CSE401
Artificial Intelligence

Practical File

Submitted to

AMITY UNIVERSITY, UTTAR PRADESH

In partial fulfilment of the requirements for the award of the degree of

Bachelor of Technology

In

Computer Science & Engineering

Submitted By:

Sumit Kumar

A2305220005

6CSE-2X

Submitted to:

Dr. Nidhi Mishra

DEPARTMENT OF AMITY SCHOOL OF ENGINEERING

AMITY UNIVERSITY UTTAR PRADESH

TABLE OF CONTENTS

Introduction to Python

Python is a widely used general-purpose, high level programming language. It was created by Guido van Rossum in 1991 and further developed by the Python Software Foundation. It was designed with an emphasis on code readability, and its syntax allows programmers to express their concepts in fewer lines of code.

Python is a programming language that lets you work quickly and integrate systems more efficiently.

It is used for:

- web development (server-side),
- software development,
- mathematics,
- system scripting.

Creating a table

```
import pandas as pd
data = {'product_name': ['laptop', 'printer', 'tablet', 'desk', 'chair'],
          'price': [1200, 150, 300, 450, 200]
        }
df = pd.DataFrame(data)
print(df)
```

```
================ RESTART: C:\Users\sumit\Desktop\Sem 5\AI\1.py ================
  product_name  price
0       laptop   1200
1      printer    150
2       tablet    300
3         desk    450
4        chair    200
```

Adding new row

```
df.loc[5] = ['Mouse', 500]
print("\n", df)
```

```
================ RESTART: C:\Users\sumit\Desktop\Sem 5\AI\1.py ================

   product_name  price
0        laptop   1200
1       printer    150
2        tablet    300
3          desk    450
4         chair    200
5         Mouse    500
```

Adding new column

df['Count']= [30, 5, 10, 15, 20, 25]
df['Total price']=df['Count'] * df['price']
print("\n", df)

```
================ RESTART: C:\Users\sumit\Desktop\Sem 5\AI\1.py ================

  product_name  price  Count  Total price
0       laptop   1200     30        36000
1      printer    150      5          750
2       tablet    300     10         3000
3         desk    450     15         6750
4        chair    200     20         4000
5        Mouse    500     25        12500
```

Renaming column

df.rename(columns = {'product_name':'Product Name', 'price':'Price'}, inplace=True)
print("\n", df)

```
================ RESTART: C:\Users\sumit\Desktop\Sem 5\AI\1.py ================

  Product Name  Price  Count  Total price
0       laptop   1200     30        36000
1      printer    150      5          750
2       tablet    300     10         3000
3         desk    450     15         6750
4        chair    200     20         4000
5        Mouse    500     25        12500
```

Updating a full row

df.loc[3] = ['Desktop', 1500, 12, 18000]

```
================ RESTART: C:\Users\sumit\Desktop\Sem 5\AI\1.py ================

  Product Name  Price  Count  Total price
0       laptop   1200     30        36000
1      printer    150      5          750
2       tablet    300     10         3000
3      Desktop   1500     12        18000
4        chair    200     20         4000
5        Mouse    500     25        12500
```

Updating a value in a row

df.loc[3, ['Price']] = [2000]
print("\n", df)

```
================ RESTART: C:\Users\sumit\Desktop\Sem 5\AI\1.py ================

  Product Name  Price  Count  Total price
0       laptop   1200     30        36000
1      printer    150      5          750
2       tablet    300     10         3000
3      Desktop   2000     12        18000
4        chair    200     20         4000
5        Mouse    500     25        12500
```

Deleting a column

del df['Count']

df.drop('Price', inplace=True, axis=1)

print("\n", df)

```
================= RESTART: C:\Users\sumit\Desktop\Sem 5\AI\1.py ================

  Product Name  Total price
0       laptop        36000
1      printer          750
2       tablet         3000
3      Desktop        18000
4        chair         4000
5        Mouse        12500
```

Deleting a row

df.drop([0], inplace=True)

print("\n", df)

```
================= RESTART: C:\Users\sumit\Desktop\Sem 5\AI\1.py ================

  Product Name  Total price
1      printer          750
2       tablet         3000
3      Desktop        18000
4        chair         4000
5        Mouse        12500
```

Indexing data frame

index=pd.Index(['p1', 'p2', 'p3', 'p4', 'p5'])

df.set_index(index, inplace=True)

print("\n", df)

```
=============== RESTART: C:\Users\sumit\Desktop\Sem 5\AI\py\1.py ===============

   Product Name  Total price
p1      printer          750
p2       tablet         3000
p3      Desktop        18000
p4        chair         4000
p5        Mouse        12500
```

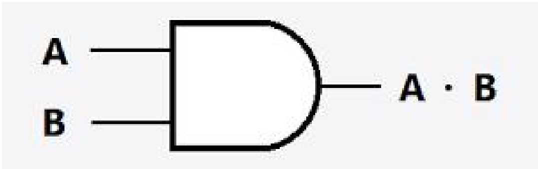
Criteria	Total Marks	Marks Obtained	Comments
Concept(A)	2		
Implementation(B)	2		
Performance	2		
Total	6 (to be scaled down to 1)		

Lab 1

Aim- Design an And, OR and XOR gate and its truth table using python.

