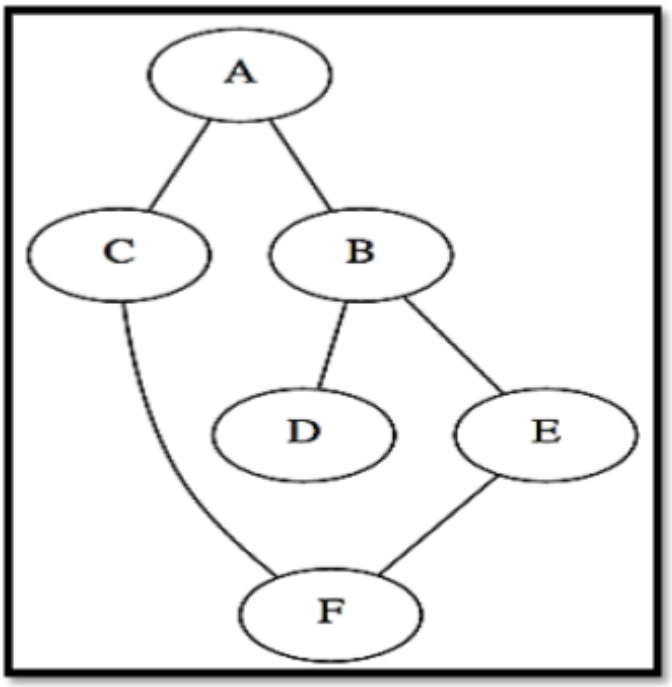
PRACTICAL NO-1

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

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

GRAPH:-

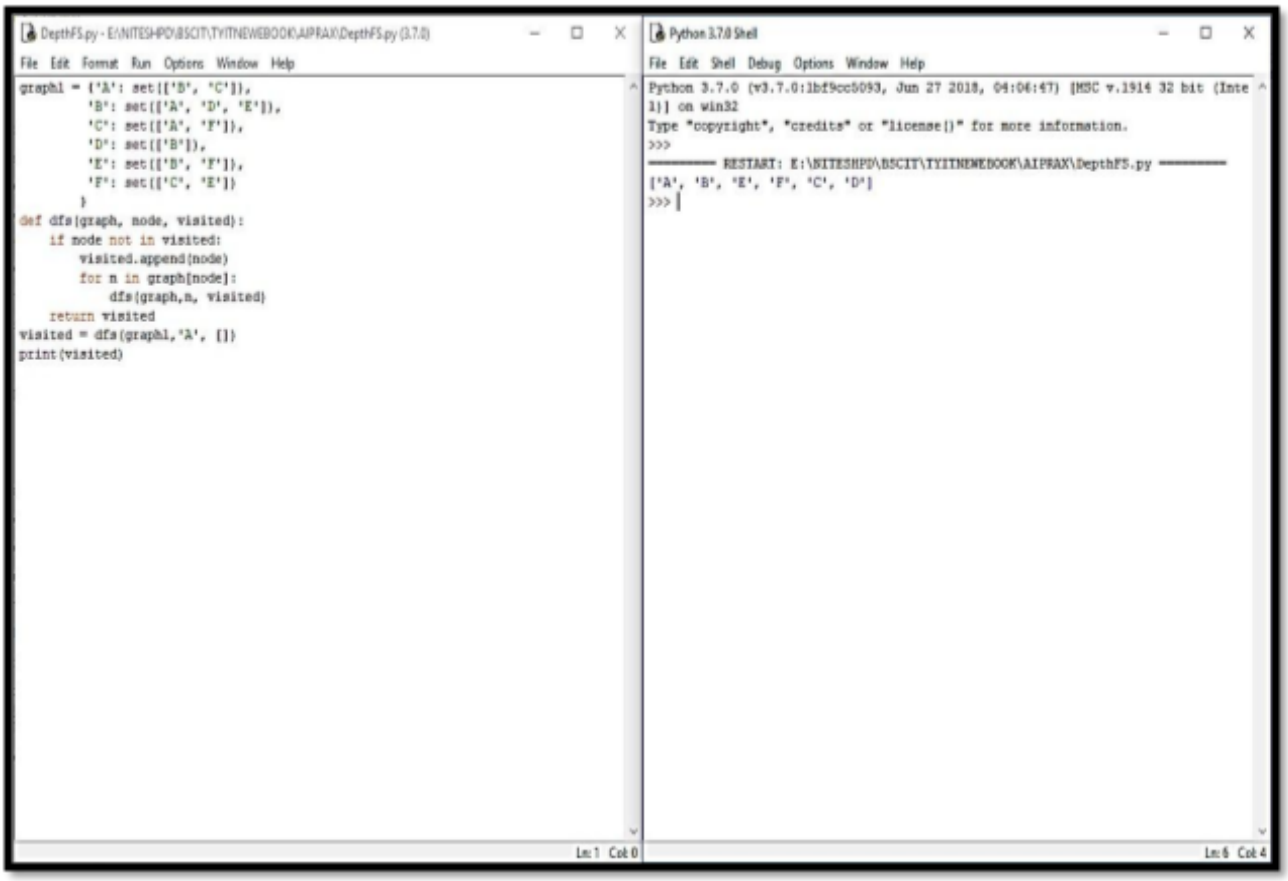


PYTHON CODE:-

```
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)
```

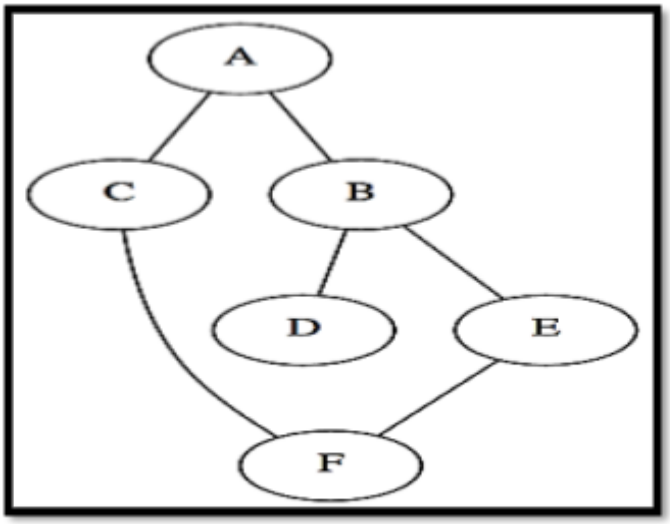
OUTPUT:-



AIM:-

Write a program to implement breadth first search algorithm.

GRAPH:-



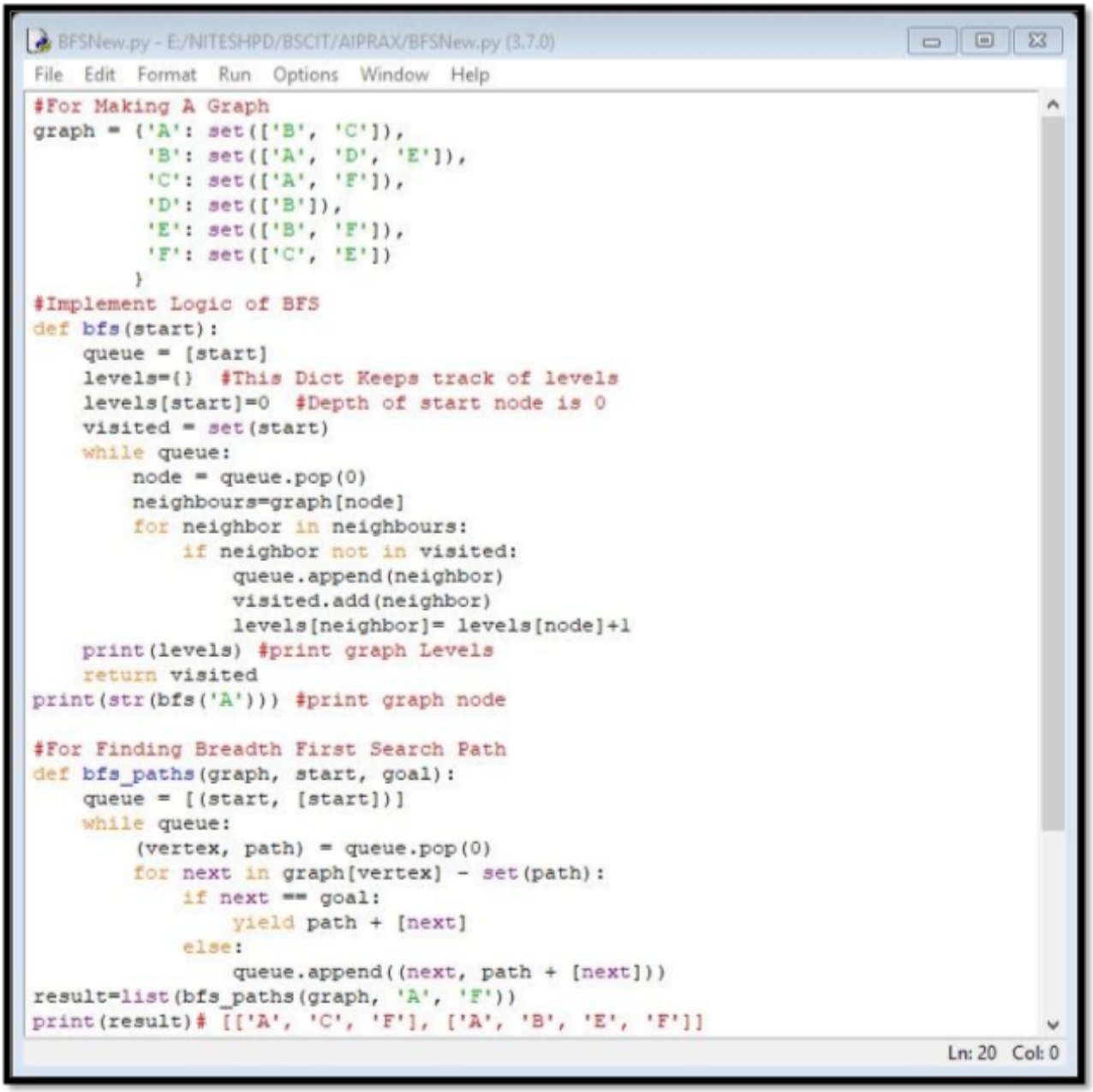
PYTHON CODE:-

```
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 = {}  # Keeps track of levels
    levels[start] = 0  # Depth of start node is 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)
    return visited

def bfs_paths(graph, start, goal):
    queue = [(start, [start])]

    while queue:
        (vertex, path) = queue.pop(0)

        for next_node in graph[vertex] - set(path):
            if next_node == goal:
                yield path + [next_node]
            else:
                queue.append((next_node, path +
[next_node]))
```

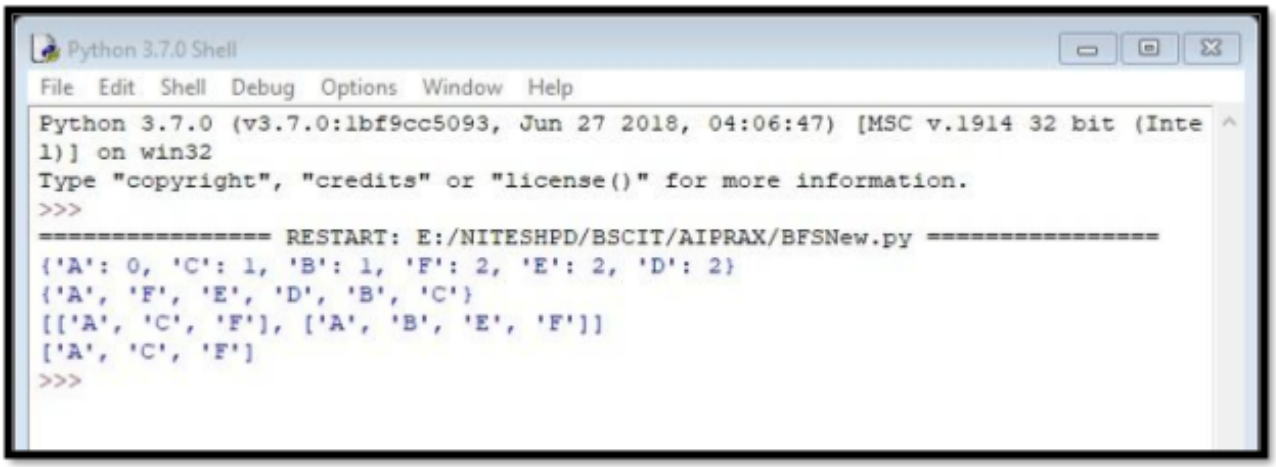
```
result = list(bfs_paths(graph, 'A', 'F'))
print(result)  # [['A', 'C', 'F'], ['A', 'B', 'E', 'F']]

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:-


BFSNew.py - E:/NITESHPD/BSCIT/AIPRAX/BFSNew.py (3.7.0)
File Edit Format Run Options Window Help
#For Making A Graph
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'])
}
#Implement Logic of BFS
def bfs(start):
queue = [start]
levels={} #This Dict Keeps track of levels
levels[start]=0 #Depth of start node is 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 Levels
return visited
print(str(bfs('A'))) #print graph node
#For Finding Breadth First Search Path
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)# [['A', 'C', 'F'], ['A', 'B', 'E', 'F']]
Ln: 20 Col: 0

