AI EXPERIMENT-3

Khushi Arora (RA1911003010198)

AIM: Implementation of Crypto Arithmetic Puzzle.

ALGORITHM USED:

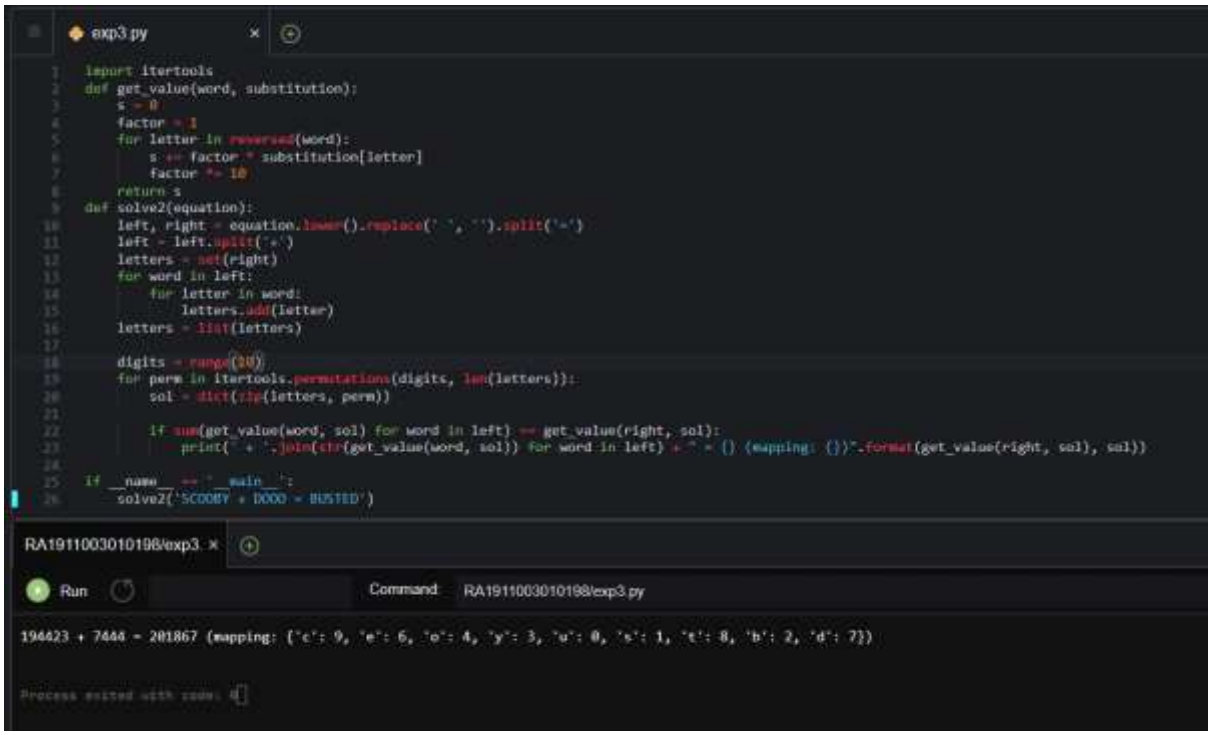
- There should be a unique digit to be replaced with a unique alphabet.
- The result should satisfy the predefined arithmetic rules, i.e., 2+2 =4, nothing else.
- Digits should be from 0-9 only.
- There should be only one carry forward, while performing the addition operation on a problem.
- The problem can be solved from both sides, i.e., lefthand side (L.H.S), or righthand side (R.H.S)

CODE USED:

```
import itertools

def get_value(word, substitution):
    s = 0
    factor = 1
    for letter in reversed(word):
        s += factor * substitution[letter]
        factor *= 10
    return s
def solve2(equation):
    left, right = equation.lower().replace(' ', '').split('=')
    left = left.split('+')
    letters = set(right)
    for word in left:
        for letter in word:
            letters.add(letter)
    letters = list(letters)
```

```
    digits = range(10)
    for perm in itertools.permutations(digits, len(letters)):
        sol = dict(zip(letters, perm))

        if sum(get_value(word, sol) for word in left) == get_value(right, sol):
            print(' + '.join(str(get_value(word, sol)) for word in left) + " = {} (mapping: {})".format(get_value(right, sol), sol))

if __name__ == '__main__':
    solve2('SCOOBY + DOOO = BUSTED')
```

OUTPUT:



RESULT: Crypto Arithmetic Puzzle is successfully implemented.

Khushi Drara
RA1911003010199

classmate
Date
Page

Experiment - 3

SCOOBY + DOOO = BUSTED

```
  SCOOBY
+  DOOO
-------
 BUSTED
```

$$I + S = B \quad -(1)$$
$$B - S = I \quad -(2)$$

After applying every permutation and combination, the best fit solution is given below:-

```
  1  9  4  4  2  3
        7  4  4  4
  -----------------
  2  0  1  8  6  7
```

SCOOBY = 194423
DOOO = 7444
BUSTED = 201867

2 + 0 + 1 + 8 + 6 + 7 = 24.

AI EXPERIMENT-04

Submitted By: Khushi Arora

Aim: Implementation and Analysis of DFS and BFS for water jug problem.

Procedure:

1. The state space for this problem can be described as the set of ordered pairs of integers (x,y).
Start State: (0,0)
Goal State: (2,0)
2. Generate production rules for the water jug problem.
3.We basically perform three operations to achieve the goal.