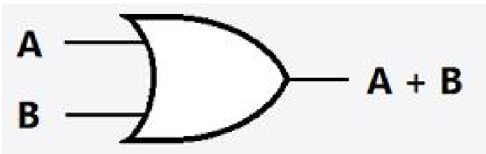
Theory

1. And Gate



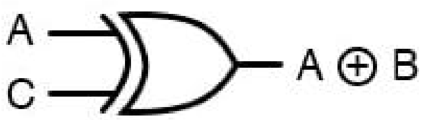
A	B	Output
0	0	0
0	1	0
1	0	0
1	1	1

2. OR Gate



Input		Output
A	**B**	**Y**
0	0	0
0	1	1
1	0	1
1	1	1

3. XOR Gate



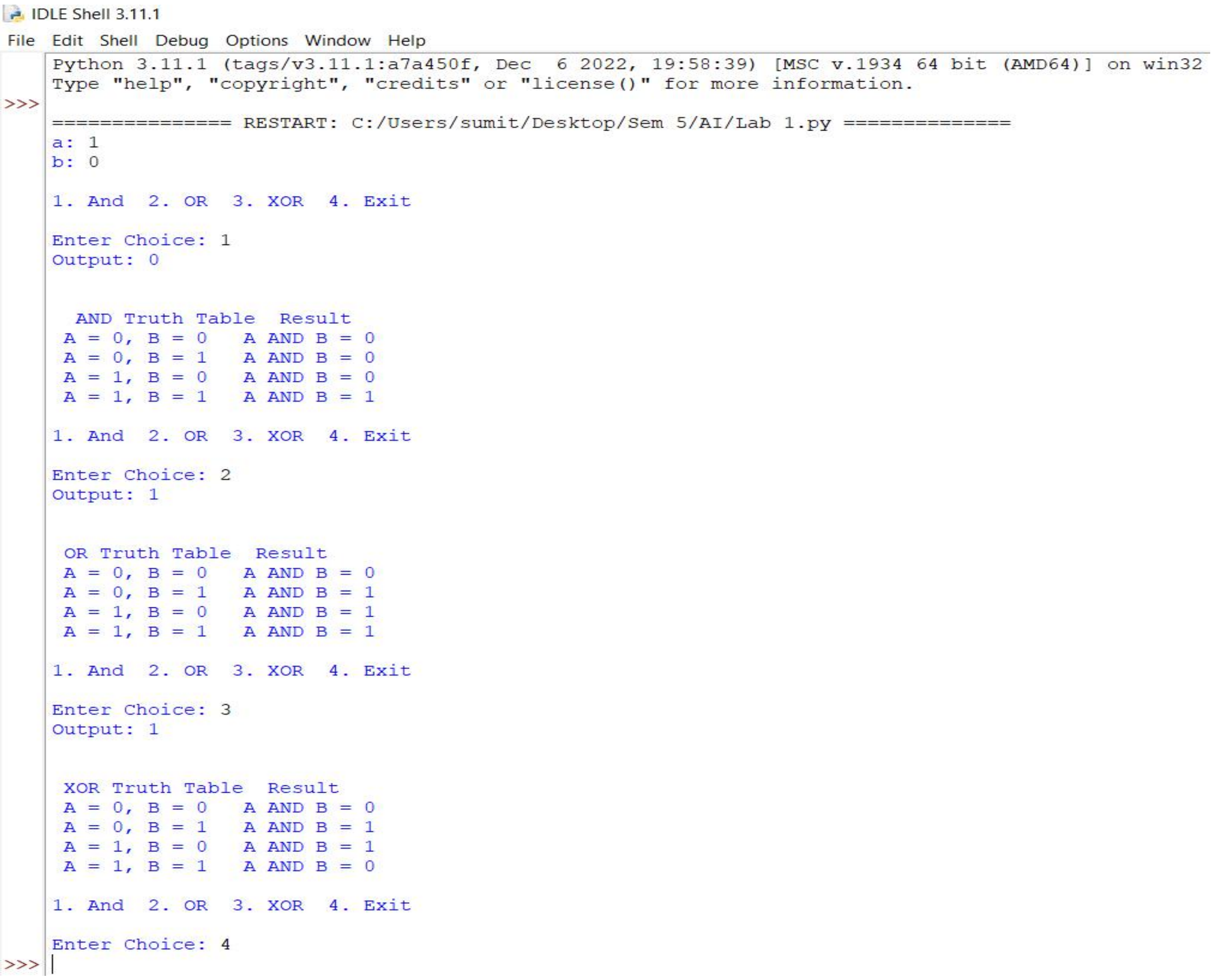
A	B	XOR
0	0	0
0	1	1
1	0	1
1	1	0

Code

```
def AND(a, b):
    if a == 1 and b == 1:
        return 1
    else:
        return 0

def OR(a, b):
    if a == 1 or b ==1:
        return 1
    else:
        return 0

def XOR(a, b):
    if a!=b:
        return 1
    else:
        return 0


# Driver code
if __name__=='__main__':

   a=int(input("a: "))
   b=int(input("b: "))

   x=1

   while x!=4:

    print("\n1. And  2. OR  3. XOR  4. Exit\n")
    ch=int(input("Enter Choice: "))

    if ch==1 :
        print("Output:",  AND(a,b))
        print("\n AND Truth Table  Result ")
        print(" A = 0, B = 0   A AND B =", AND(0,0))
        print(" A = 0, B = 1   A AND B =", AND(0,1))
        print(" A = 1, B = 0   A AND B =", AND(1,0))
        print(" A = 1, B = 1   A AND B =", AND(1,1))

    elif(ch==2):
        print("Output:",  OR(a,b))
        print("\n OR Truth Table  Result ")
        print(" A = 0, B = 0   A AND B =", OR(0,0))
        print(" A = 0, B = 1   A AND B =", OR(0,1))
        print(" A = 1, B = 0   A AND B =", OR(1,0))
        print(" A = 1, B = 1   A AND B =", OR(1,1))
```

```
elif(ch==3):
    print("Output:",  XOR(a,b))
    print("\n XOR Truth Table  Result ")
    print(" A = 0, B = 0   A AND B =", XOR(0,0))
    print(" A = 0, B = 1   A AND B =", XOR(0,1))
    print(" A = 1, B = 0   A AND B =", XOR(1,0))
    print(" A = 1, B = 1   A AND B =", XOR(1,1))

else:
    quit()
```