```python
#For finding shortest path
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']
```

Ln: 20 Col: 0



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:

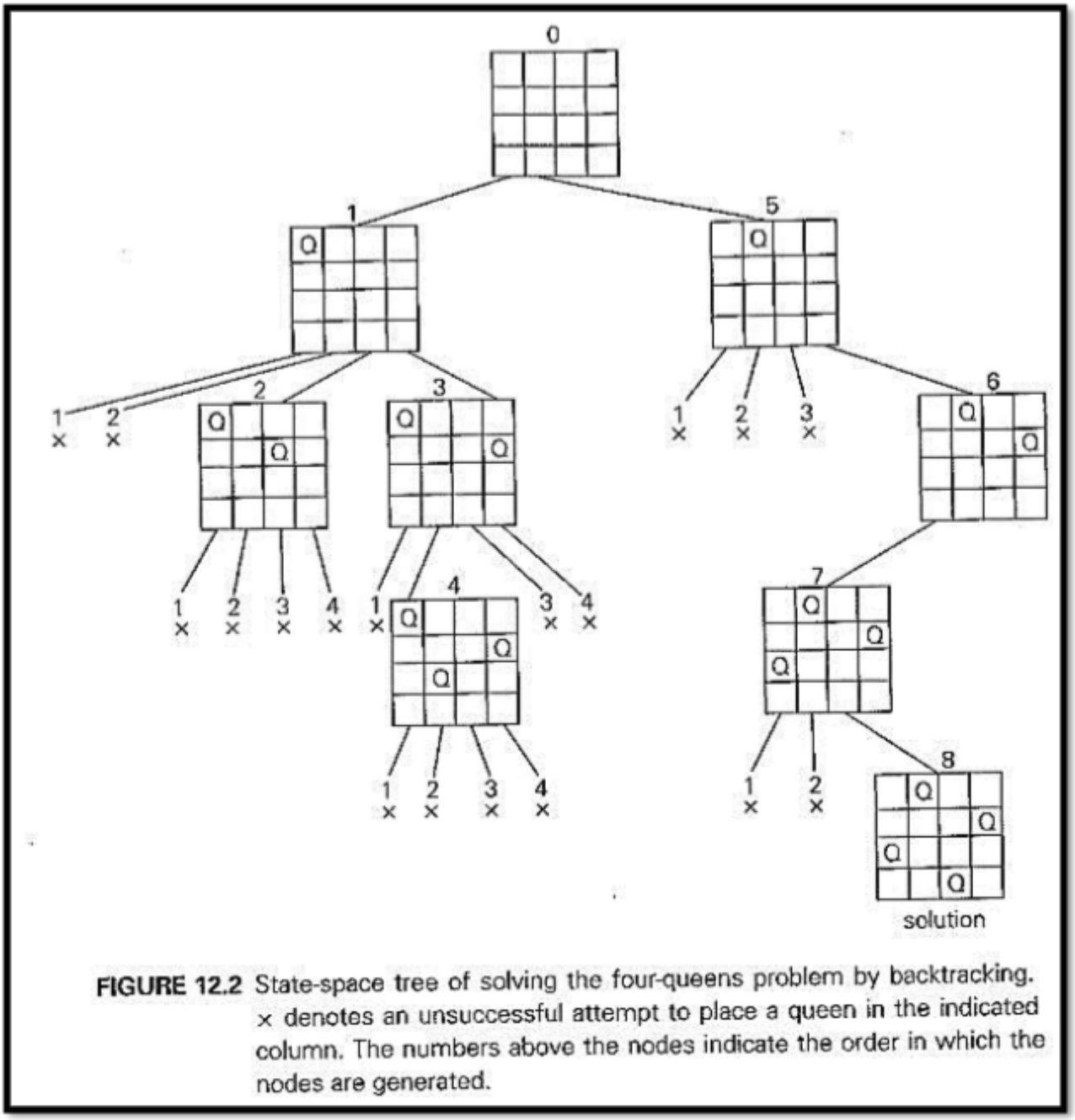


FIGURE 12.2 State-space tree of solving the four-queens problem by backtracking. × denotes an unsuccessful attempt to place a queen in the indicated column. The numbers above the nodes indicate the order in which the nodes are generated.

PYTHON CODE:-

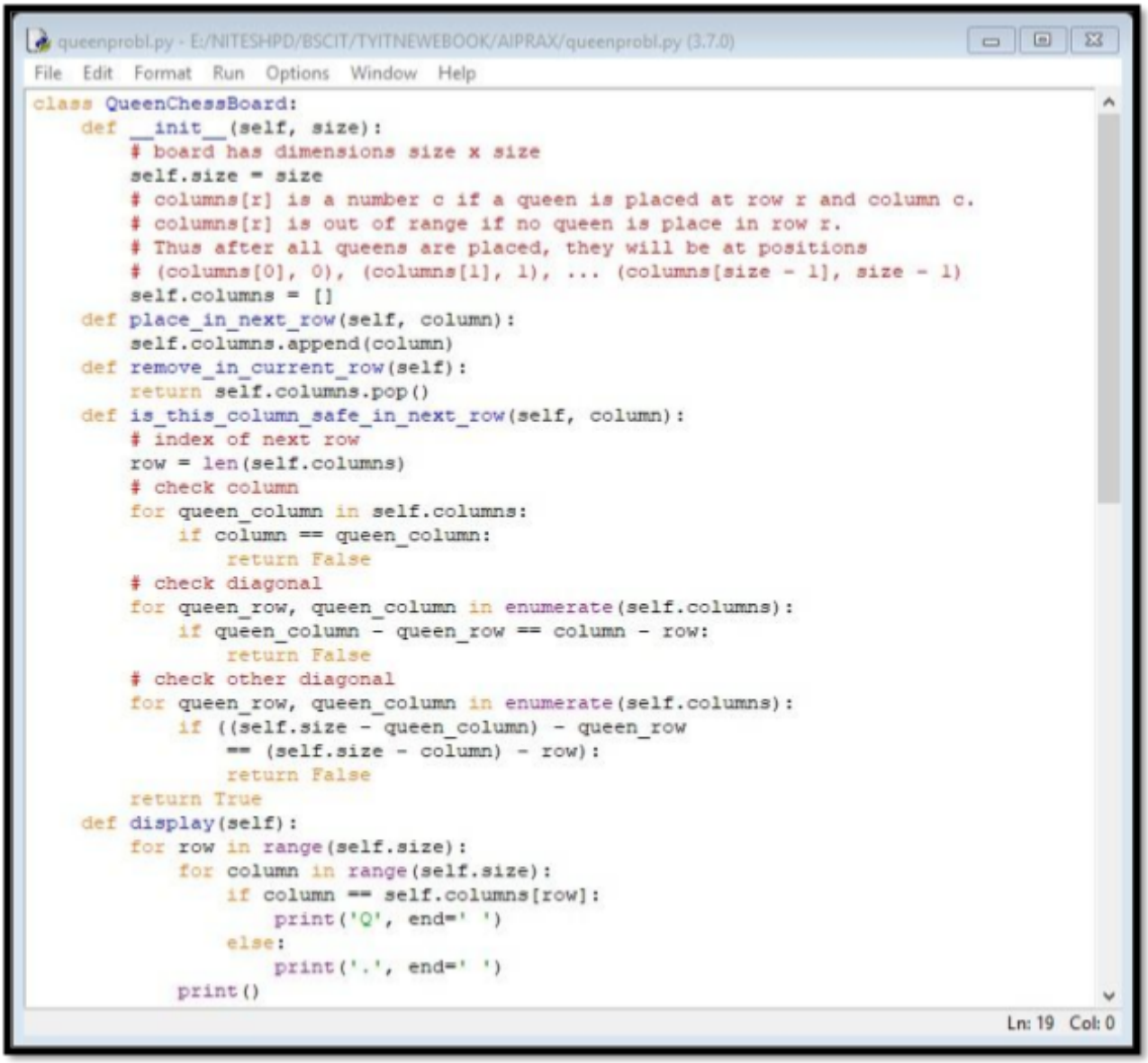
```
class QueenChessBoard:
    def __init__(self, size):
```

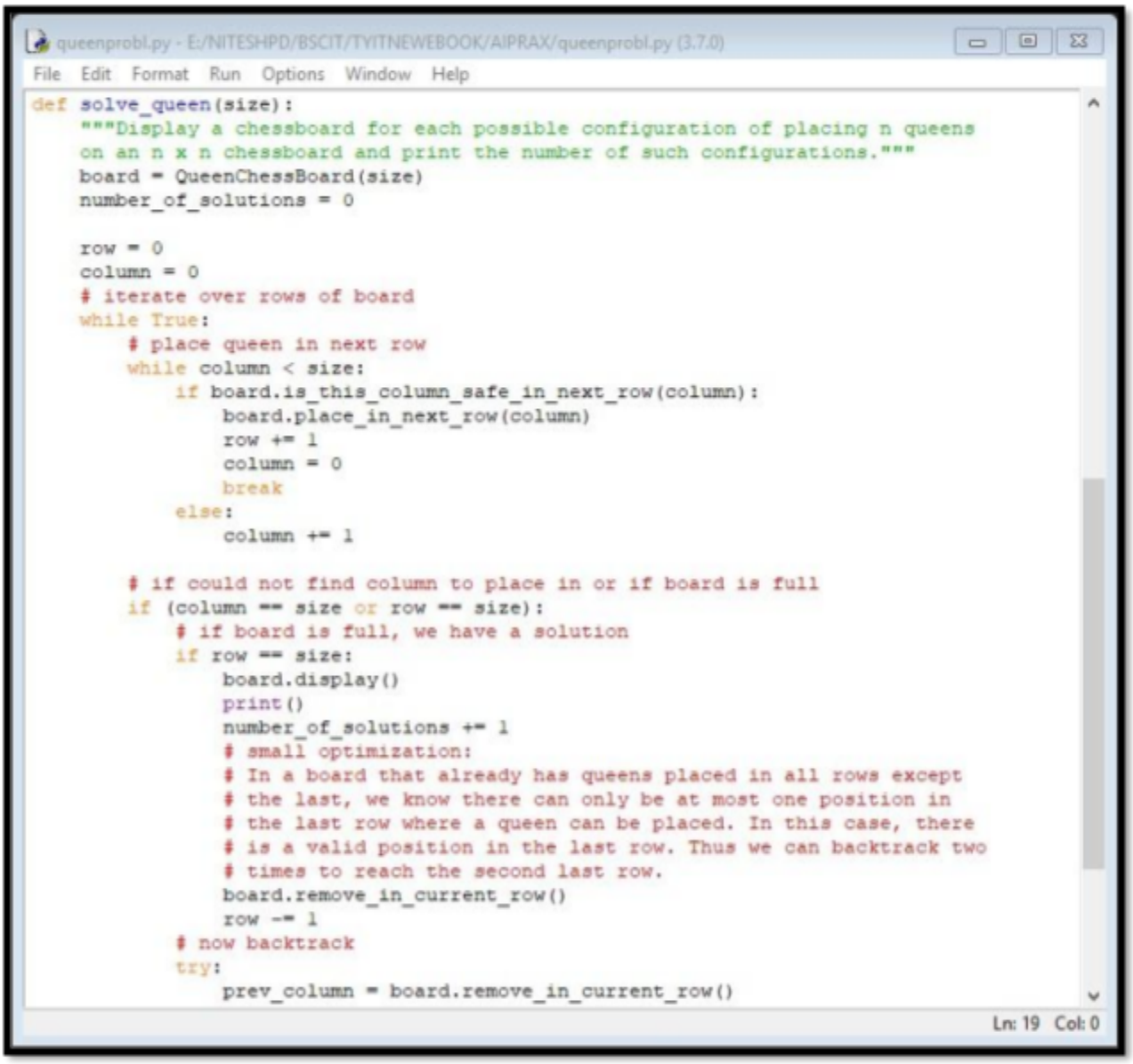
```
        self.size = size
        self.columns = []

    def place_in_next_row(self, column):
        self.columns.append(column)

    def remove_in_current_row(self):
        if not self.columns:
            return None  # Return None or handle the case
appropriately
        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 or
\
                    (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=' ')
```

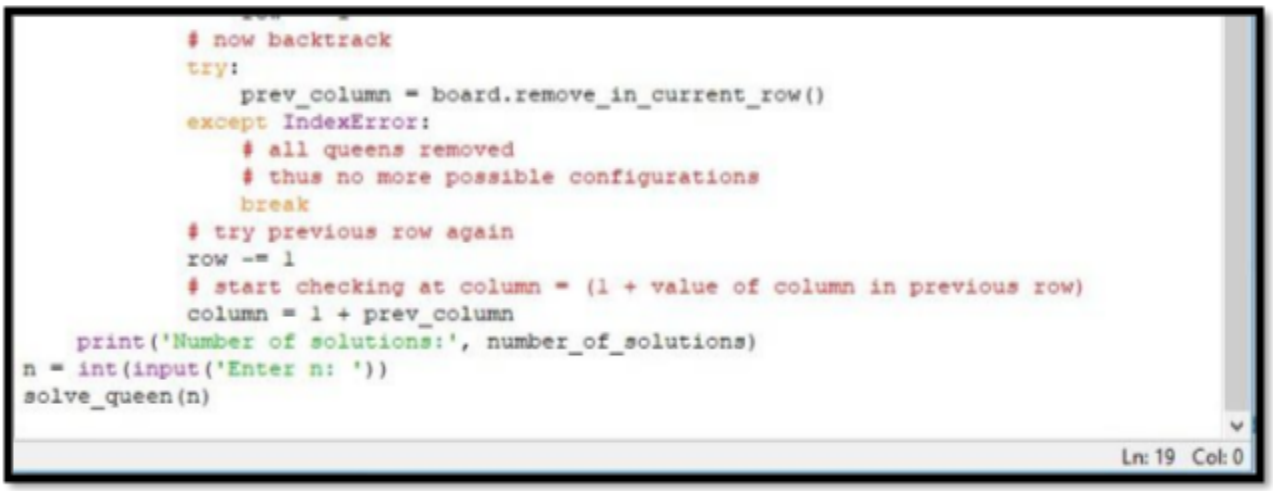
```
            print()

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

            removed_column = board.remove_in_current_row()
            if removed_column is None:
                break