- Fill water jug.
- Empty water jug
- and Transfer water jug

S.No.	Initial State	Condition	Final state	Description of action taken
1.	(x,y)	If x<4	(4,y)	Fill the 4 gallon jug completely
2.	(x,y)	if y<3	(x,3)	Fill the 3 gallon jug completely
3.	(x,y)	If x>0	(x-d,y)	Pour some part from the 4 gallon jug
4.	(x,y)	If y>0	(x,y-d)	Pour some part from the 3 gallon jug
5.	(x,y)	If x>0	(0,y)	Empty the 4 gallon jug
6.	(x,y)	If y>0	(x,0)	Empty the 3 gallon jug
7.	(x,y)	If (x+y)<7	(4, y-[4-x])	Pour some water from the 3 gallon jug to fill the four gallon jug
8.	(x,y)	If (x+y)<7	(x-[3-y],y)	Pour some water from the 4 gallon Jug to fill the 3 gallon jug.
9.	(x,y)	If (x+y)<4	(x+y,0)	Pour all water from 3 gallon jug to the 4 gallon jug
10.	(x,y)	if (x+y)<3	(0, x+y)	Pour all water from the 4 gallon jug to the 3 gallon jug

Solution of water jug problem according to the production rules:

S.No.	4 gallon jug contents	3 gallon jug contents	Rule followed
1.	0 gallon	0 gallon	Initial state
2.	0 gallon	3 gallons	Rule no.2
3.	3 gallons	0 gallon	Rule no. 9
4.	3 gallons	3 gallons	Rule no. 2
5.	4 gallons	2 gallons	Rule no. 7
6.	0 gallon	2 gallons	Rule no. 5
7.	2 gallons	0 gallon	Rule no. 9

Code Used:

->USING BFS (PYTHON)

```
from collections import deque

def BFS(a, b, target):
    m = {}
    isSolvable = False
    path = []

    q = deque()

    q.append((0, 0))

    while (len(q) > 0):

        u = q.popleft()

        if ((u[0], u[1]) in m):
            continue

        if ((u[0] > a or u[1] > b or
             u[0] < 0 or u[1] < 0)):
            continue

        path.append([u[0], u[1]])

        m[(u[0], u[1])] = 1
```

```
        if (u[0] == target or u[1] == target):
            isSolvable = True

            if (u[0] == target):
                if (u[1] != 0):

                    path.append([u[0], 0])
            else:
                if (u[0] != 0):

                    path.append([0, u[1]])

            sz = len(path)
            for i in range(sz):
                print("(", path[i][0], ",",
                        path[i][1], ")")
            break

        q.append([u[0], b])
        q.append([a, u[1]])

        for ap in range(max(a, b) + 1):

            c = u[0] + ap
            d = u[1] - ap

            if (c == a or (d == 0 and d >= 0)):
                q.append([c, d])

            c = u[0] - ap
            d = u[1] + ap

            if ((c == 0 and c >= 0) or d == b):
                q.append([c, d])

        q.append([a, 0])

        q.append([0, b])

    if (not isSolvable):
        print ("No solution")

if __name__ == '__main__':
```

```
    Jug1, Jug2, target = 4, 3, 2
    print("Path from initial state "
          "to solution state ::")

    BFS(Jug1, Jug2, target)
```

->USING DFS (CPP)

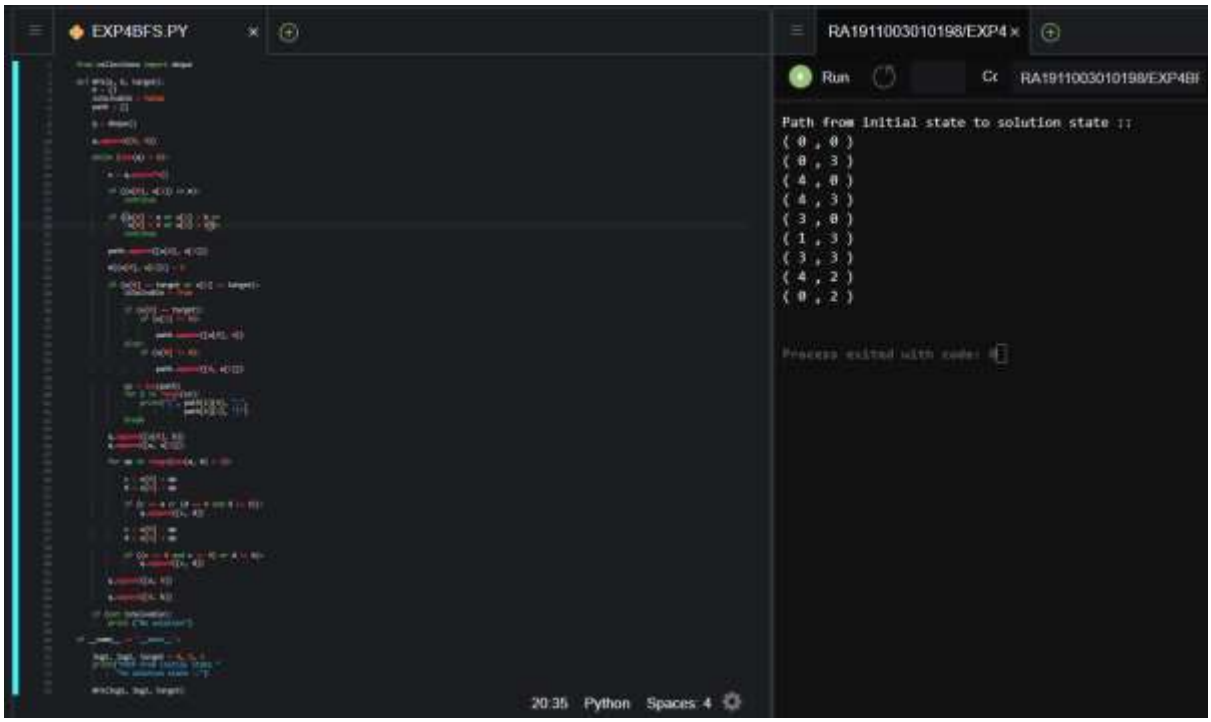
```
#include<stdio.h>
struct node
{
    int x, y;
    struct node *next;
}*root, *left, *right;

int main()
{
    int jug1, jug2, f1, f2;
    printf("Capacity of jug1 : ");
    scanf("%d", &jug1);
    printf("Capacity of jug2 : ");
    scanf("%d", &jug2);
    printf("Required water in jug1 : ");
    scanf("%d", &f1);
    printf("Required water in jug2 : ");
    scanf("%d", &f2);
    generateTree(jug1, jug2, f1, f2);
    DFS();
}

int isNodePresent(struct node *next, int jug1, int jug2, int f1, int f2)
{
    struct node *temp;
    if((next->x == f1) && (next->y == f2)){
        return(0);
    }
    if((next->x == jug1) && (next->y == jug2)){
        return(1);
    }
    if((next->x == 0) && (next->y == 0)){
        return(1);
    }
```