Output

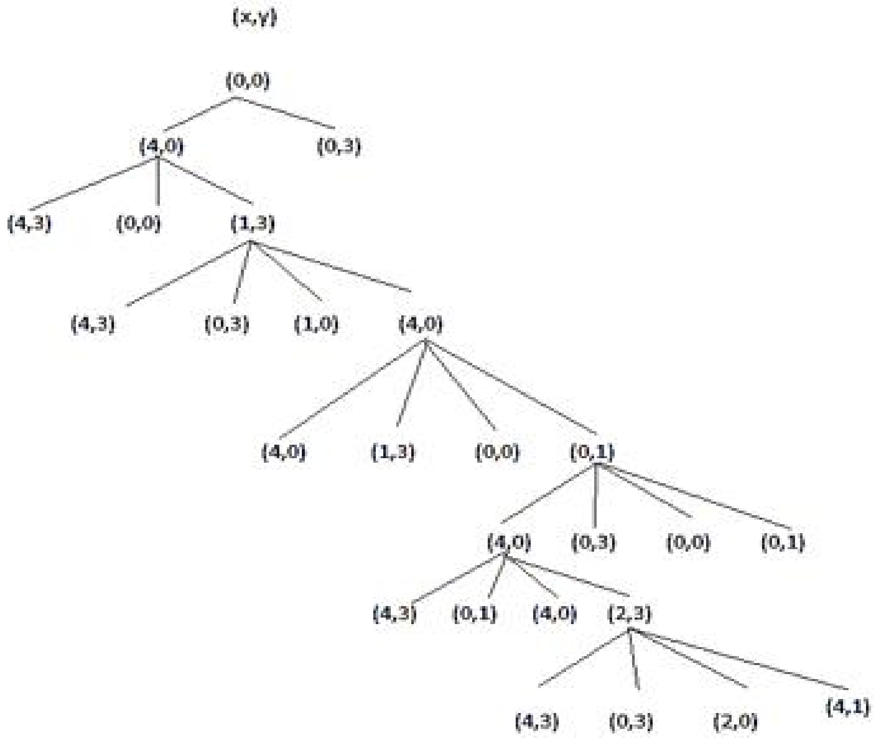


```
IDLE Shell 3.11.1
File  Edit  Shell  Debug  Options  Window  Help
Python 3.11.1 (tags/v3.11.1:a7a450f, Dec  6 2022, 19:58:39) [MSC v.1934 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
=============== RESTART: C:/Users/sumit/Desktop/Sem 5/AI/Lab 1.py ==============
a: 1
b: 0

1. And  2. OR  3. XOR  4. Exit

Enter Choice: 1
Output: 0


  AND Truth Table  Result
 A = 0, B = 0   A AND B = 0
 A = 0, B = 1   A AND B = 0
 A = 1, B = 0   A AND B = 0
 A = 1, B = 1   A AND B = 1

1. And  2. OR  3. XOR  4. Exit

Enter Choice: 2
Output: 1


 OR Truth Table  Result
 A = 0, B = 0   A AND B = 0
 A = 0, B = 1   A AND B = 1
 A = 1, B = 0   A AND B = 1
 A = 1, B = 1   A AND B = 1

1. And  2. OR  3. XOR  4. Exit

Enter Choice: 3
Output: 1


 XOR Truth Table  Result
 A = 0, B = 0   A AND B = 0
 A = 0, B = 1   A AND B = 1
 A = 1, B = 0   A AND B = 1
 A = 1, B = 1   A AND B = 0

1. And  2. OR  3. XOR  4. Exit

Enter Choice: 4
>>>
```

Criteria	Total Marks	Marks Obtained	Comments
Concept(A)	2		
Implementation(B)	2		
Performance	2		
Total	6 (to be scaled down to 1)		

Lab 2

Aim- Write a program to implement BFS for water jug problem using Python.



Code

```
from collections import deque

def BFS(a, b, target):

        # Map is used to store the states, every
        # state is hashed to binary value to
        # indicate either that state is visited
        # before or not

        m = {}
        isSolvable = False
        path = []

        q = deque()                              # Queue to maintain states

        q.append((0, 0))                         # Initializing with initial state.

        while (len(q) > 0):

                u = q.popleft()                  # Current state
                                                 # q.pop() #pop off used state
                                                 # If this state is already visited
```

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

# Doesn't met jug constraints
if ((u[0] > a or u[1] > b or
        u[0] < 0 or u[1] < 0)):
        continue

# Filling the vector for constructing the solution path
path.append([u[0], u[1]])

# Marking current state as visited
m[(u[0], u[1])] = 1

# If we reach solution state, put ans=1
if (u[0] == target or u[1] == target):
        isSolvable = True

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

                        # Fill final state
                        path.append([u[0], 0])
        else:
                if (u[0] != 0):

                        # Fill final state
                        path.append([0, u[1]])

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

# If we have not reached final state
# then, start developing intermediate
# states to reach solution state
q.append([u[0], b]) # Fill Jug2
q.append([a, u[1]]) # Fill Jug1
for ap in range(max(a, b) + 1):