            row -= 1
            column = 1 + removed_column

    print('Number of solutions:', number_of_solutions)
```

```
n = int(input('Enter n: '))
solve_queen(n)
```

OUTPUT:-


queenprobl.py - E:/NITESHPD/BSCIT/TYITNEWEBOOK/AIPRAX/queenprobl.py (3.7.0)
File Edit Format Run Options Window Help
class QueenChessBoard:
def __init__(self, size):
# board has dimensions size x size
self.size = size
# columns[r] is a number c if a queen is placed at row r and column c.
# columns[r] is out of range if no queen is place in row r.
# Thus after all queens are placed, they will be at positions
# (columns[0], 0), (columns[1], 1), ... (columns[size - 1], size - 1)
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):
# index of next row
row = len(self.columns)
# check column
for queen_column in self.columns:
if column == queen_column:
return False
# check diagonal
for queen_row, queen_column in enumerate(self.columns):
if queen_column - queen_row == column - row:
return False
# check other diagonal
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()
Ln: 19 Col: 0

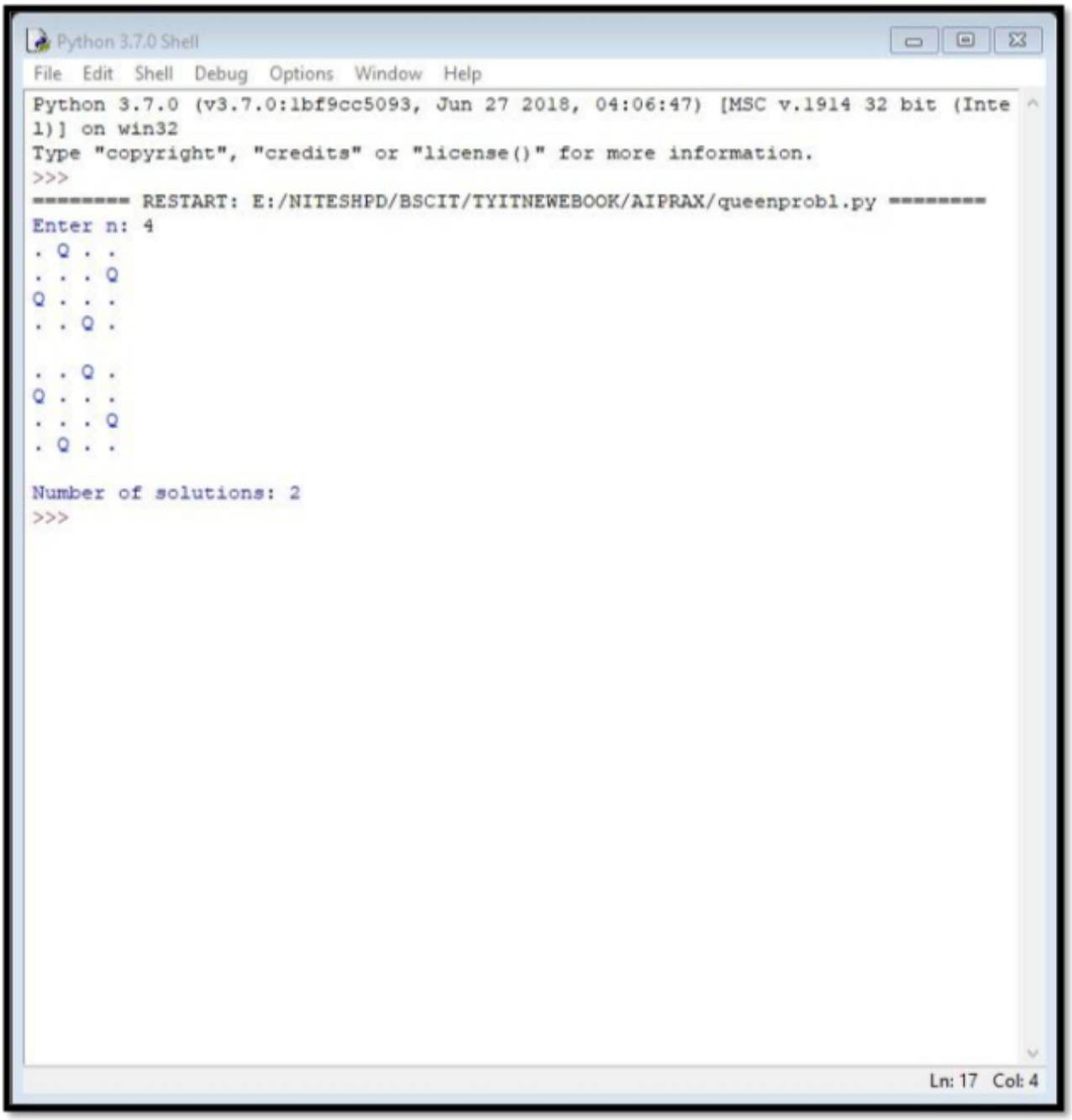

queenprobl.py - E:/NITESHPD/BSCIT/TYITNEWEBOOK/AIPRAX/queenprobl.py (3.7.0)
File Edit Format Run Options Window Help
def solve_queen(size):
"""Display a chessboard for each possible configuration of placing n queens
on an n x n chessboard and print the number of such configurations."""
board = QueenChessBoard(size)
number_of_solutions = 0
row = 0
column = 0
# iterate over rows of board
while True:
# place queen in next row
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 could not find column to place in or if board is full
if (column == size or row == size):
# if board is full, we have a solution
if row == size:
board.display()
print()
number_of_solutions += 1
# small optimization:
# In a board that already has queens placed in all rows except
# the last, we know there can only be at most one position in
# the last row where a queen can be placed. In this case, there
# is a valid position in the last row. Thus we can backtrack two
# times to reach the second last row.
board.remove_in_current_row()
row -= 1
# now backtrack
try:
prev_column = board.remove_in_current_row()
Ln: 19 Col: 0


# now backtrack
try:
prev_column = board.remove_in_current_row()
except IndexError:
all queens removed
thus no more possible configurations
break
try previous row again
row -= 1
# start checking at column = (1 + value of column in previous row)
column = 1 + prev_column
print('Number of solutions:', number_of_solutions)
n = int(input('Enter n: '))
solve_queen(n)
Ln: 19 Col: 0

NOTE:

1. The user is prompted to enter n where n is the number of queens to place and the size of the board.

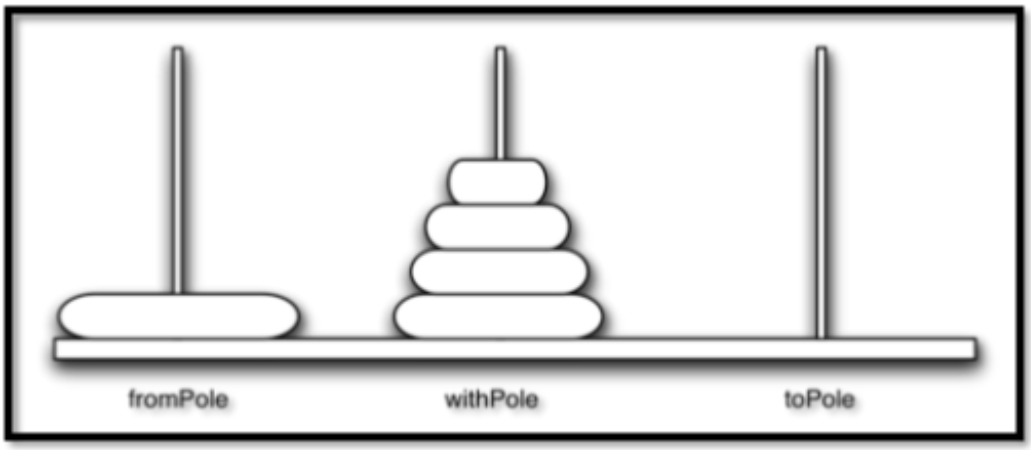
2. Solve queens is called on n to display all possible board configurations and the number of solutions.



AIM:-

Write a program to solve tower of Hanoi problem.

DIAGRAM:



PYTHON CODE:

```
def moveTower(height, fromPole, toPole, withPole):
    if height >= 1:
        # Move tower of height-1 from 'fromPole' to
'withPole' using 'toPole' as auxiliary
        moveTower(height-1, fromPole, withPole, toPole)

        # Move the remaining disk from 'fromPole' to
'toPole'
        moveDisk(fromPole, toPole)

        # Move the tower of height-1 from 'withPole' to
'toPole' using 'fromPole' as auxiliary
        moveTower(height-1, withPole, toPole, fromPole)

def moveDisk(fp, tp):
    # Function to simulate moving a disk from one pole to
another
    print("Moving disk from", fp, "to", tp)
```

```
# Initial call to moveTower with a tower of height 3, starting from pole 'A' to pole 'B', using pole 'C' as auxiliary
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:-

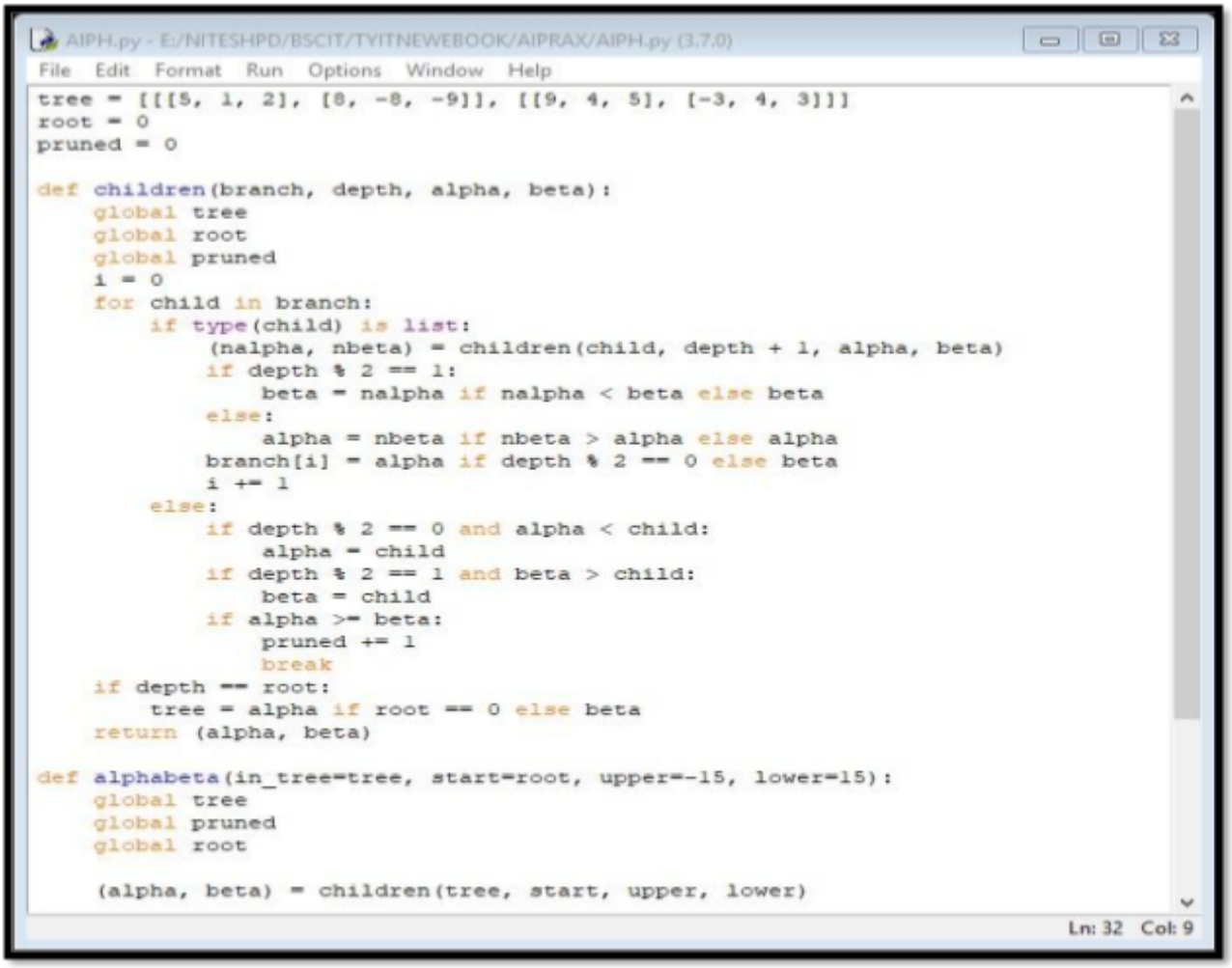
Write a program to implement alpha beta search.

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
```

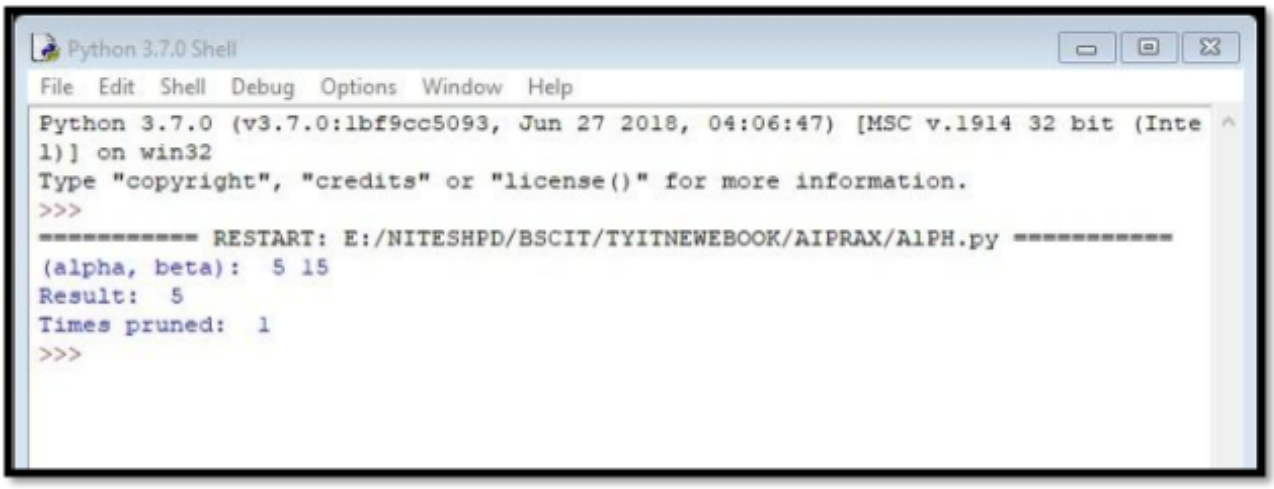
```
                break
    if depth == root:
        tree = alpha if root == 0 else beta
    return (alpha, beta)

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


AIPH.py - E:/NITESHPD/BSCIT/TYITNEWEBOOK/AIPRAX/AIPH.py (3.7.0)
File Edit Format Run Options Window Help
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)
Ln: 32 Col: 9

```
    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)
```

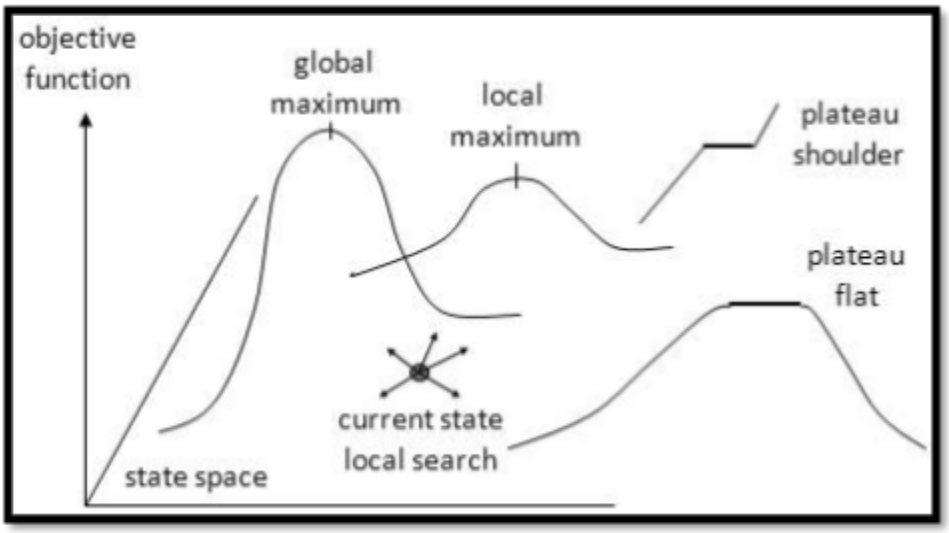
Ln: 32 Col: 9



AIM:-

Write a program for Hill climbing problem.