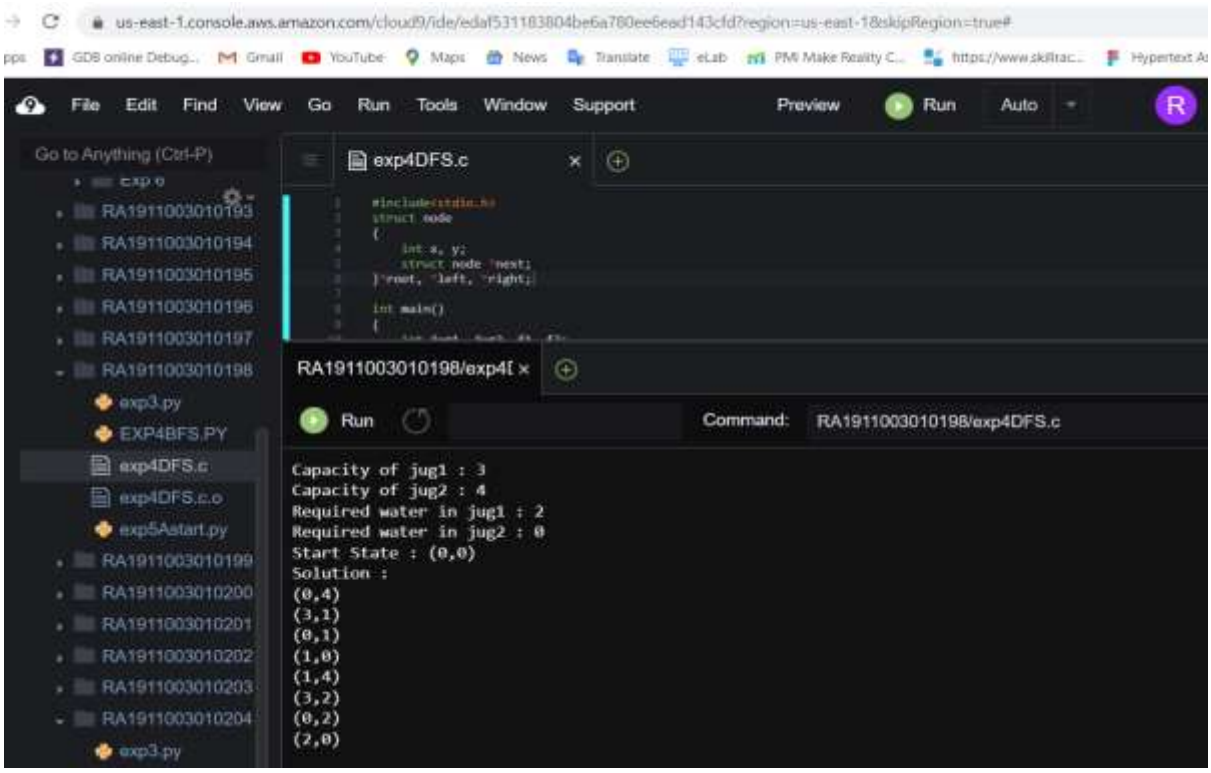
```
   temp = left;
   while(1)
   {
      if((temp->x == next->x) && (temp->y == next->y)){
         return(1);
      }
      else if(temp->next == NULL){
         break;
      } else {
         temp = temp->next;
      }
   }
   temp = right;
   while(1)
   {
      if((temp->x == next->x) && (temp->y == next->y)){
         return(1);
      } else if(temp->next == NULL){
         break;
      }
      temp = temp->next;
   }
   return(0);
}

void DFS()
{
   struct node *temp;
   temp = left;
   printf("Start State : (%d,%d)\n", root->x, root->y);
   printf("Solution : \n");
   while(1)
   {
      printf("(%d,%d)\n", temp->x, temp->y);
      if(temp->next == NULL){
         break;
      }
      temp = temp->next;
   }
   temp = right;
}


struct node* genNewState(struct node *current, int jug1, int jug2, int f1, int f2)
```

```
{
   int d;
   struct node *next;
   next = (struct node*)malloc(sizeof(struct node));
   next->x = jug1;
   next->y = current->y;
   if(isNodePresent(next, jug1, jug2, f1, f2) != 1){
      return(next);
   }
   next->x = current->x;
   next->y = jug2;
   if(isNodePresent(next, jug1, jug2, f1, f2) != 1){
      return(next);
   }
   next->x = 0;
   next->y = current->y;
   if(isNodePresent(next, jug1, jug2, f1, f2) != 1){
      return(next);
   }
   next->y = 0;
   next->x = current->x;
   if(isNodePresent(next, jug1, jug2, f1, f2) != 1){
      return(next);
   }
   if((current->y < jug2) && (current->x != 0))
   {
      d = jug2 - current->y;
      if(d >= current->x)
      {
         next->x = 0;
         next->y = current->y + current->x;
      } else {
         next->x = current->x - d;
         next->y = current->y + d;
      }
      if(isNodePresent(next, jug1, jug2, f1, f2) != 1){
         return(next);
      }
   }
   if((current->x < jug1) && (current->y != 0))
   {
      d = jug1 - current->x;
      if(d >= current->y) {
         next->y = 0;
```

```
            next->x = current->x + current->y;
        } else {
            next->y = current->y - d;
            next->x = current->x + d;
        }
        if(isNodePresent(next, jug1, jug2, f1, f2) != 1){
            return(next);
        }
    }
    return(NULL);
}

void generateTree(int jug1, int jug2, int f1, int f2)
{
    int flag1, flag2;
    struct node *tempLeft, *tempRight;
    root  = (struct node*)malloc(sizeof(struct node));
    root->x = 0; root->y = 0; root->next = NULL;
    left = (struct node*)malloc(sizeof(struct node));
    left->x = 0; left->y = jug2; left->next = NULL;
    right = (struct node*)malloc(sizeof(struct node));
    right->x = jug1; right->y = 0; right->next = NULL;
    tempLeft = left;
    tempRight = right;
    while(1)
    {
        flag1 = 0; flag2 = 0;
        if((tempLeft->x != f1) || (tempLeft->y != f2))
        {
            tempLeft->next = genNewState(tempLeft, jug1, jug2, f1, f2);
            tempLeft = tempLeft->next;
            tempLeft->next = NULL;
            flag1 = 1;
        }
        if((tempRight->x != f1) || (tempRight->y != f2))
        {
            tempRight->next = genNewState(tempRight, jug1, jug2, f1, f2);
            tempRight = tempRight->next;
            tempRight->next = NULL;
            flag2 = 1;
        }
        if((flag1 == 0) && (flag2 == 0))
            break;
    } }
```