        # Pour amount ap from Jug2 to Jug1
```

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

                # Check if this state is possible or not
                if (c == a or (d == 0 and d >= 0)):
                    q.append([c, d])

                # Pour amount ap from Jug 1 to Jug2
                c = u[0] - ap
                d = u[1] + ap

                # Check if this state is possible or not
                if ((c == 0 and c >= 0) or d == b):
                    q.append([c, d])

            q.append([a, 0])                    # Empty Jug2

            q.append([0, b])                    # Empty Jug1

        if (not isSolvable):                    # No, solution exists if ans=0
            print("No solution")

# Driver code
if __name__ == '__main__':

    Jug1, Jug2, target = 4, 3, 2
    print("Path from initial state to solution state:")
    print("\nJug 1   Jug 2")

    BFS(Jug1, Jug2, target)
```

Output

```
============ RESTART: C:/Users/sumit/Desktop/Sem 5/AI/py/Lab 2.py ============
Path from initial state to solution state:

Jug 1   Jug 2
0       0
0       3
4       0
4       3
3       0
1       3
3       3
4       2
0       2
```

Lab 3

Aim- Write a program to implement DFS using python.

Theory

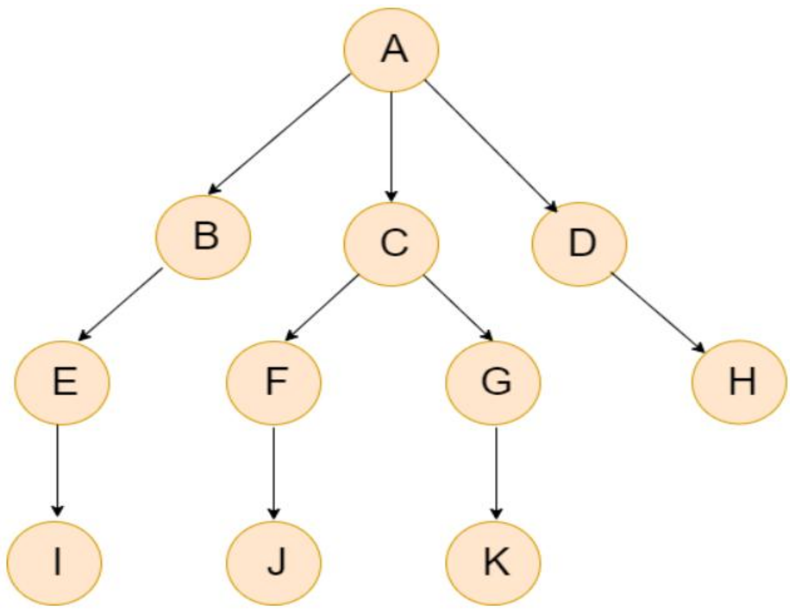
Depth-first search (DFS) is an algorithm for traversing or searching tree or graph data structures. The algorithm starts at the root node (selecting some arbitrary node as the root node in the case of a graph) and explores as far as possible along each branch before backtracking. Extra memory, usually a stack, is needed to keep track of the nodes discovered so far along a specified branch which helps in backtracking of the graph.

The DFS algorithm:

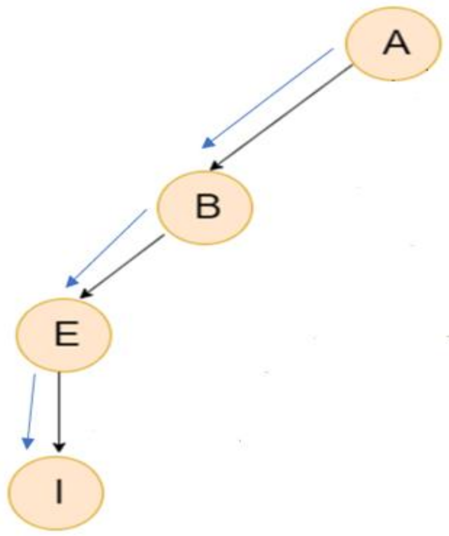
1. Start by putting any one of the graph's vertices on top of a stack.
2. Take the top item of the stack and add it to the visited list.
3. Create a list of that vertex's adjacent nodes. Add the ones which aren't in the visited list to the top of the stack.
4. Keep repeating steps 2 and 3 until the stack is empty.

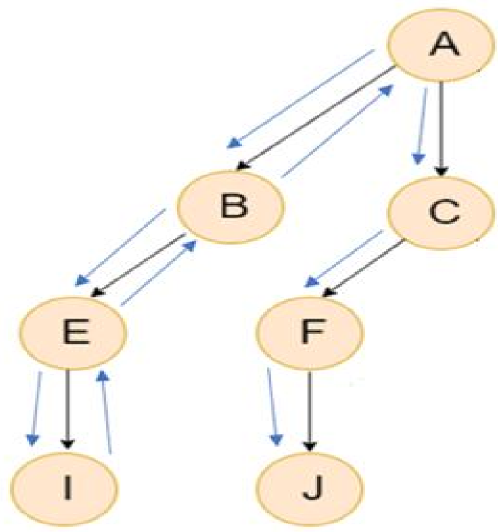
```
DFS(visited, u):
    visited[u]=True
    Print u
    for each v in adj[u]:
        if(visited[v]=False)
            DFS(visited,v)
```