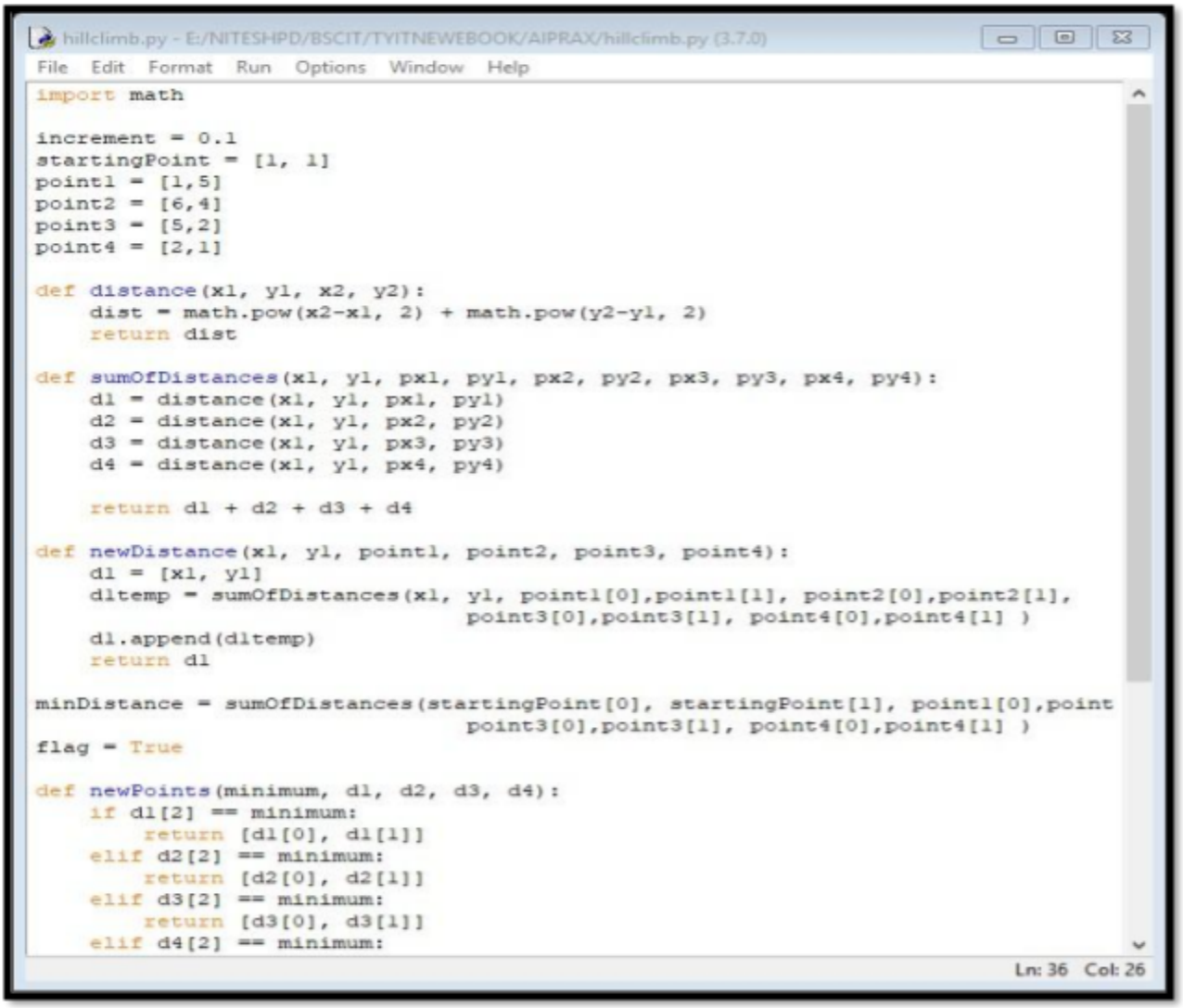
DIAGRAM:-



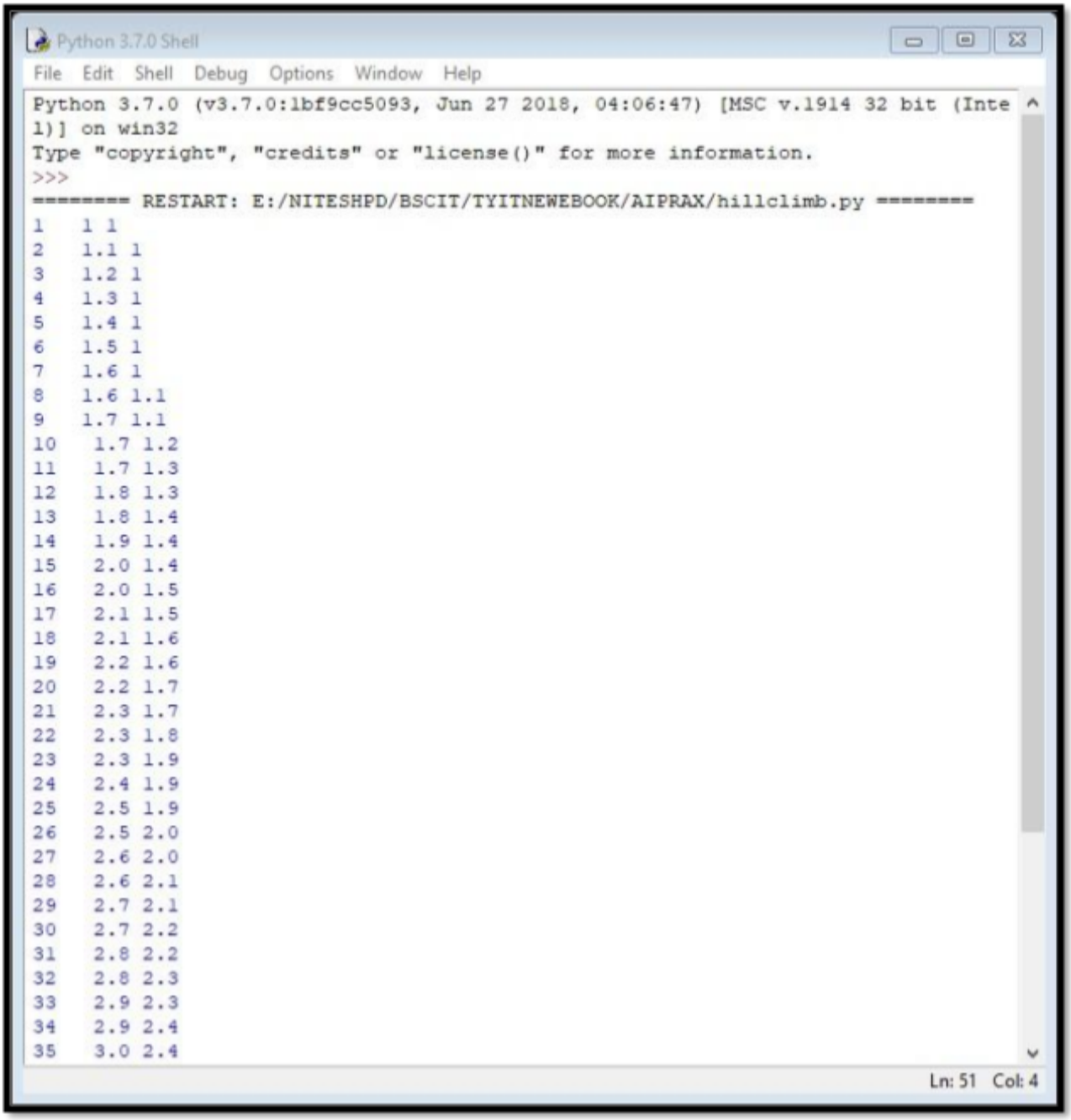

```
import math

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

max_iterations = 100  # Set a maximum number of iterations
i = 1

while i <= max_iterations:
    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)

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

    if minimum < sumOfDistances(startingPoint[0],
startingPoint[1], point1[0], point1[1], point2[0],
point2[1],
                                point3[0], point3[1],
```

```
point4[0], point4[1]):
        startingPoint = [d1[0], d1[1]] if d1[2] == minimum
else startingPoint
        startingPoint = [d2[0], d2[1]] if d2[2] == minimum
else startingPoint
        startingPoint = [d3[0], d3[1]] if d3[2] == minimum
else startingPoint
        startingPoint = [d4[0], d4[1]] if d4[2] == minimum
else startingPoint

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


hillclimb.py - E:/NITESHPD/BSCIT/TYITNEWEBOOK/AIPRAX/hillclimb.py (3.7.0)
File Edit Format Run Options Window Help
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],point
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:
Ln: 36 Col: 26

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

i = 1
while flag:
    d1 = newDistance(startingPoint[0]+increment, startingPoint[1], point1, point
    d2 = newDistance(startingPoint[0]-increment, startingPoint[1], point1, point
    d3 = newDistance(startingPoint[0], startingPoint[1]+increment, point1, point
    d4 = newDistance(startingPoint[0], startingPoint[1]-increment, point1, point
    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
```


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/TYITNEWEBOOK/AIPRAX/hillclimb.py ========
1 1 1
2 1.1 1
3 1.2 1
4 1.3 1
5 1.4 1
6 1.5 1
7 1.6 1
8 1.6 1.1
9 1.7 1.1
10 1.7 1.2
11 1.7 1.3
12 1.8 1.3
13 1.8 1.4
14 1.9 1.4
15 2.0 1.4
16 2.0 1.5
17 2.1 1.5
18 2.1 1.6
19 2.2 1.6
20 2.2 1.7
21 2.3 1.7
22 2.3 1.8
23 2.3 1.9
24 2.4 1.9
25 2.5 1.9
26 2.5 2.0
27 2.6 2.0
28 2.6 2.1
29 2.7 2.1
30 2.7 2.2
31 2.8 2.2
32 2.8 2.3
33 2.9 2.3
34 2.9 2.4
35 3.0 2.4
Ln: 51 Col: 4

Practical no-4

A. **Write a program to implement A* algorithm.**
B. **Write a program to implement AO* algorithm.**

Aim:-

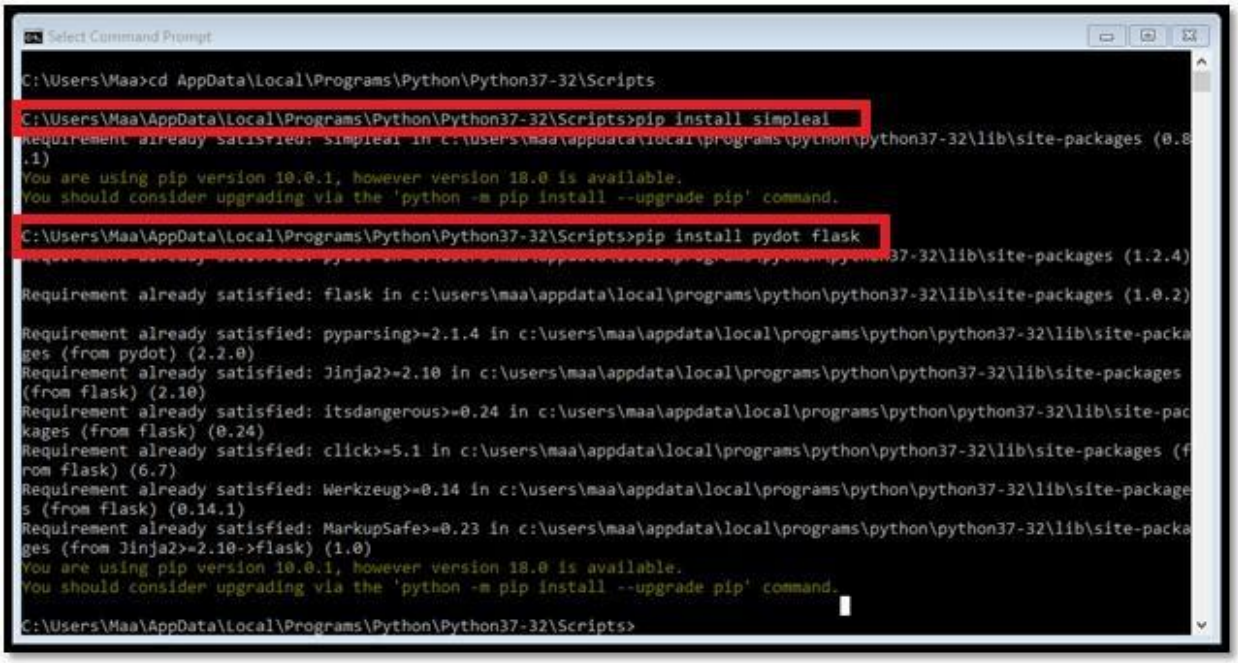
Write a program to implement A* algorithm.

Note:

Install 2 package in python scripts directory using pip command.

1. pip install simpleai

2. pip install pydot flask



PYTHON CODE:-

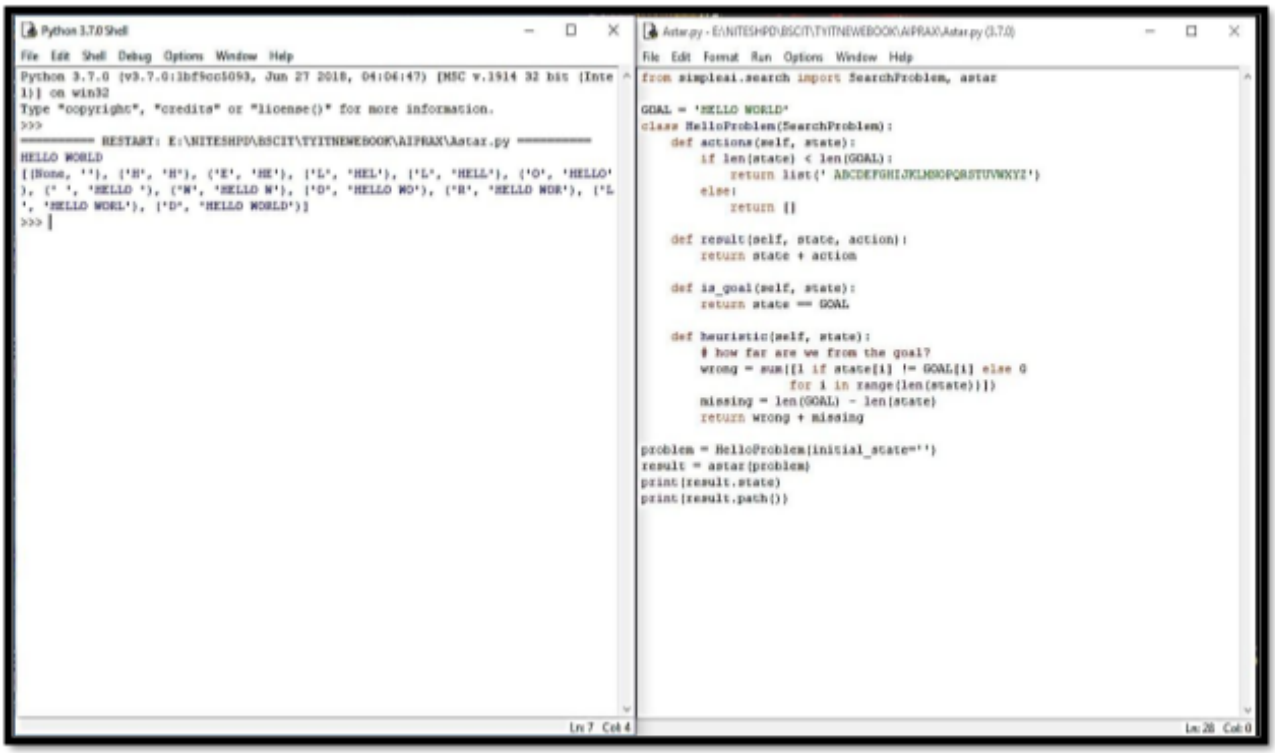
```
import heapq

class Node:
    def __init__(self, x, y, cost, heuristic):
        self.x = x
        self.y = y
        self.cost = cost
```

```
        self.heuristic = heuristic
        self.total_cost = cost + heuristic

    def __lt__(self, other):
        return self.total_cost < other.total_cost

def manhattan_distance(current, goal):
    return abs(current[0] - goal[0]) + abs(current[1] -
goal[1])

def astar(grid, start, goal):
    rows, cols = len(grid), len(grid[0])
    visited = [[False] * cols for _ in range(rows)]

    # Priority queue for open set
    open_set = []

    # Initialize start node
    start_node = Node(start[0], start[1], 0,
manhattan_distance(start, goal))
    heapq.heappush(open_set, start_node)

    while open_set:
        current_node = heapq.heappop(open_set)

        # Check if goal reached
        if (current_node.x, current_node.y) == goal:
            return True  # Goal reached

        # Mark current node as visited
        visited[current_node.x][current_node.y] = True

        # Generate neighbors
```

```
        neighbors = [
            (current_node.x - 1, current_node.y),
            (current_node.x + 1, current_node.y),
            (current_node.x, current_node.y - 1),
            (current_node.x, current_node.y + 1)
        ]

        for neighbor in neighbors:
            x, y = neighbor

            # Check if neighbor is within bounds
            if 0 <= x < rows and 0 <= y < cols and not
visited[x][y] and grid[x][y] != 1:
                cost = current_node.cost + 1
                heuristic = manhattan_distance((x, y),
goal)
                neighbor_node = Node(x, y, cost, heuristic)

                # Check if the neighbor is already in the
open set with a lower cost
                if any(node.x == x and node.y == y and
node.total_cost < neighbor_node.total_cost for node in
open_set):
                    continue

                # Add the neighbor to the open set
                heapq.heappush(open_set, neighbor_node)

    return False  # Goal not reachable

# Example usage:
grid = [
    [0, 0, 0, 0, 0],
```

```
    [0, 1, 0, 1, 0],
    [0, 1, 0, 1, 0],
    [0, 0, 0, 0, 0],
]

start = (0, 0)
goal = (3, 4)

result = astar(grid, start, goal)
print("Goal Reached!" if result else "Goal Not Reached.")
```

OUTPUT:-



Practical no-5

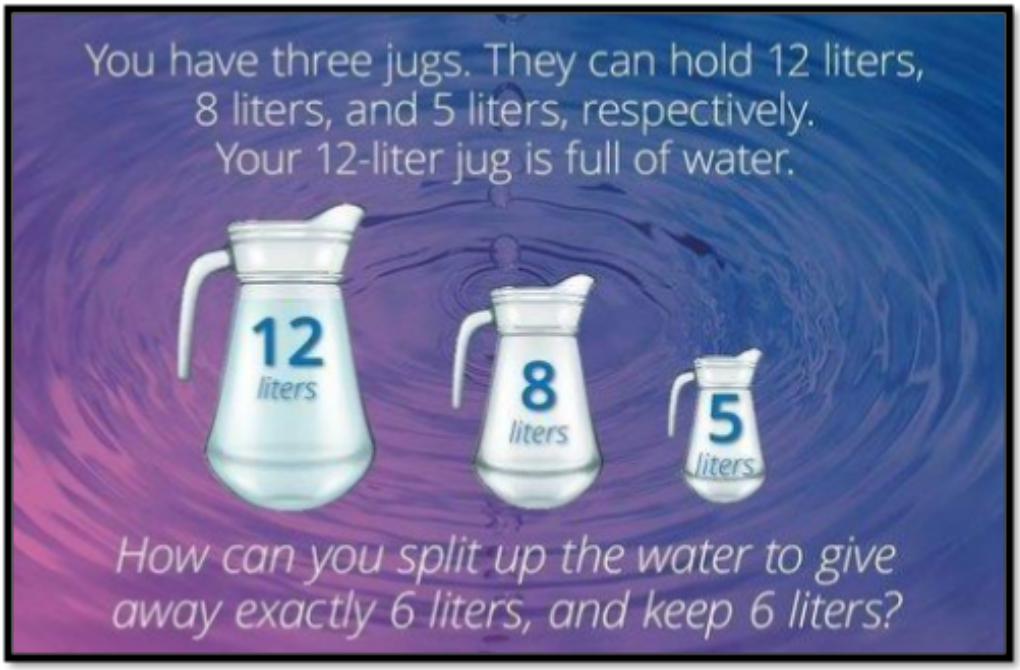
A. Write a program to solve water jug problem.