OUTPUT





Result: Analysis of DFS and BFS for water jug problem is successfully implemented.

Khushi Arora
RA1911003010198

Experiment - 4

⇒ Water Jar Problem.

We will represent a state of the problem as a tuple (x, y) where x represents the amount of water in 4-gallon jug and y represents the amount of water in the 3-gallon jug.

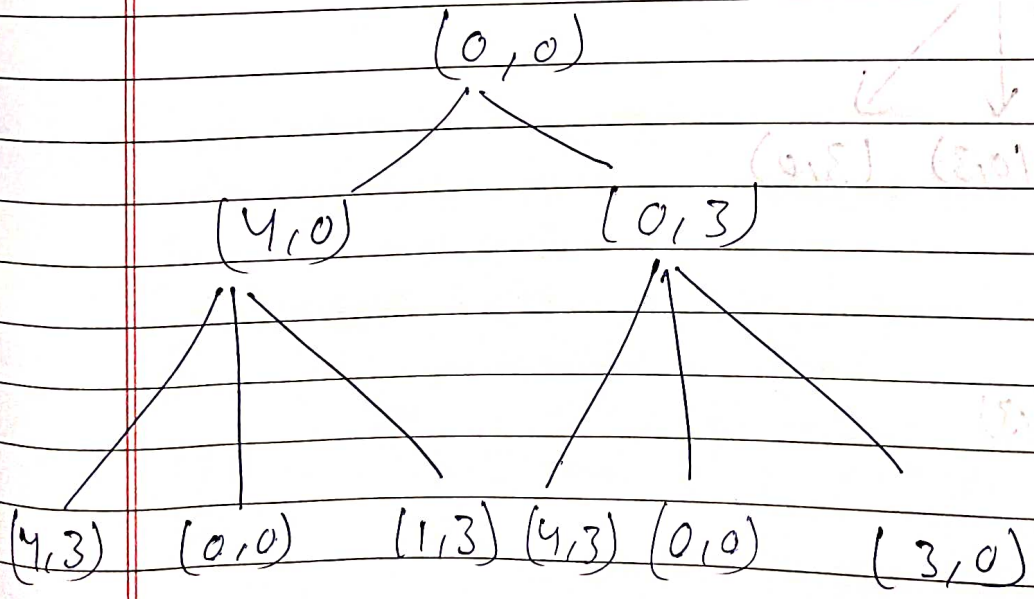
Initial state as (0, 0)
goal state as (2, y)