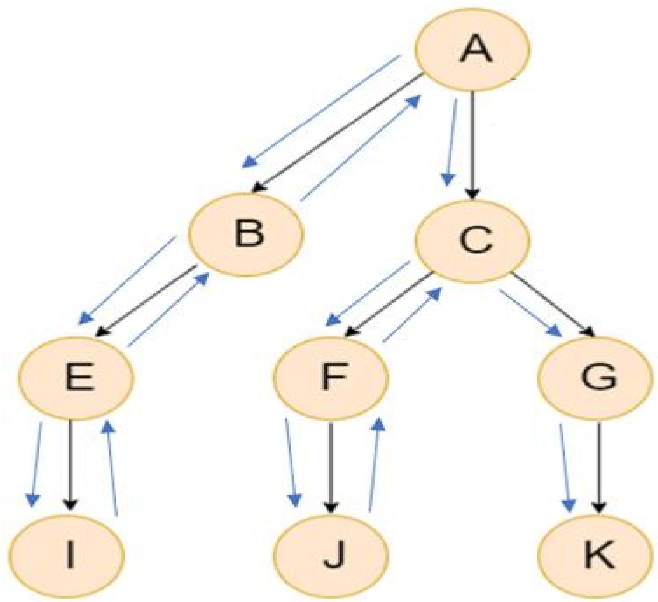
O (V + E)

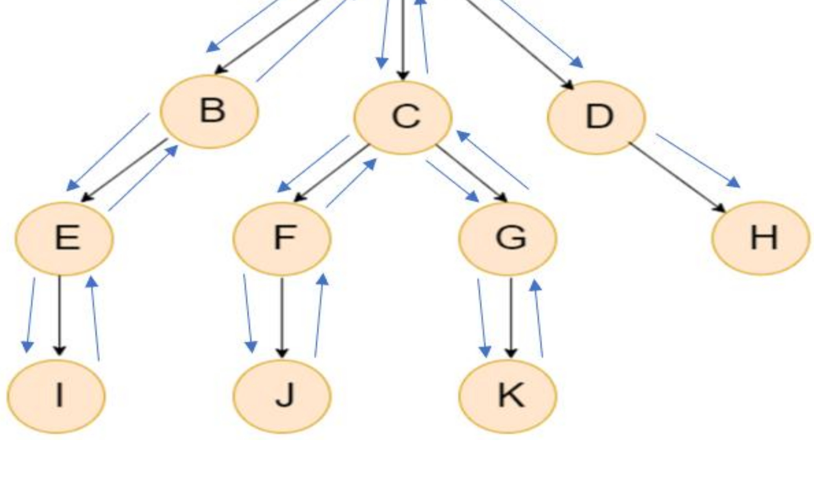


Graph used for the program.


A
B
E
I


A
B
C
E
F
I
J

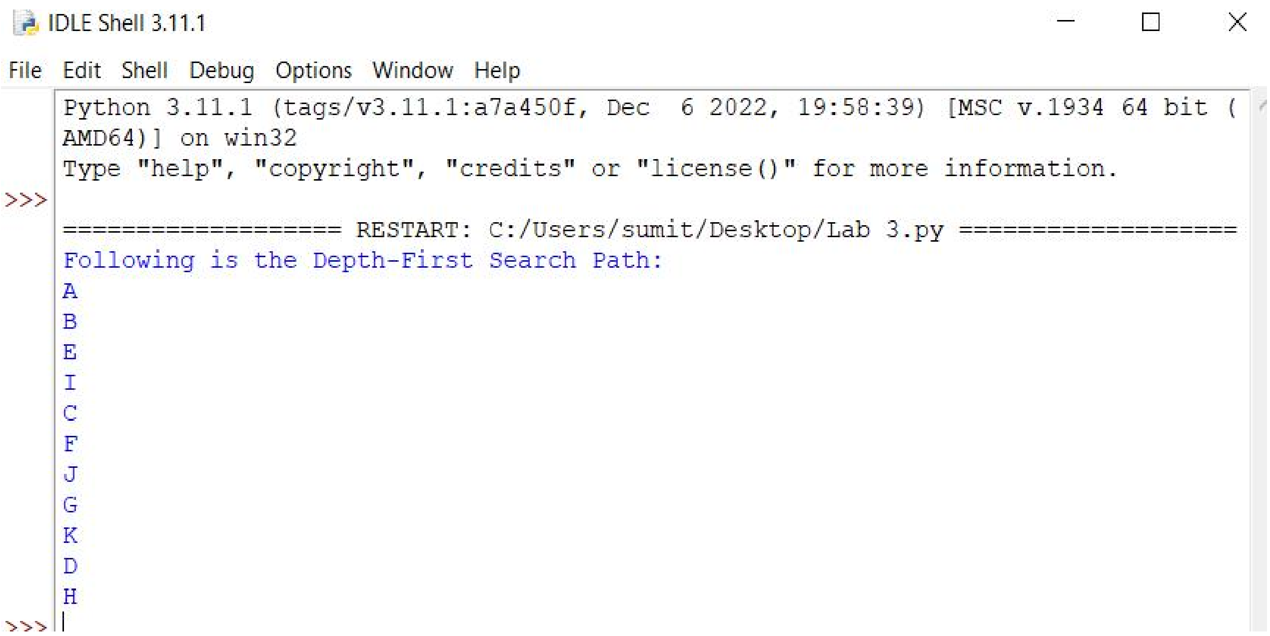

A
B
C
E
F
G
I
J
K


A
B
C
D
E
F
G
H
I
J
K

Code

```
graph = {
        'A' : ['B', 'C', 'D'],
        'B' : ['E'],
        'C' : ['F', 'G'],
        'D' : ['H'],
        'E' : ['I'],
        'F' : ['J'],
        'G' : ['K'],
        'H' : [],
        'I' : [],
        'J' : [],
        'K' : []
       }

visited = set()                          # Set to keep track of visited nodes of graph.

def dfs(visited, graph, node):           #function for dfs.
    if node not in visited:
        print (node)
        visited.add(node)
        for neighbour in graph[node]:
            dfs(visited, graph, neighbour)

# Driver Code

print("Following is the Depth-First Search Path:")
dfs(visited, graph, 'A')
```

Output



Lab 4

Aim- Write a program to solve an 8-puzzle problem using A* algorithm in python.