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

Aim:-

Write a program to solve water jug problem.

Diagram:-



Python Code:-

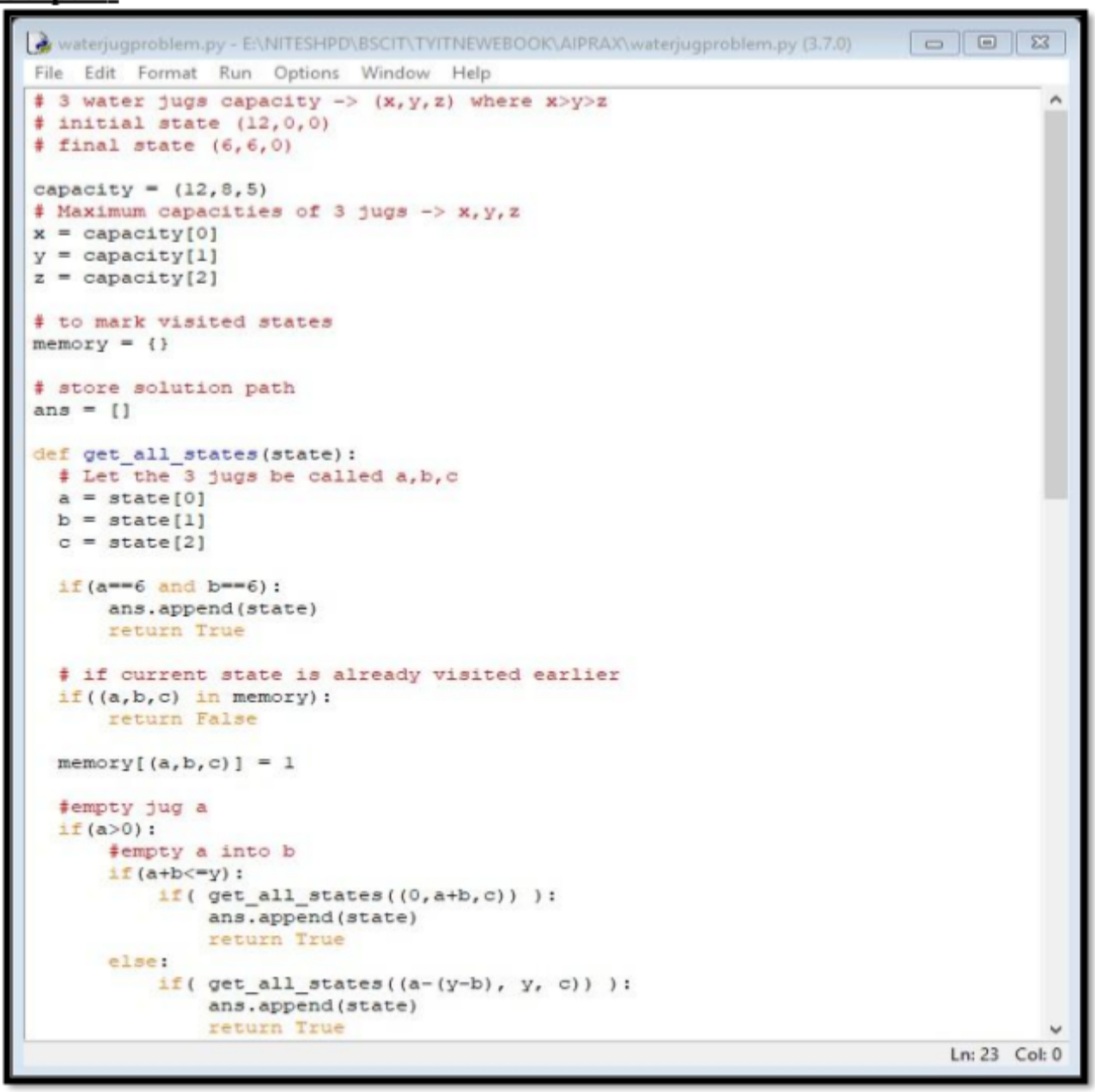
```
# 3 water jugs capacity -> (x,y,z) where x>y>z
# initial state (12,0,0)
# final state (6,6,0)

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
```

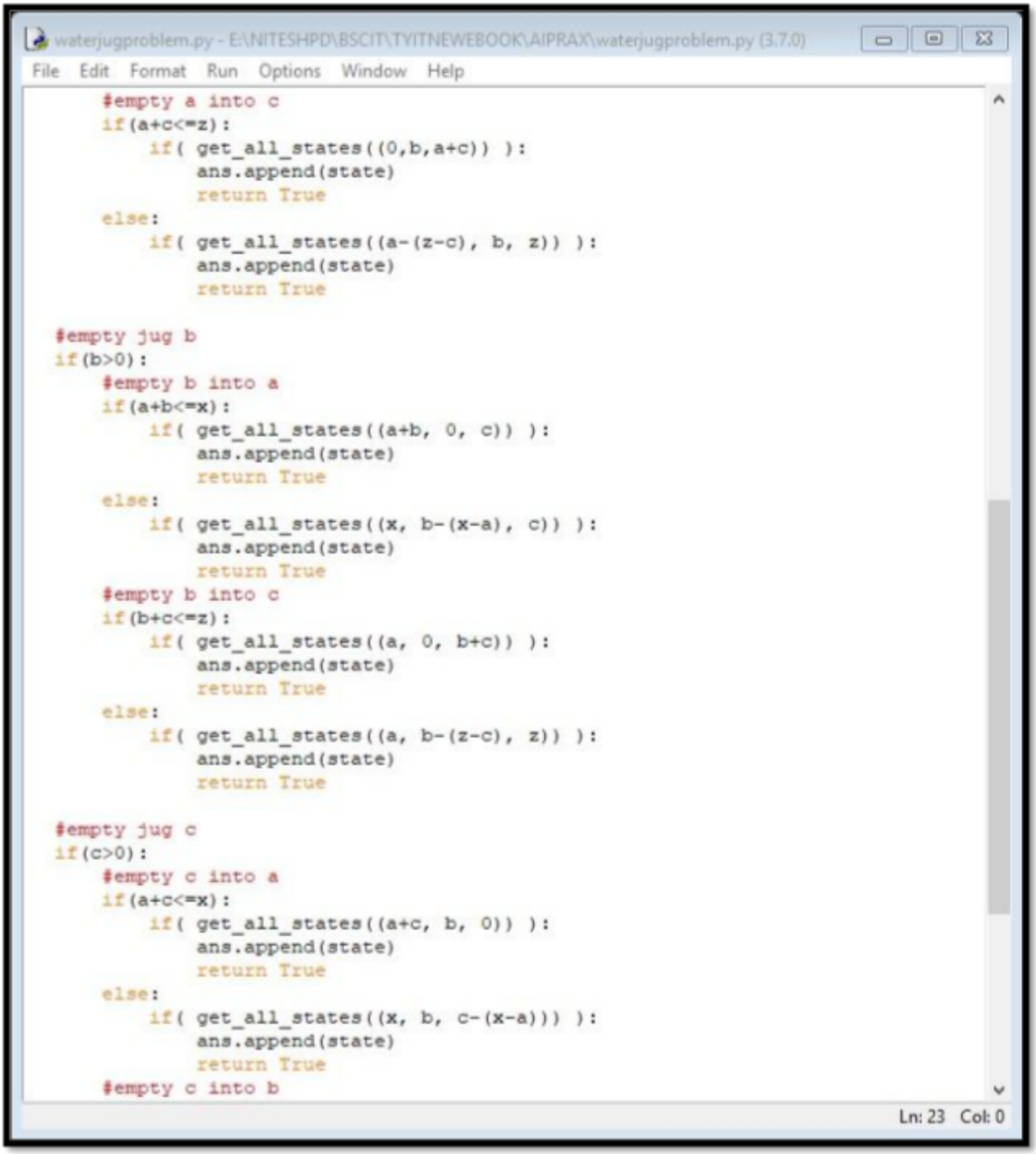
```
memory = {}

# 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:-


waterjugproblem.py - E:\NITESHPD\BSCIT\TYITNEWEBOOK\AIPRAX\waterjugproblem.py (3.7.0)
File Edit Format Run Options Window Help
# 3 water jugs capacity -> (x,y,z) where x>y>z
# initial state (12,0,0)
# final state (6,6,0)
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 = {}
# 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
Ln: 23 Col: 0


waterjugproblem.py - E:\NITESHPD\BSCIT\TYITNEWEBOOK\AIPRAX\waterjugproblem.py (3.7.0)
File Edit Format Run Options Window Help
#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
Ln: 23 Col: 0

```
        #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)
```

Python 3.7.0 Shell

File Edit Shell Debug Options Window Help

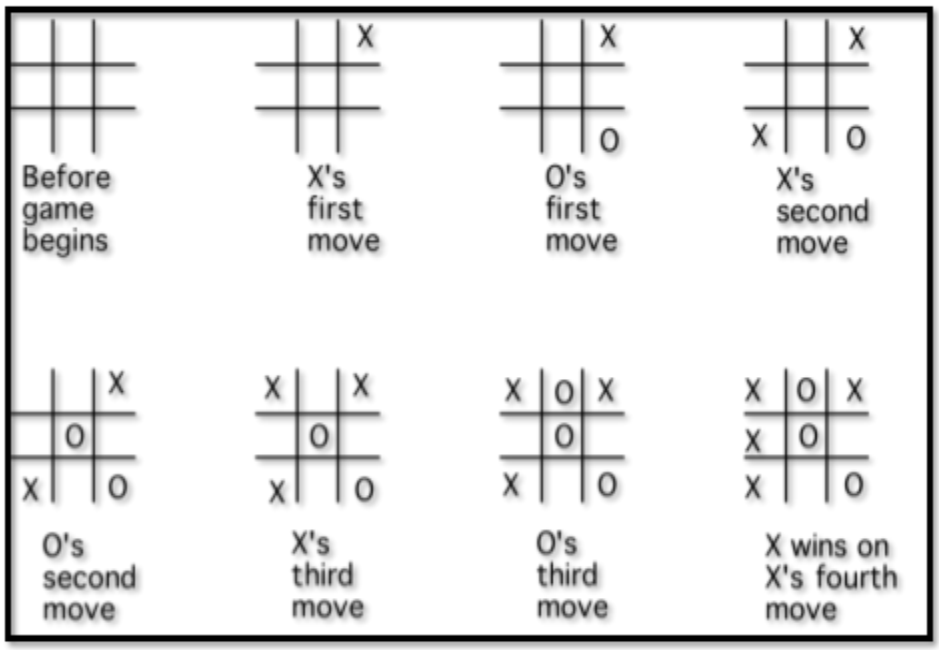
```
Python 3.7.0 (v3.7.0:1bf9cc5093, Jun 27 2018, 04:06:47) [MSC v.1914 32 bit (Inte
l)] on win32
Type "copyright", "credits" or "license()" for more information.
>>>
===== RESTART: E:\NITESHPD\BSCIT\TYITNEWEBOOK\AIPRAX\waterjugproblem.py =====
Starting work...

(12, 0, 0)
(4, 8, 0)
(0, 8, 4)
(8, 0, 4)
(8, 4, 0)
(3, 4, 5)
(3, 8, 1)
(11, 0, 1)
(11, 1, 0)
(6, 1, 5)
(6, 6, 0)
>>>
```

Ln: 18 Col: 4

Aim:-

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

Diagram:-



Python 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
```

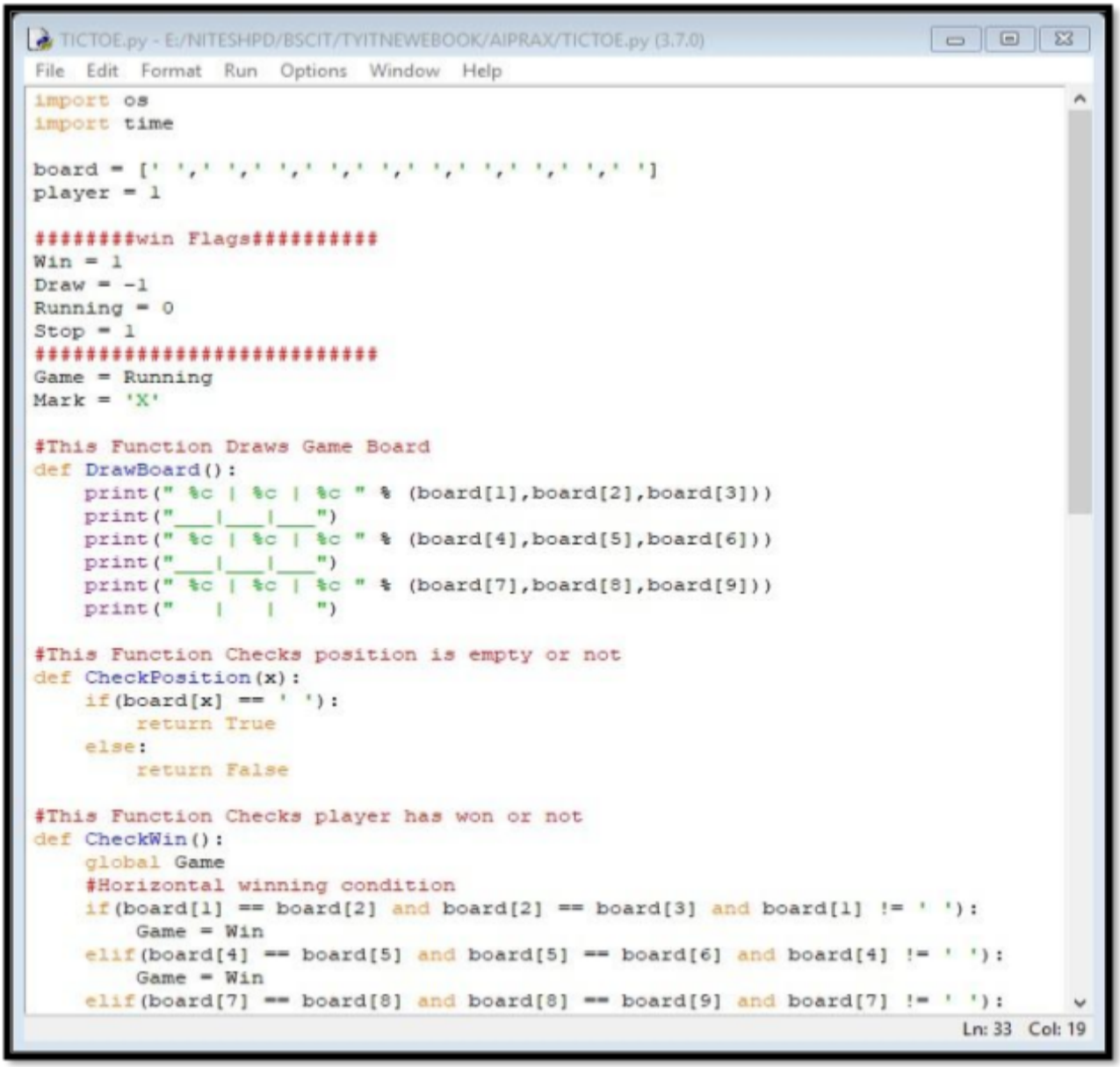
```
    elif(board[2] == board[5] and board[5] == board[8] and
board[2] != ' '):
        Game = Win
    elif(board[3] == board[6] and board[6] == board[9] and
board[3] != ' '):
        Game = Win
    # Diagonal Winning Condition
    elif(board[1] == board[5] and board[5] == board[9] and
board[5] != ' '):
        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("\nPlease 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")
```