Production rule

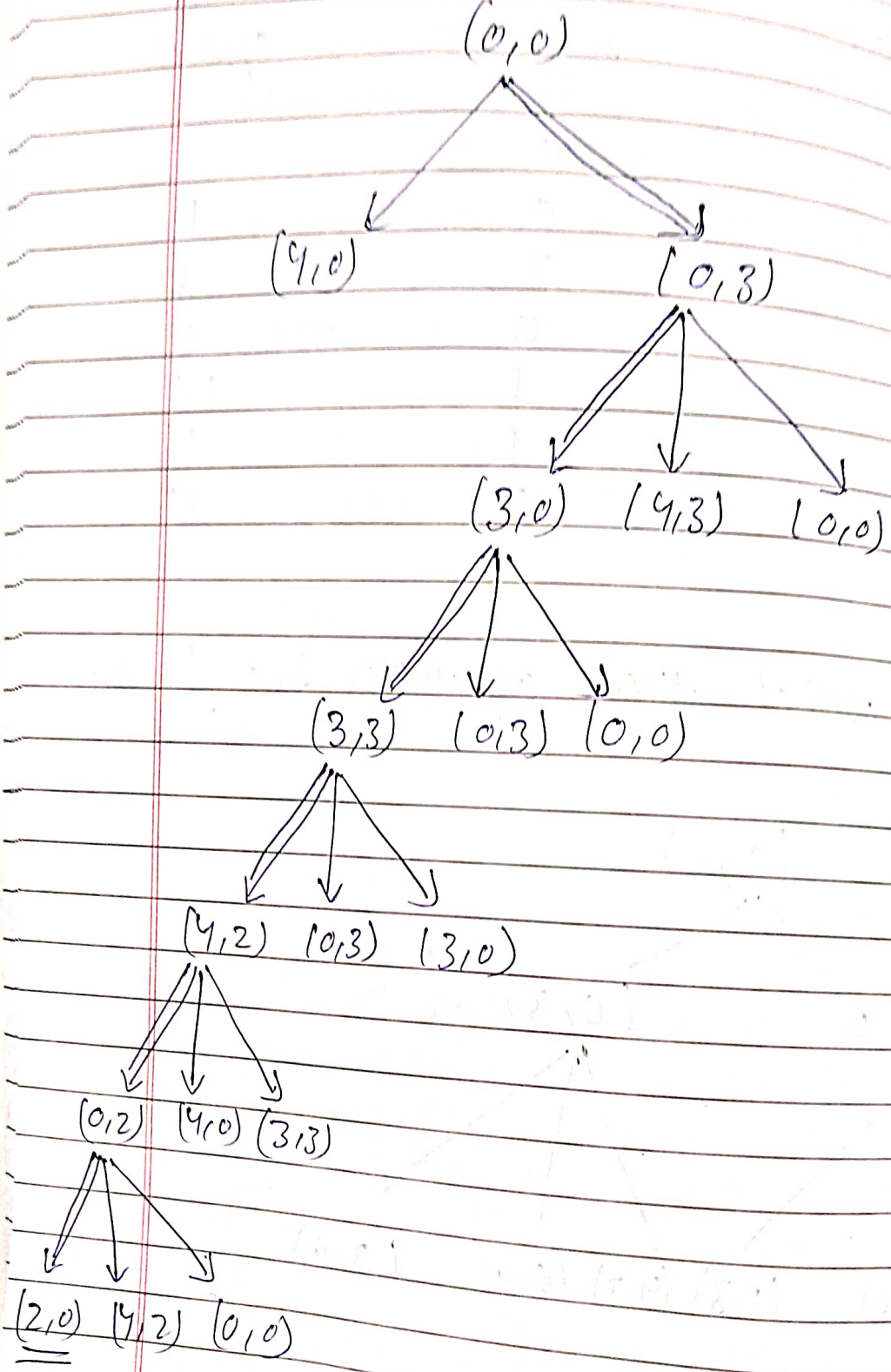
1)	(x, y) if $x < 4$	→ (4, y)
2)	(x, y) if $y < 3$	→ (x, 3)
3)	(x, y) if $x > 0$	→ (x − d, y)
4)	(x, y) if $y > 0$	→ (x, y − d)
5)	(x, y) if $x > 0$	→ (0, y)
6)	(x, y) if $y > 0$	→ (x, 0)
7)	(x, y) if $(x+y) >= 4$ and $y > 0$)	→ (4, y − (4 − x))
8)	(x, y) if $(x+y >= 3$ and $x > 0)$	→ (x − (3 − y), 3)
9)	(x, y) if $(x+y <= 4$ and $y > 0)$	→ (x + y, 0)
10)	(x, y) if $(x+y <= 3$ and $> 0)$	→ (0, x + y)
11)	(0, 2)	→ (2, 0)

4 gallan Jug	3 gallan Jug	Rule applied
0	0	
4	0	1
1	3	8
1	0	6
0	1	10
4	1	1
2	3	8

⇒ Search tree of water Jug problem.



→ State space tree



AI EXPERIMENT-05

Submitted By: Khushi Arora (RA1911003010198)

Aim: Solving 8-Puzzle using A* Algorithm.

N-Puzzle or sliding puzzle is a popular puzzle that consists of N tiles where N can be 8, 15, 24, and so on. In our example N = 8. The puzzle is divided into sqrt(N+1) rows and sqrt(N+1) columns. Eg. 15-Puzzle will have 4 rows and 4 columns and an 8-Puzzle will have 3 rows and 3 columns. The puzzle consists of N tiles and one empty space where the tiles can be moved. Start and Goal configurations (also called state) of the puzzle are provided. The puzzle can be solved by moving the tiles one by one in the single empty space and thus achieving the Goal configuration. Instead of moving the tiles in the empty space, we can visualize moving the empty space in place of the tile, basically swapping the tile with the empty space. The empty space can only move in four directions viz.,

- Up
- Down
- Right
- Left

The empty space cannot move diagonally and can take only one step at a time (i.e. move the empty space one position at a time).
f-score = h-score + g-score
A* uses a combination of heuristic value (h-score: how far the goal node is) as well as the g-score (i.e. the number of nodes traversed from the start node to current node).

Procedure:

1. We first move the empty space in all the possible directions in the start state and calculate the f-score for each state. This is called expanding the current state.

2. After expanding the current state, it is pushed into the closed list and the newly generated states are pushed into the open list.

3. A state with the least f-score is selected and expanded again. This process continues until the goal state occurs as the current state. Basically, here we are providing the algorithm a measure to choose its actions. The algorithm chooses the best possible action and proceeds in that path.

4. This solves the issue of generating redundant child states, as the algorithm will expand the node with the least f-score.

Algorithm of A*

1. Initialize the open list
2. Initialize the closed list

   put the starting node on the open

   list (you can leave its f at zero)

3. while the open list is not empty

   a) find the node with the least f on

      the open list, call it "q"

   b) pop q off the open list

   c) generate q's 8 successors and set their

      parents to q

   d) for each successor

      i) if successor is the goal, stop search

      ii) else, compute both g and h for successor

      successor.g = q.g + distance between

                    successor and q

      successor.h = distance from goal to

      successor (This can be done using many

      ways, we will discuss three heuristics-

      Manhattan, Diagonal and Euclidean

      Heuristics)

      successor.f = successor.g + successor.h

iii) if a node with the same position as
successor is in the OPEN list which has a
lower f than successor, skip this successor

iV) if a node with the same position as
successor is in the CLOSED list which has
a lower f than successor, skip this successor
otherwise, add the node to the open list
end (for loop)

e) push q on the closed list
end (while loop)

Code Used:

```
class Node:
    def __init__(self,data,level,fval):
        """ Initialize the node with the data, level of the node and the calculated fvalue """
        self.data = data
        self.level = level
        self.fval = fval
    def generate_child(self):
        """ Generate child nodes from the given node by moving the blank space
            either in the four directions {up,down,left,right} """
        x,y = self.find(self.data,'_')
        """ val_list contains position values for moving the blank space in either of
            the 4 directions [up,down,left,right] respectively. """
        val_list = [[x,y-1],[x,y+1],[x-1,y],[x+1,y]]
        children = []
```