Theory

A* Search Algorithm is a simple and efficient search algorithm that can be used to find the optimal path between two nodes in a graph. It will be used for the shortest path finding. It is an extension of Dijkstra's shortest path algorithm (Dijkstra's Algorithm). The extension here is that, instead of using a priority queue to store all the elements, we use heaps (binary trees) to store them. The A* Search Algorithm also uses a heuristic function that provides additional information regarding how far away from the goal node we are. This function is used in conjunction with the f-heap data structure in order to make searching more efficient.

h(n) - It is the heuristic value of a node. In this program h(n) is the number of displaced tiles (not counting the spaces).

g(n) - It is the distance travelled from one node to another. In this program it is the depth of the node.

f(n) - It denotes the cost of travelling from one node to another.

$$\boxed{f(n) = g(n) + h(n)}$$

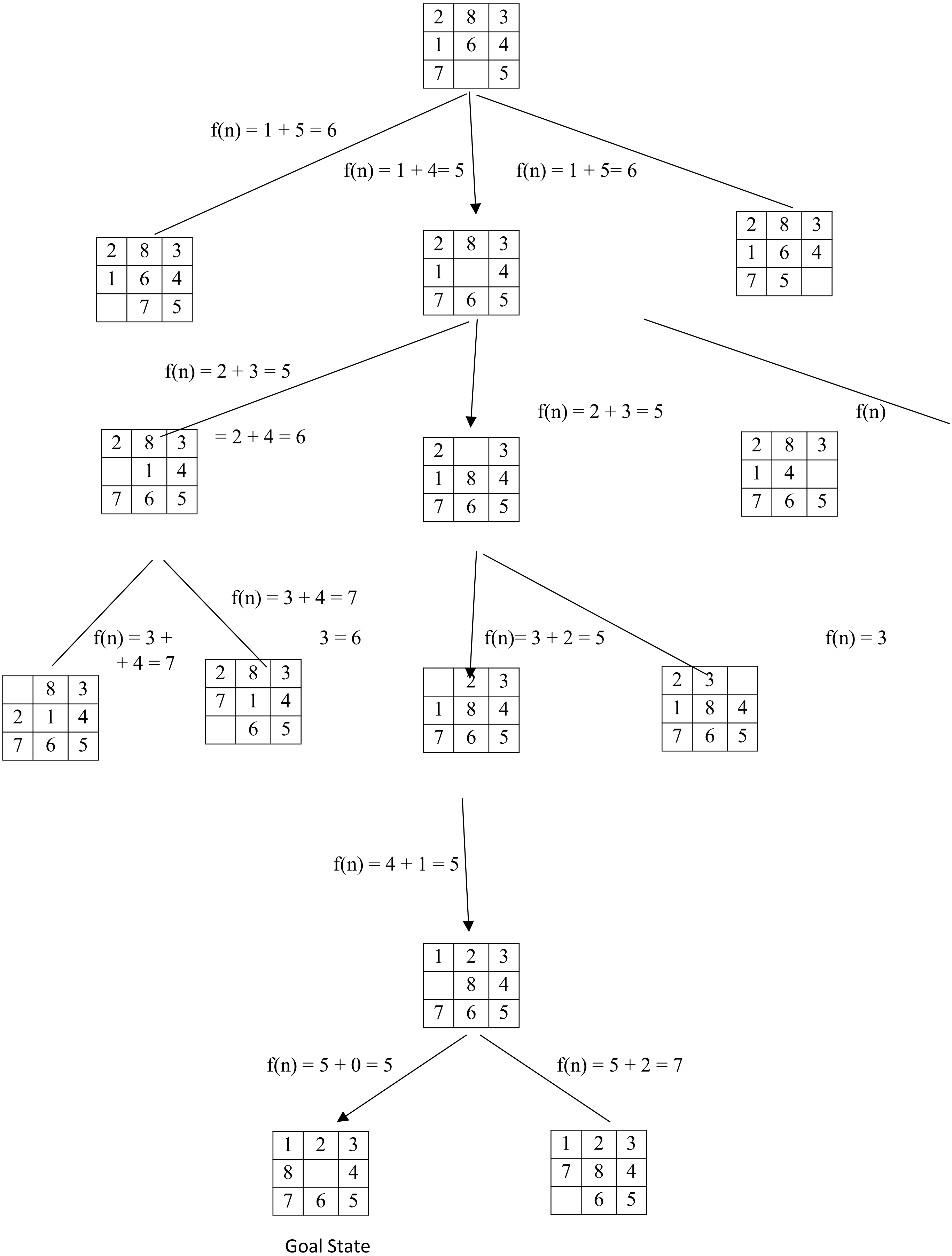
A* Algorithm	Greedy Algorithm
A* uses a heuristics to find the solution as quickly as possible.	Greedy algorithms don't use a heuristic value and simply make the locally optimal choice at each step.
A* uses both the cost to reach a node and a heuristic that estimates the cost to get from that node to the goal. This allows A* to prioritize exploring paths that are likely to lead to the goal, resulting in faster convergence.	Greedy only use the cost and choose the path with the lowest estimated cost to the goal, without considering the actual cost to reach that node. This can result in the algorithm getting stuck in suboptimal paths.
A* is more efficient than greedy algorithms as it uses both the cost to reach a node and a heuristic.	Greedy algorithms are less efficient than A* since they only consider cost of reaching a node.