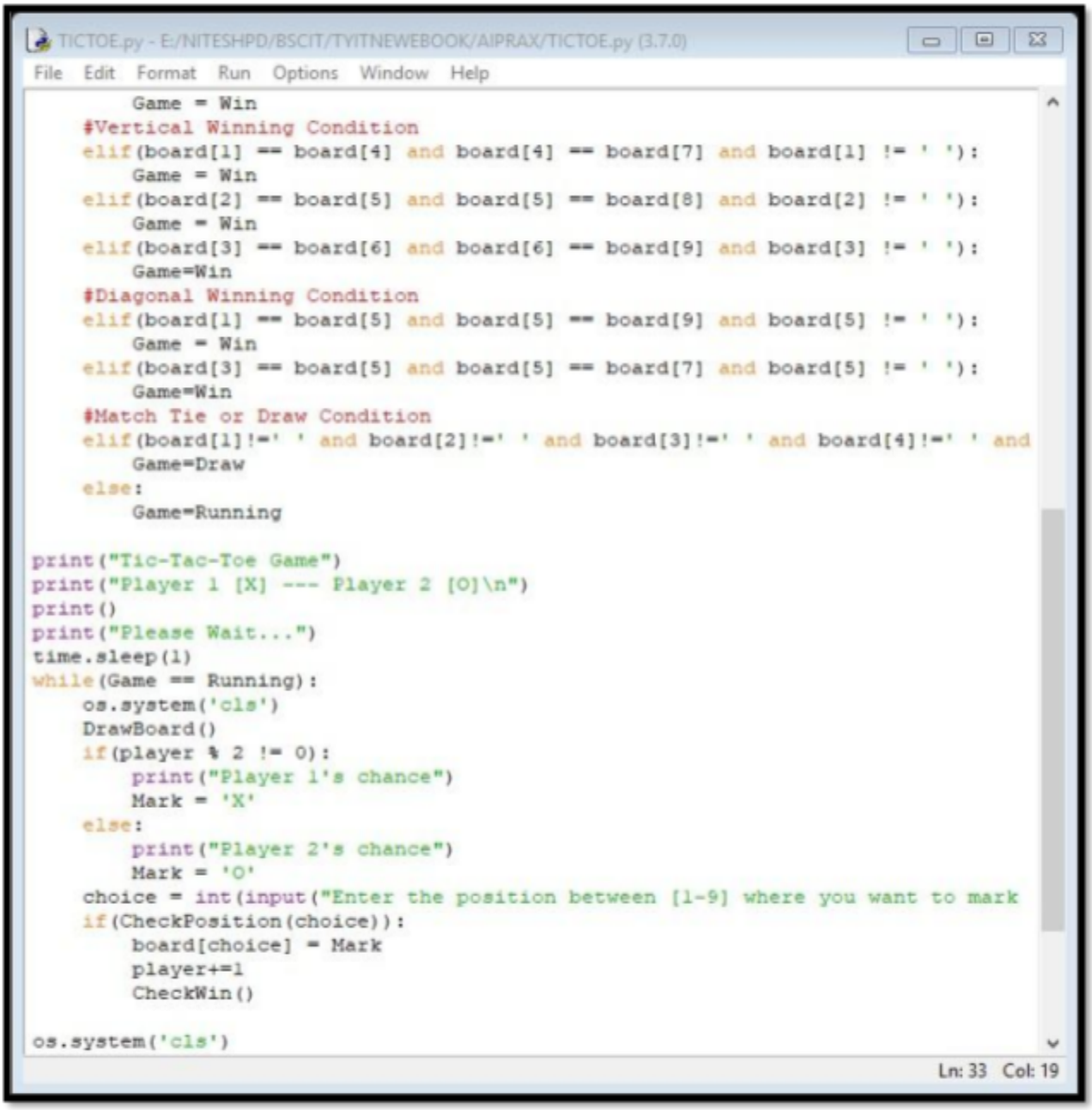
```
        Mark = 'O'
    choice = int(input("Enter the position between [1-9] where you want to mark : "))
    if(CheckPosition(choice)):
        board[choice] = Mark
        player+=1
        CheckWin()

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")
```

NOTE:- Game Rules

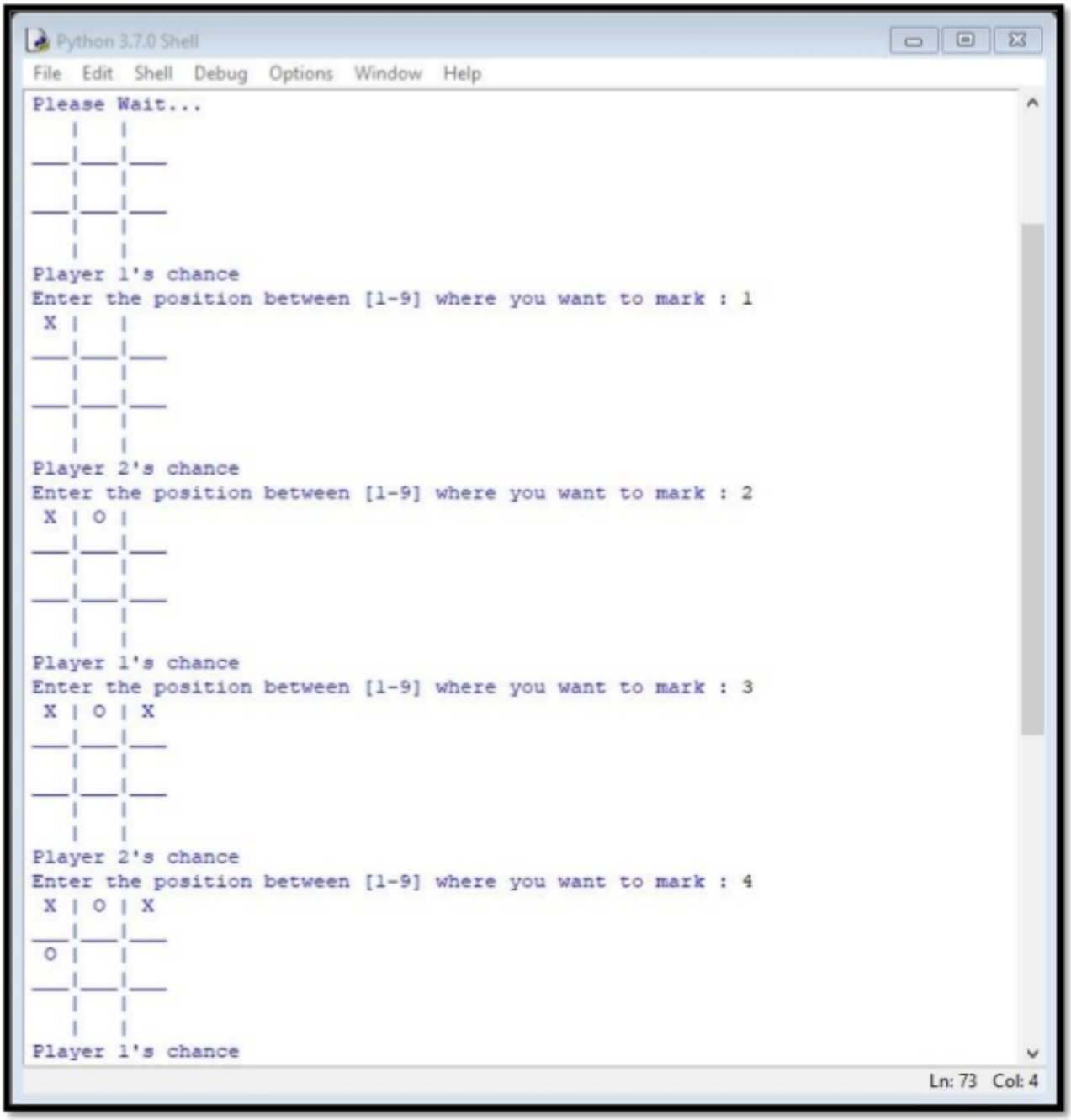
1. Traditionally the first player plays with "X". So you can decide who wants to go with "X" and who wants go with "O".
2. Only one player can play at a time.
3. If any of the players have filled a square then the other player and the same player cannot override that square.
4. There are only two conditions that may match will be draw or may win.
5. The player that succeeds in placing three respective marks (X or O) in a horizontal, vertical or diagonal row wins the game.

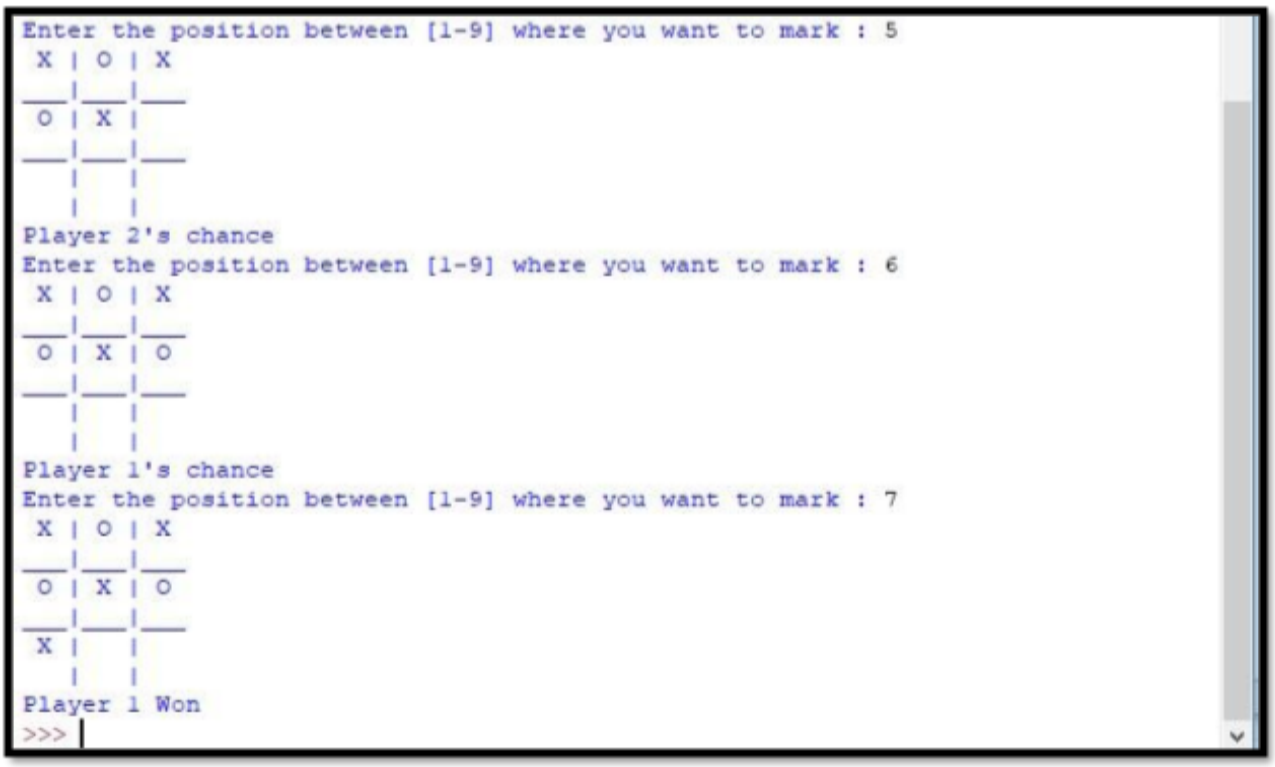
OUTPUT:-


TICTOE.py - E:/NITESHPD/BSCIT/TYITNEWEBOOK/AIPRAX/TICTOE.py (3.7.0)
File Edit Format Run Options Window Help
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] != ' '):
Ln: 33 Col: 19


TICTOE.py - E:/NITESHPD/BSCIT/TYITNEWEBOOK/AIPRAX/TICTOE.py (3.7.0)
File Edit Format Run Options Window Help
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
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')
Ln: 33 Col: 19


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")
Ln: 33 Col: 19


Python 3.7.0 Shell
File Edit Shell Debug Options Window Help
Please Wait...
Player 1's chance
Enter the position between [1-9] where you want to mark : 1
Player 2's chance
Enter the position between [1-9] where you want to mark : 2
Player 1's chance
Enter the position between [1-9] where you want to mark : 3
Player 2's chance
Enter the position between [1-9] where you want to mark : 4
Player 1's chance
Ln: 73 Col: 4


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

A. Write a program to solve Missionaries and Cannibals problem.

B. Design an application to simulate number puzzle problem.

Aim:-

Write a program to solve Missionaries and Cannibals problem.