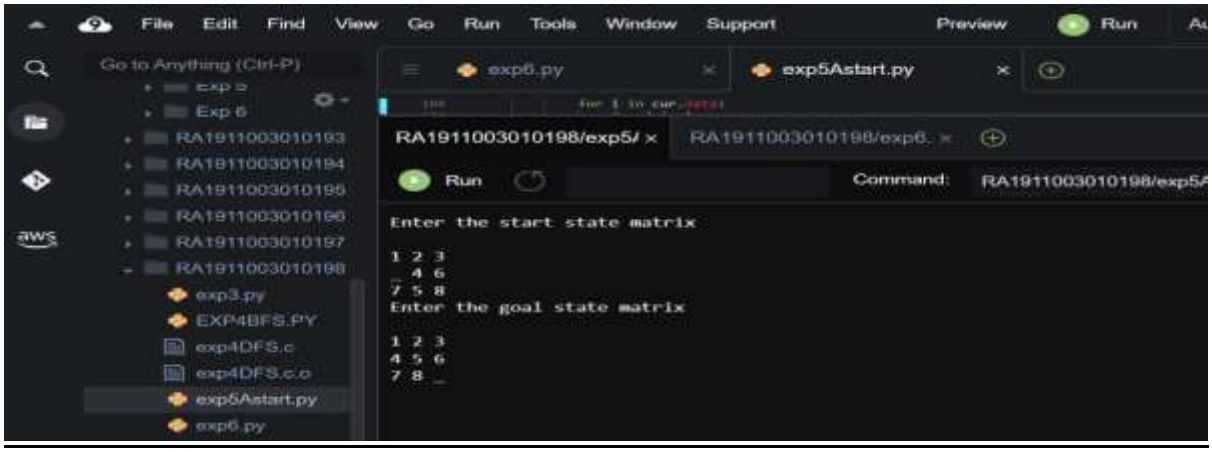
```
    for i in val_list:
        child = self.shuffle(self.data,x,y,i[0],i[1])
        if child is not None:
            child_node = Node(child,self.level+1,0)
            children.append(child_node)
    return children
def shuffle(self,puz,x1,y1,x2,y2):
    """ Move the blank space in the given direction and if the position value are out
        of limits the return None """
    if x2 >= 0 and x2 < len(self.data) and y2 >= 0 and y2 < len(self.data):
        temp_puz = []
        temp_puz = self.copy(puz)
        temp = temp_puz[x2][y2]
        temp_puz[x2][y2] = temp_puz[x1][y1]
        temp_puz[x1][y1] = temp
        return temp_puz
    else:
        return None

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

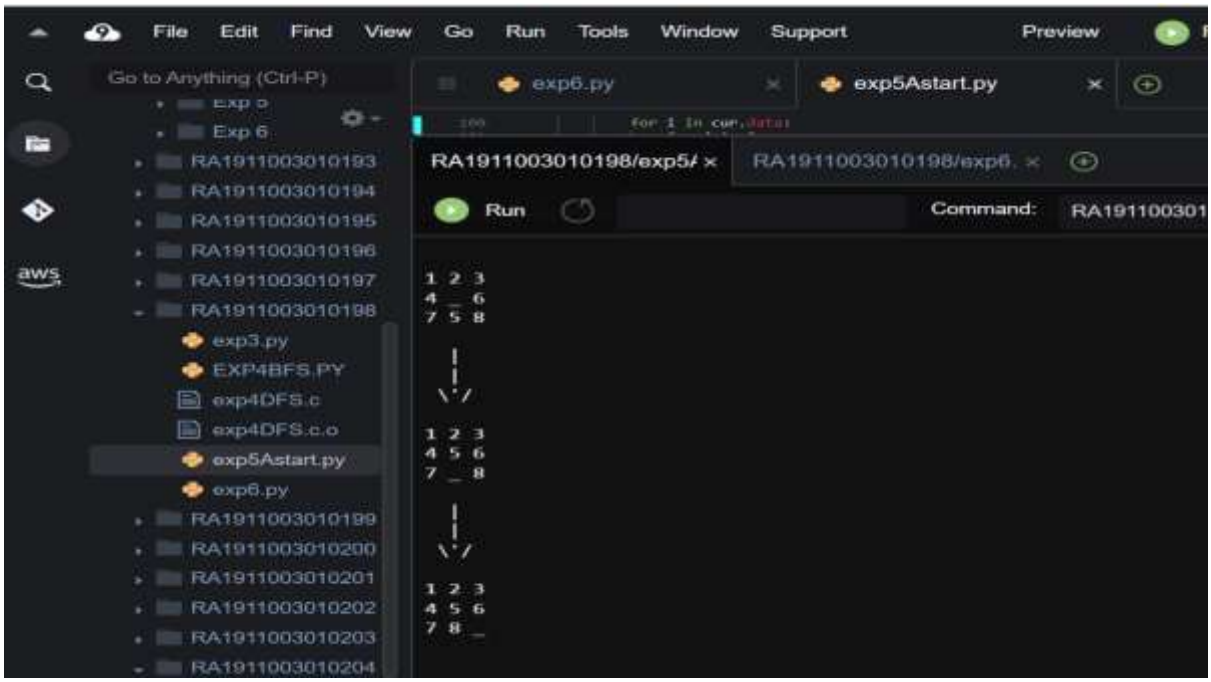
```
        """ Specifically used to find the position of the blank space """
        for i in range(0,len(self.data)):
            for j in range(0,len(self.data)):
                if puz[i][j] == x:
                    return i,j


class Puzzle:
    def __init__(self,size):
        """ Initialize the puzzle size by the specified size,open and closed lists to empty """
        self.n = size
        self.open = []
        self.closed = []
    def accept(self):
        """ Accepts the puzzle from the user """
        puz = []
        for i in range(0,self.n):
            temp = input().split(" ")
            puz.append(temp)
        return puz
    def f(self,start,goal):
        """ Heuristic Function to calculate hueristic value f(x) = h(x) + g(x) """
        return self.h(start.data,goal)+start.level