2	8	3
1	6	4
7		5

Initial State

1	2	3
8		4
7	6	5

Final State



f(n) = 1 + 5 = 6
f(n) = 1 + 4= 5
f(n) = 1 + 5= 6
f(n) = 2 + 3 = 5
= 2 + 4 = 6
f(n) = 2 + 3 = 5
f(n)
f(n) = 3 + 4 = 7
f(n) = 3 +
+ 4 = 7
3 = 6
f(n)= 3 + 2 = 5
f(n) = 3
f(n) = 4 + 1 = 5
f(n) = 5 + 0 = 5
f(n) = 5 + 2 = 7
Goal State

Code

```
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("\nEnter 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")
        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

```
IDLE Shell 3.11.1
File  Edit  Shell  Debug  Options  Window  Help
Python 3.11.1 (tags/v3.11.1:a7a450f, Dec  6 2022, 19:58:39) [MSC v
Type "help", "copyright", "credits" or "license()" for more informa
>>>
============= RESTART: C:\Users\sumit\Desktop\Sem 6\AI\py\Lab 4.py
Enter the start state matrix

2 8 3
1 6 4
7 _ 5

Enter the goal state matrix

1 2 3
8 _ 4
7 6 5



  |
  |
 \'/

2 8 3
1 6 4
7 _ 5

  |
  |
 \'/

2 8 3
1 _ 4
7 6 5

  |
  |
 \'/

2 8 3
_ 1 4
7 6 5

  |
  |
 \'/

2 _ 3
1 8 4
7 6 5

  |
  |
 \'/

_ 2 3
1 8 4
7 6 5

  |
  |
 \'/

1 2 3
_ 8 4
7 6 5

  |
  |
 \'/

1 2 3
8 _ 4
7 6 5
```

<u>**Lab 5**</u>

<u>**Aim-**</u> To implement 8 Puzzle Single Player Game using Breadth First Search.

Introduction

An instance of the n-puzzle game consists of a board holding n^2 -1 distinct movable tiles, plus an empty space. The tiles are numbers from the set 1….. n^2 -1. For any such board the empty space may be legally swapped with any tile horizontally or vertically adjacent to it. In this assignment the blank space is going to be represented with the number 0. Given an initial state fo the board, the combinatorial search problem is to find a sequence of moves that transitions this state to the goal state that is the configuration with all tiles arranged in ascending order 0, 1, … n^2 -1.

The search space is the set of all possible states from reachable from the initial state. The blank space may be swapped with a component in one of the four directions {'Up', 'Down', 'Left', 'Right'}, one move at a time.

In this 8 puzzle problem a 3 into 3 board with 8 tiles (every tile has one number from 1 to 8) and one empty space. The objective is to place the number on tiles to match final configuration using the empty space. We can slide four adjacent (left, right, above and below) tiles into the empty space.

Breadth First Search (BFS)

We can perform a breadth first search on state space (Set of all configurations of a given problem i.e. all states that can be reached from the initial state) tree

Algorithm Review

The searches begin by visiting the root node of the search tree, given by the initial state. Among other book-keeping details, three major things happen in sequence in order to visit a node:

1. First, we remove a node from the frontier set.
2. Second, we check the state against the goal state to determine if a solution has been found.
3. Finally, if the result of the check is negative, we then expand the node. To expand a given node, we generate successor nodes adjacent to the current node, and add them to the frontier set. Note that if these successor nodes are already in the frontier, or have already been visited, then they should not be added to the frontier again.

Initial state:

1	2	5
3	4	
6	7	8

The nodes expanded by BFS (also the nodes that are in the fringe / frontier of the queue) are shown in the following figure:

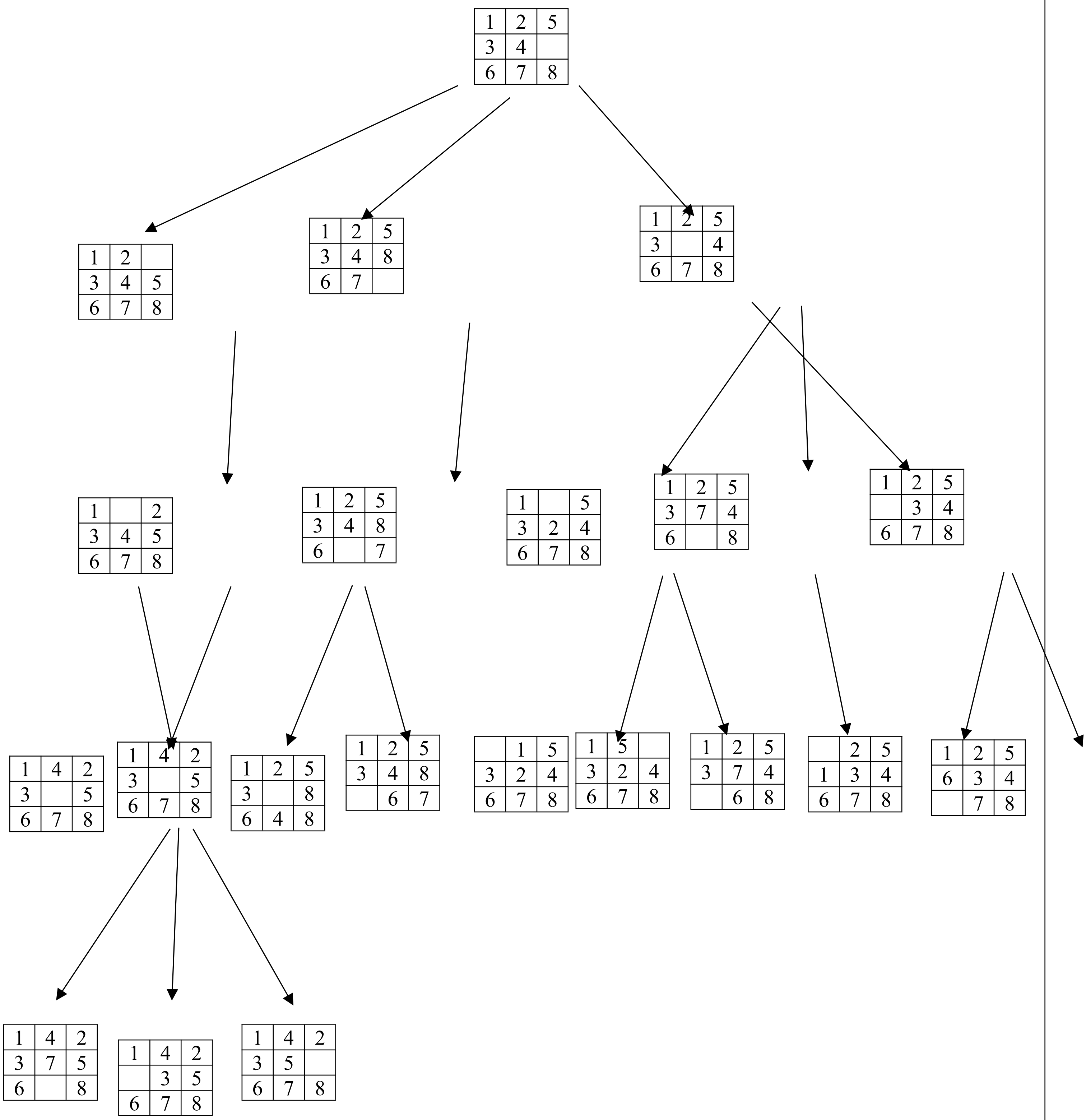


Figure 5.1 BFS State Space Tree

Code