Diagram:-

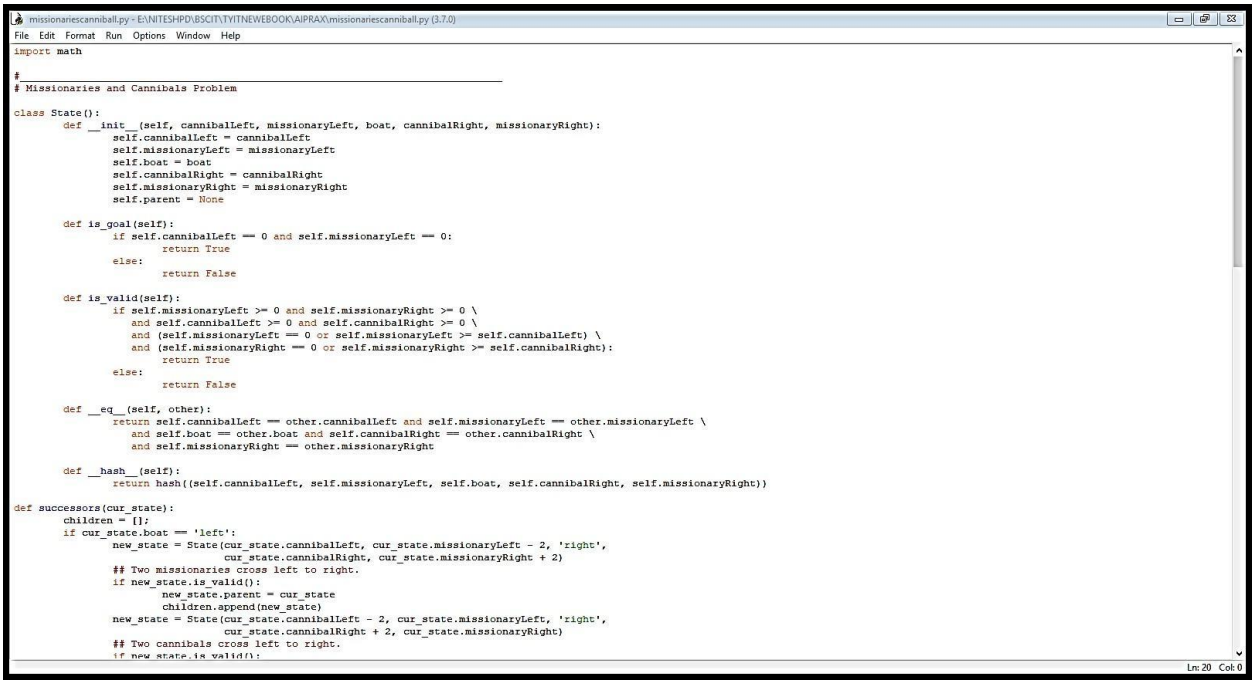
Python Code:-

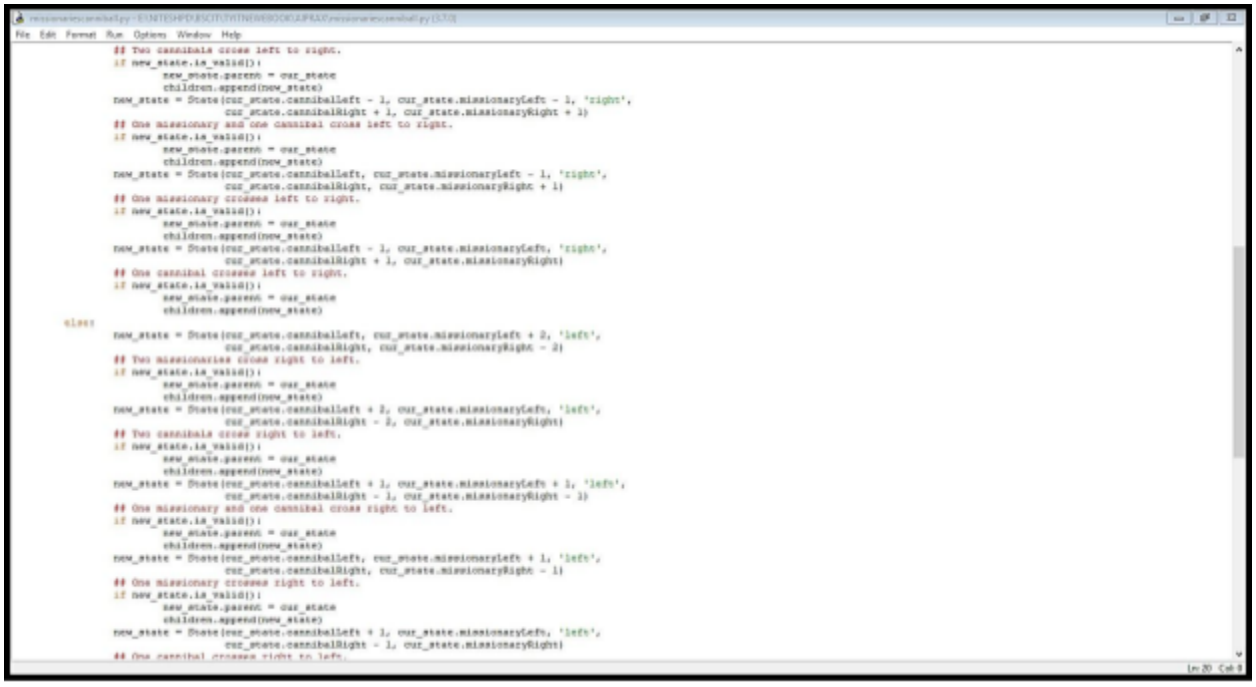
```
import math

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):
        return self.cannibalLeft == 0 and
self.missionaryLeft == 0

    def is_valid(self):
        if self.missionaryLeft < 0 or self.missionaryRight
< 0 or self.cannibalLeft < 0 or self.cannibalRight < 0:
            return False
```

```
        if (self.missionaryLeft > 0 and self.missionaryLeft
< self.cannibalLeft) or \
           (self.missionaryRight > 0 and
self.missionaryRight < self.cannibalRight):
            return False
        return True

    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':
        for i in range(3):
            for j in range(3):
                if 1 <= i + j <= 2:
                    new_state =
State(cur_state.cannibalLeft - i, cur_state.missionaryLeft
- j, 'right',

cur_state.cannibalRight + i, cur_state.missionaryRight + j)
                    if new_state.is_valid():
                        new_state.parent = cur_state
```

```
                        children.append(new_state)
    else:
        for i in range(3):
            for j in range(3):
                if 1 <= i + j <= 2:
                    new_state =
State(cur_state.cannibalLeft + i, cur_state.missionaryLeft
+ j, 'left',

cur_state.cannibalRight - i, cur_state.missionaryRight - j)
                    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 = [initial_state]
    explored = set()
    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 and child not in
frontier:
                frontier.append(child)
    return None
```

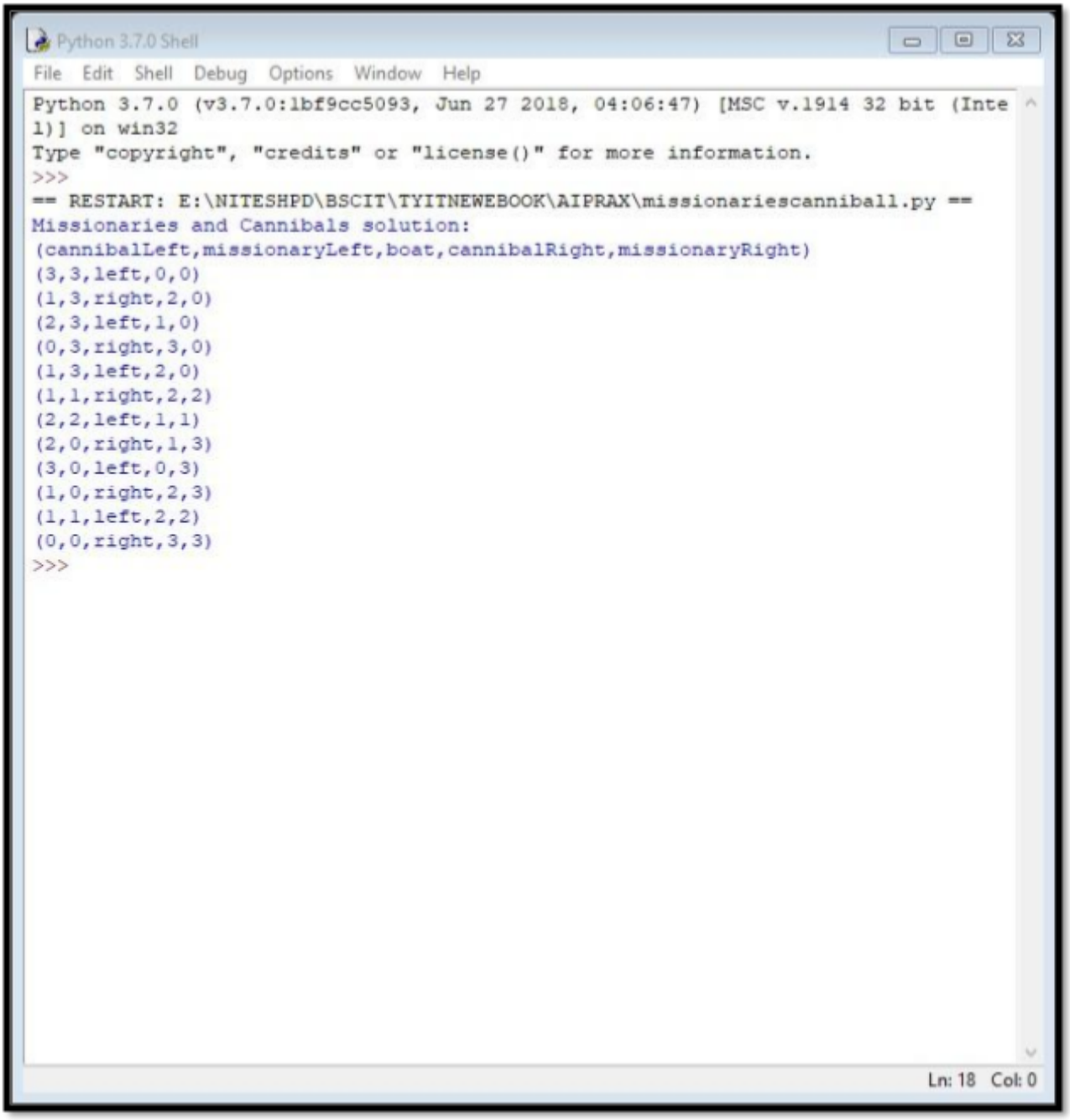
```
def print_solution(solution):
    path = []
    path.append(solution)
    parent = solution.parent
    while parent:
        path.append(parent)
        parent = parent.parent
    for t in range(len(path)):
        state = path[len(path) - t - 1]
        print("(" + str(state.cannibalLeft) + "," +
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,miss
ionaryRight)")
    print_solution(solution)

if __name__ == "__main__":
    main()
```


missionariescannibal.py - E:\NITESHPD\BSCIT\TYITNEWEBOOK\AIPRAX\missionariescannibal.py (3.7.0)
File Edit Format Run Options Window Help
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():
Ln: 20 Col: 0



```
File  Edit  Format  Run  Options  Window  Help
                ## 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.
Ln: 20  Col: 0
```

```
File  Edit  Format  Run  Options  Window  Help
                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()
Ln: 20  Col: 0
```



AIM:-

Design an application to simulate number puzzle problem.

PYHTON CODE:-

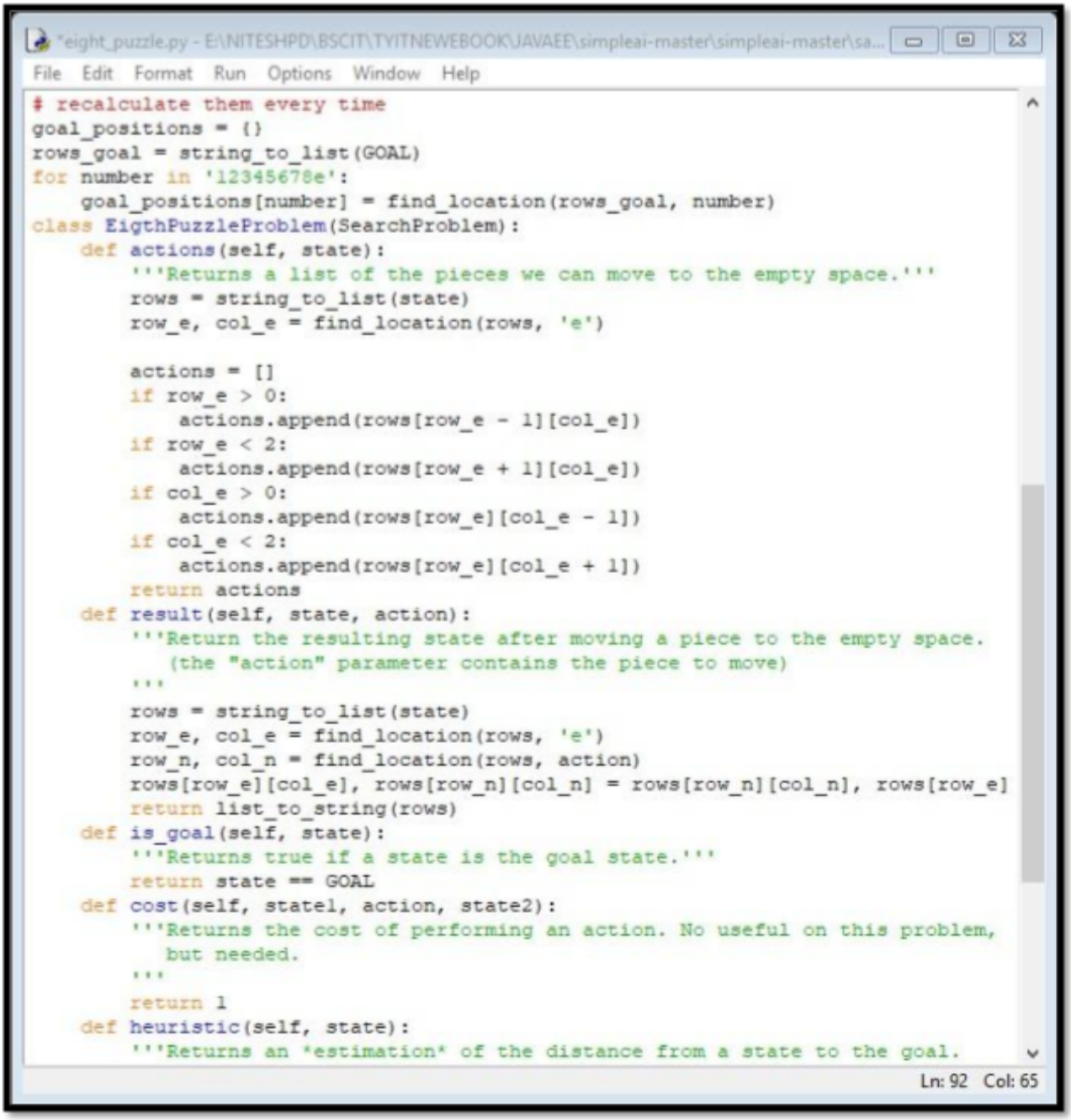
```
'''
8 puzzle problem, a smaller version of the fifteen puzzle:
States are defined as string representations of the pieces
on the puzzle.
Actions denote what piece will be moved to the empty space.
States must always be immutable. We will use strings, but
internally most of the time we will convert 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 simpleai.search import astar, SearchProblem
from simpleai.search.viewers import WebViewer

GOAL = '''1-2-3
4-5-6
8-e'''

INITIAL = '''4-1-2
7-e-3
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,
           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:


*eight_puzzle.py - E:\NITESHPD\BSCIT\TYITNEWEBOOK\JAVAEE\simpleai-master\simpleai-master\sa...
File Edit Format Run Options Window Help
# 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]
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,
but needed.
'''
return 1
def heuristic(self, state):
'''Returns an 'estimation' of the distance from a state to the goal.
Ln: 92 Col: 65

```
        '''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))
# if you want to use the visual debugger, use this instead:
# result = astar(EigthPuzzleProblem(INITIAL), viewer=WebViewer())
for action, state in result.path():
    print('Move number', action)
    print(state)
```

Ln: 92 Col: 65

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:-

```
import random

# List holders for cards
cardfaces = []
suits = ["Hearts", "Diamonds", "Clubs", "Spades"]
royals = ["J", "Q", "K", "A"]
deck = []