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

```
                temp += 1
        return temp
    def process(self):
        """ Accept Start and Goal Puzzle state"""
        print("Enter the start state matrix \n")
        start = self.accept()
        print("Enter the goal state matrix \n")
        goal = self.accept()
        start = Node(start,0,0)
        start.fval = self.f(start,goal)
        """ Put the start node in the open list"""
        self.open.append(start)
        print("\n\n")
        while True:
            cur = self.open[0]
            print("")
            print("  | ")
            print("  | ")
            print(" \\\'/ \n")
            for i in cur.data:
                for j in i:
                    print(j,end=" ")
                print("")
            """ If the difference between current and goal node is 0 we have reached the goal node"""
            if(self.h(cur.data,goal) == 0):
                break
            for i in cur.generate_child():
                i.fval = self.f(i,goal)
                self.open.append(i)
```

```
        self.closed.append(cur)
        del self.open[0]
        """ sort the opne list based on f value """
        self.open.sort(key = lambda x:x.fval,reverse=False)
puz = Puzzle(3)
puz.process()
```

Output:





Result: 8-Puzzle using A* Algorithm is successfully implemented.

Khushi Arora
RA1911003010198

classmate
Date
Page

Experiment - 5

8 Puzzle problem using A* Algo

» initial state

1	2	3
_	4	6
7	5	8

» goal state

1	2	3
4	5	6
7	8	_

1	2	3
_	4	6
7	5	8

swap the empty space with right

1	2	3
4	_	6
7	5	8

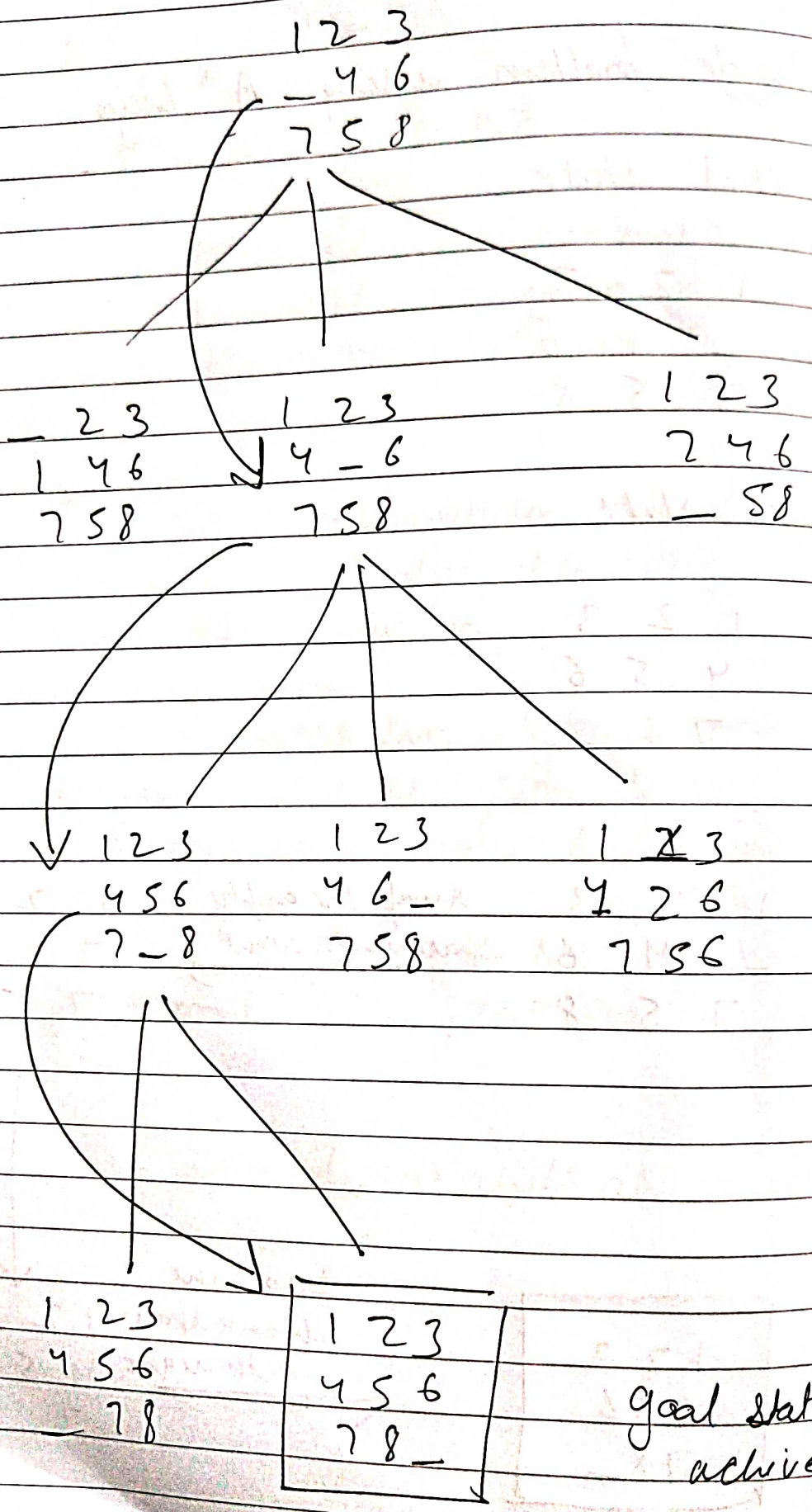
Swap the blank space with bottom

1	2	3
4	5	6
7	_	8

Swap the blank space with right

1	2	3
4	5	6
7	8	_

goal state achived.


1 2 3
_ 4 6
7 5 8
_ 2 3
1 4 6
7 5 8
1 2 3
4 _ 6
7 5 8
1 2 3
2 4 6
_ 5 8
1 2 3
4 5 6
7 _ 8
1 2 3
4 6 _
7 5 8
1 X 3
4 2 6
7 5 6
1 2 3
4 5 6
_ 7 8
1 2 3
4 5 6
7 8 _
goal state
achived

ARTIFICIAL INTELLIGENCE LAB

EXPERIMENT NO.:-06

Khushi Arora

RA1911003010198

Aim: - To implement tic tac toe problem using minimax algorithm.

Algorithm:-

1. We will write a code for Tic-Tac-Toe problem using minimax and evaluation function that plays a perfect game. This AI will consider all possible scenarios and makes the most optimal move.
2. We shall be introducing a new function called findBestMove(). This function evaluates all the available moves using minimax() and then returns the best move the maximizer can make.
3. To check whether or not the current move is better than the best move we take the help of minimax() function which will consider all the possible ways the game can go and returns the best value for that move, assuming the opponent also plays optimally.
4. The code for the maximizer and minimizer in the minimax() function is similar to findBestMove(), the only difference is, instead of returning a move, it will return a value.
5. To check whether the game is over and to make sure there are no moves left we use isMovesLeft() function. It is a simple straightforward function which checks whether a move is available or not and returns true or false respectively.
6. One final step is to make our AI a little bit smarter. Even though the following AI plays perfectly, it might choose to make a move which will result in a slower victory or a faster loss.
7. We add the depth value as the minimizer always tries to get, as negative a value as possible. We can subtract the depth either inside the evaluation function or outside it. I have chosen to do it outside the function

Code:-

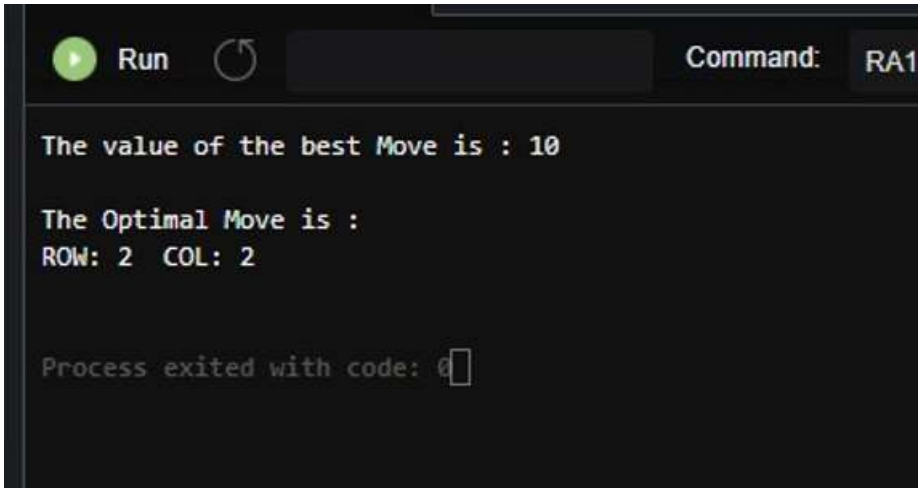
```
player, opponent = 'x', 'o'

def isMovesLeft(board) :

    for i in range(3) :

        for j in range(3) :
```

```
            if (board[i][j] == '_') :
                return True
    return False
def evaluate(b) :
    for row in range(3) :
        if (b[row][0] == b[row][1] and b[row][1] == b[row][2]) :
            if (b[row][0] == player) :
                return 10
            elif (b[row][0] == opponent) :
                return -10
    for col in range(3) :
        if (b[0][col] == b[1][col] and b[1][col] == b[2][col]) :
            if (b[0][col] == player) :
                return 10
            elif (b[0][col] == opponent) :
                return -10
    if (b[0][0] == b[1][1] and b[1][1] == b[2][2]) :
        if (b[0][0] == player) :
            return 10
        elif (b[0][0] == opponent) :
            return -10
    if (b[0][2] == b[1][1] and b[1][1] == b[2][0]) :
        if (b[0][2] == player) :
            return 10
        elif (b[0][2] == opponent) :
            return -10
    return 0
```

```
def minimax(board, depth, isMax) :
    score = evaluate(board)
    if (score == 10) :
        return score
    if (score == -10) :
        return score
    if (isMovesLeft(board) == False) :
        return 0
    if (isMax) :
        best = -1000
        for i in range(3) :
            for j in range(3) :
                if (board[i][j]=='_') :
                    board[i][j] = player
                    best = max( best, minimax(board, depth + 1, not isMax) )
                    board[i][j] = '_'
        return best
    else :
        best = 1000
        for i in range(3) :
            for j in range(3) :
                if (board[i][j] == '_') :
                    board[i][j] = opponent
                    best = min(best, minimax(board, depth + 1, not isMax))
                    board[i][j] = '_'
        return best
def findBestMove(board) :
```

```
    bestVal = -1000
    bestMove = (-1, -1)
    for i in range(3) :
        for j in range(3) :
            if (board[i][j] == '_') :
                board[i][j] = player
                moveVal = minimax(board, 0, False)
                board[i][j] = '_'
                if (moveVal>bestVal) :
                    bestMove = (i, j)
                    bestVal = moveVal
    print("The value of the best Move is :", bestVal)
    print()
    return bestMove
# Driver code
board = [
    [ 'x', 'o', 'x' ],
    [ 'o', 'o', 'x' ],
    [ '_', '_', '_' ]
]
bestMove = findBestMove(board)
print("The Optimal Move is :")
print("ROW:", bestMove[0], " COL:", bestMove[1])
```

OUTPUT:-



Result: - The Tic-Tac-Toe problem was successfully implemented using minimax algorithm

The 3 possible scenarios in the above example are :

Left Move : If X plays [2,0]. Then O will play [2,1] and win the game. The value of this move is -10
Middle Move : If X plays [2,1]. Then O will play [2,2] which draws the game. The value of this move is 0
Right Move : If X plays [2,2]. Then he will win the game. The value of this move is +10;