# Adding numbers 2-10 as strings to cardfaces list
for i in range(2, 11):
    cardfaces.append(str(i))
```

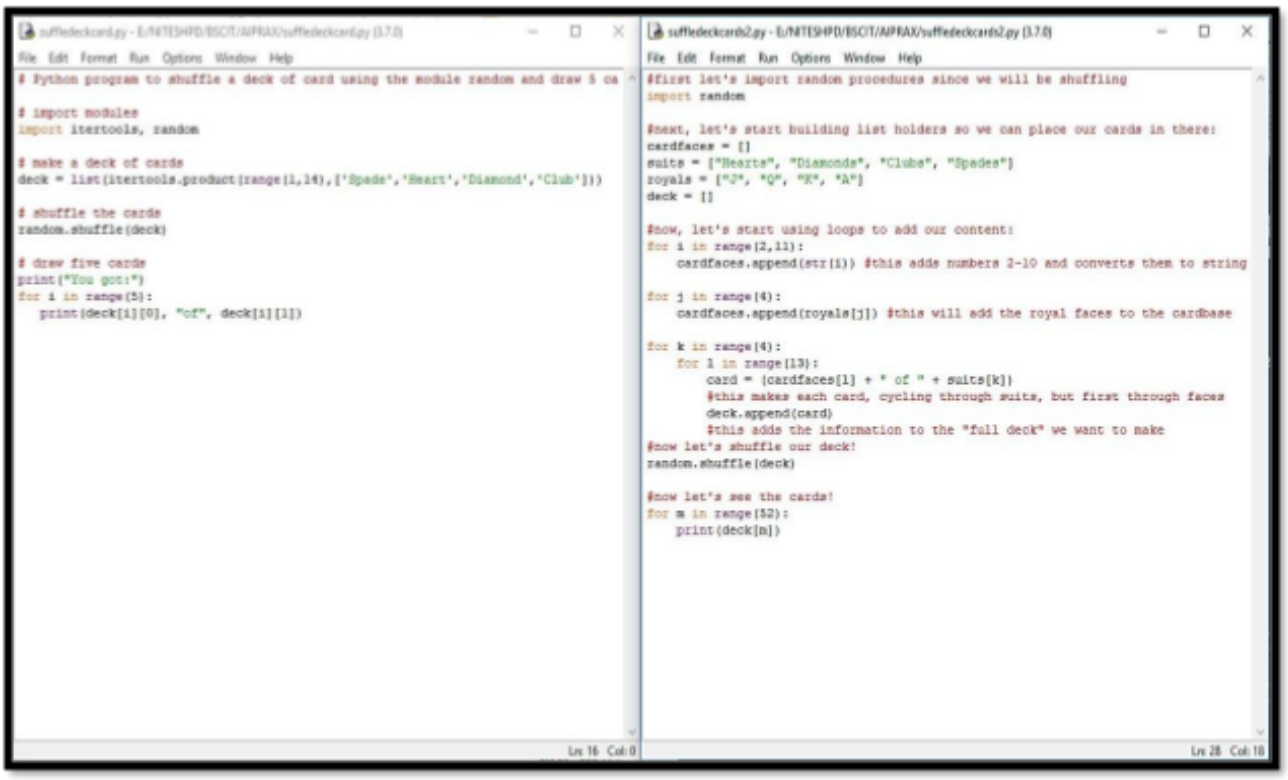
```
# Adding royal faces to the card base
for j in range(4):
    cardfaces.append(royals[j])

# Generating the deck of cards
for k in range(4):
    for l in range(13):
        card = (cardfaces[l] + " of " + suits[k])
        deck.append(card)

# Shuffling the deck
random.shuffle(deck)

# Displaying the shuffled deck
for m in range(52):
    print(deck[m])
```

OR

```
import itertools
import random

# Making a deck of cards
deck = list(itertools.product(range(1, 14), ['Spade',
'Heart', 'Diamond', 'Club']))

# Shuffling the cards
random.shuffle(deck)

# Drawing five cards
print("You got:")
for i in range(5):
```

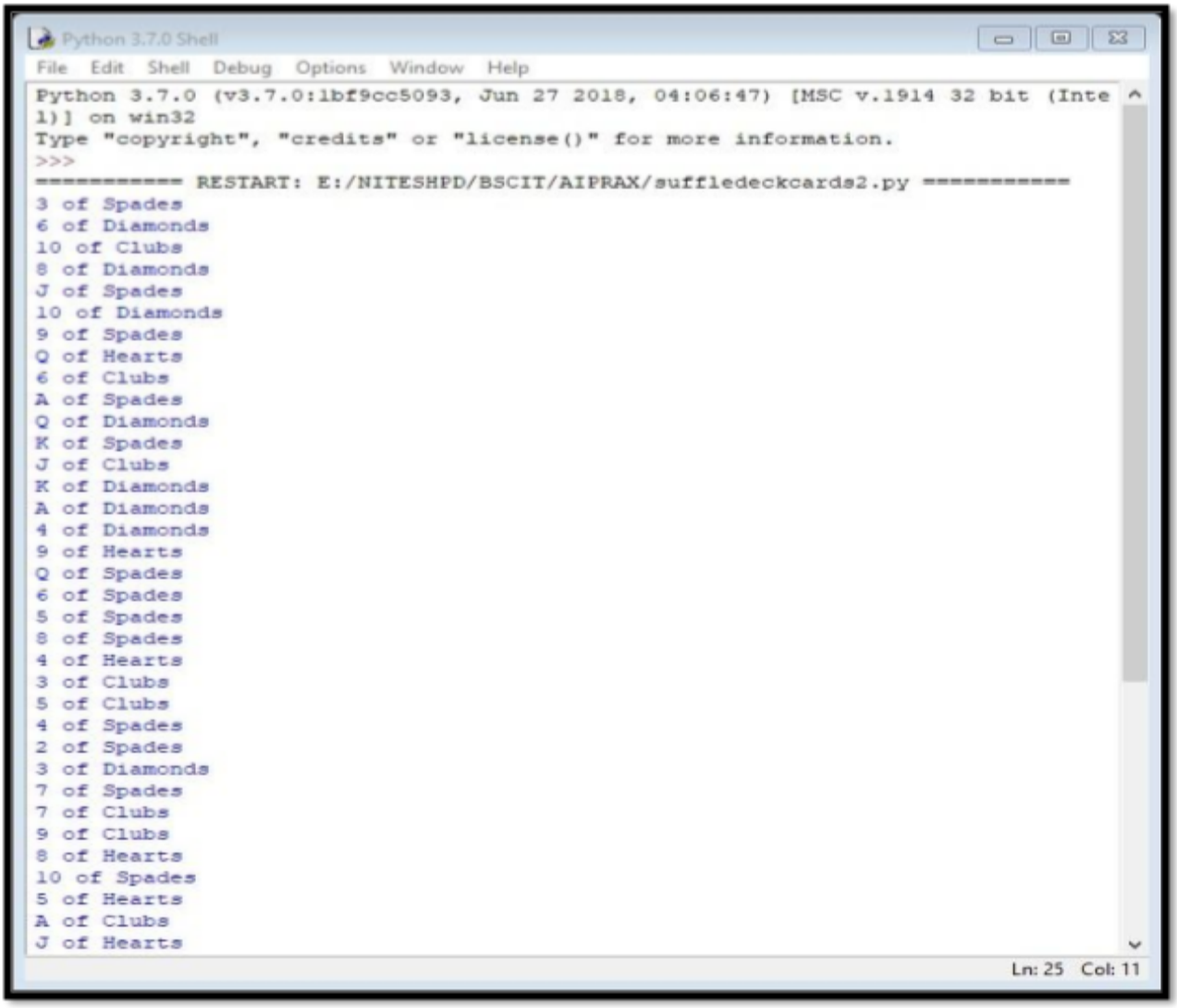
```
print(deck[i][0], "of", deck[i][1])
```

Output:-

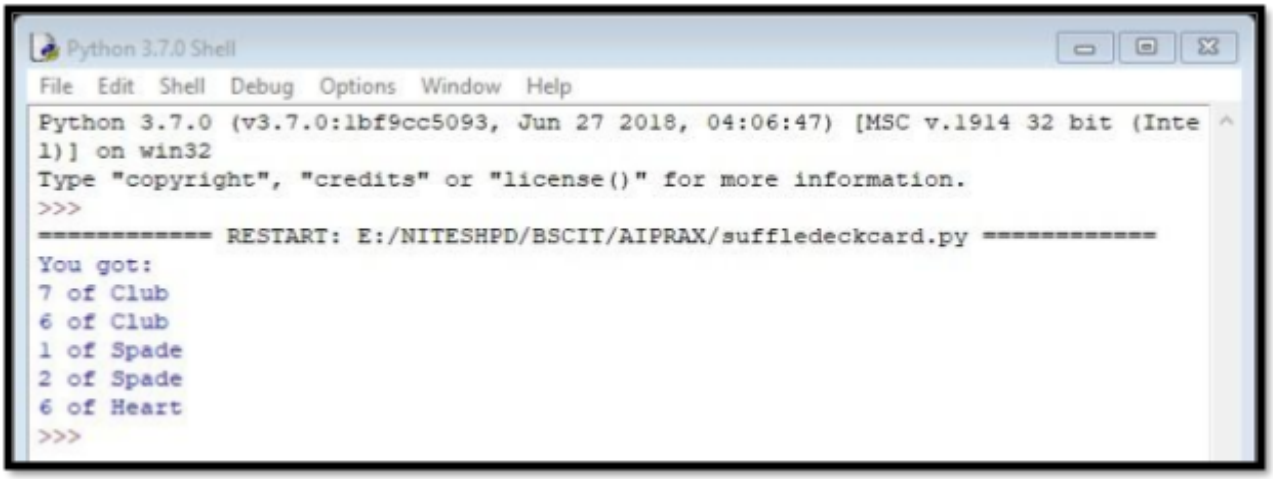


```
# Python program to shuffle a deck of card using the module random and draw 5 ca

# 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])
```

```
#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

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])
```


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/suffledeckcards2.py ==========
3 of Spades
6 of Diamonds
10 of Clubs
8 of Diamonds
J of Spades
10 of Diamonds
9 of Spades
Q of Hearts
6 of Clubs
A of Spades
Q of Diamonds
K of Spades
J of Clubs
K of Diamonds
A of Diamonds
4 of Diamonds
9 of Hearts
Q of Spades
6 of Spades
5 of Spades
8 of Spades
4 of Hearts
3 of Clubs
5 of Clubs
4 of Spades
2 of Spades
3 of Diamonds
7 of Spades
7 of Clubs
9 of Clubs
8 of Hearts
10 of Spades
5 of Hearts
A of Clubs
J of Hearts
Ln: 25 Col: 11

OR


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

A. Solve the block of World problem.

B. Solve constraint satisfaction problem

Aim:-

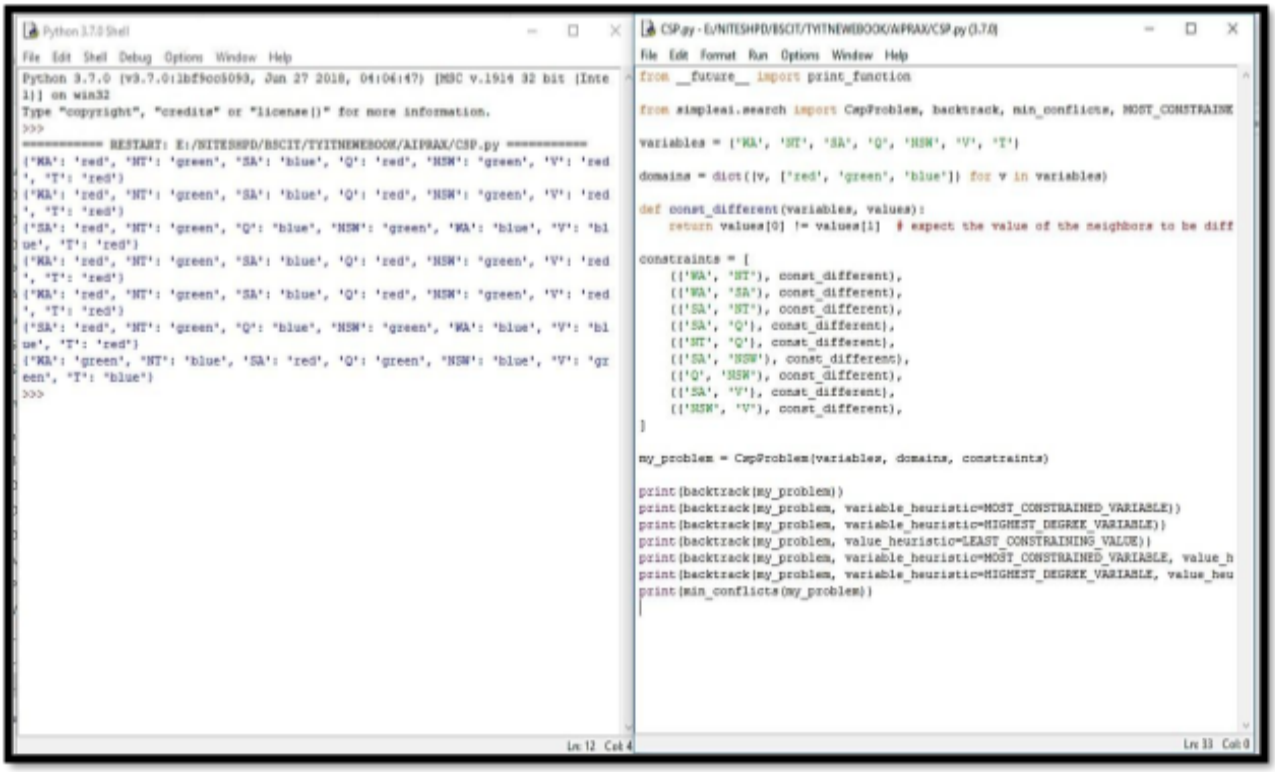
Implementation Of Constraints Satisfactions Problem

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:-


Python 3.7.0 Shell
File Edit Shell Debug Options Window Help
Python 3.7.0 (v3.7.0:1bf9cc5093, Jan 27 2018, 04:06:47) [MSC v.1914 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>>
========== RESTART: E:/BITESHPD/BSCIT/TYITNEWEBOOK/AIPRAX/CSP.py ==========
{'WA': 'red', 'NT': 'green', 'SA': 'blue', 'Q': 'red', 'NSW': 'green', 'V': 'red', 'T': 'red'}
{'WA': 'red', 'NT': 'green', 'SA': 'blue', 'Q': 'red', 'NSW': 'green', 'V': 'red', 'T': 'red'}
{'SA': 'red', 'NT': 'green', 'Q': 'blue', 'NSW': 'green', 'WA': 'blue', 'V': 'blue', 'T': 'red'}
{'WA': 'red', 'NT': 'green', 'SA': 'blue', 'Q': 'red', 'NSW': 'green', 'V': 'red', 'T': 'red'}
{'WA': 'red', 'NT': 'green', 'SA': 'blue', 'Q': 'red', 'NSW': 'green', 'V': 'red', 'T': 'red'}
{'SA': 'red', 'NT': 'green', 'Q': 'blue', 'NSW': 'green', 'WA': 'blue', 'V': 'blue', 'T': 'red'}
{'WA': 'green', 'NT': 'blue', 'SA': 'red', 'Q': 'green', 'NSW': 'blue', 'V': 'green', 'T': 'blue'}
>>>
Ln: 12 Col: 4
CSP.py - E:/NITESHPD/BSCIT/TYITNEWEBOOK/AIPRAX/CSP.py (3.7.0)
File Edit Format Run Options Window Help
from __future__ import print_function
from simpleai.search import CspProblem, backtrack, min_conflicts, MOST_CONSTRAINE
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 diff
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_h
print(backtrack(my_problem, variable_heuristic=HIGHEST_DEGREE_VARIABLE, value_heu
print(min_conflicts(my_problem))
Ln: 33 Col: 0

PRACTICAL No.-9

A. Derive the expressions based on Associative law

B. Derive the expressions based on Distributive law

PRACTICAL No.-10

C. Write a program to derive the predicate. (for e.g.: Sachin is batsman , batsman is cricketer) - > Sachin is Cricketer.

D. Write a program which contains three predicates: male, female, parent. Make rules for following family relations: father, mother, grandfather,grandmother, brother, sister, uncle, aunt, nephew and niece, cousin.

Question:

i. Draw Family Tree

ii. Define: Clauses, Facts, Predicates and Rules with conjunction and disjunction