```
#Import the necessary libraries
from time import time
from queue import Queue

#Creating a class Puzzle
class Puzzle:
   #Setting the goal state of 8-puzzle
   goal_state=[1,2,3,8,0,4,7,6,5]
   num_of_instances=0
   #default constructor to initialize the class members
   def __init__(self,state,parent,action):
      self.parent=parent
      self.state=state
      self.action=action   #action is used to generate current state from parent state

      #TODO: incrementing the number of instance by 1
      # num_of_instances = num_of_instances + 1
      Puzzle.num_of_instances+= 1
      print("Current state: ", Puzzle.num_of_instances)
      print(self.state)

   #function used to display a state of 8-puzzle
   def __str__(self):
      return str(self.state[0:3])+'\n'+str(self.state[3:6])+'\n'+str(self.state[6:9])

   #method to compare the current state with the goal state
   def goal_test(self):
      #TODO: include a condition to compare the current state with the goal state
      if self.state == Puzzle.goal_state:
         print("Goal state found and printed in reverse order: ")

print(str(Puzzle.goal_state[0:3])+'\n'+str(Puzzle.goal_state[3:6])+'\n'+str(Puzzle.goal_state[6:9]))
         return True
      else:
         return False

   #static method to find the legal action based on the current board position
   @staticmethod
   def find_legal_actions(i,j):
      legal_action = ['U', 'D', 'L', 'R']
      if i == 0:
         # if row is 0 in board then up is disable
         legal_action.remove('U')
```

```
        elif i == 2:
            #TODO: down is disable
            legal_action.remove('D')
        if j == 0:
            #TODO: Left is disable
            legal_action.remove('L')
        elif j == 2:
            #TODO: Right is disable
            legal_action.remove('R')
        return legal_action

    #method to generate the child of the current state of the board
    def generate_child(self):
        #TODO: create an empty list
        children=[]
        x = self.state.index(0)
        i = int(x / 3)
        j = int(x % 3)
        #TODO: call the method to find the legal actions based on i and j values
        legal_actions=Puzzle.find_legal_actions(i, j)

        #TODO:Iterate over all legal actions
        for action in legal_actions:
            new_state = self.state.copy()
            #if the legal action is UP
            if action == 'U':
                #Swapping between current index of 0 with its up element on the board
                new_state[x], new_state[x-3] = new_state[x-3], new_state[x]
            elif action == 'D':
                #TODO: Swapping between current index of 0 with its down element on the board
                new_state[x], new_state[x+3] = new_state[x+3], new_state[x]
            elif action == 'L':
                #TODO: Swapping between the current index of 0 with its left element on the board
                new_state[x], new_state[x-1] = new_state[x-1], new_state[x]
            elif action == 'R':
                #TODO: Swapping between the current index of 0 with its right element on the board
                new_state[x], new_state[x+1] = new_state[x+1], new_state[x]
            children.append(Puzzle(new_state,self,action))
        #TODO: return the children
        return children


        #TODO:Iterate over all legal actions
        for action in legal_actions:
            new_state = self.state.copy()
```

```
            #if the legal action is UP
            if action == 'U':
                #Swapping between current index of 0 with its up element on the board
                new_state[x], new_state[x-3] = new_state[x-3], new_state[x]
            elif action == 'D':
                #TODO: Swapping between current index of 0 with its down element on the board
                new_state[x], new_state[x+3] = new_state[x+3], new_state[x]
            elif action == 'L':
                #TODO: Swapping between the current index of 0 with its left element on the board
                new_state[x], new_state[x-1] = new_state[x-1], new_state[x]
            elif action == 'R':
                #TODO: Swapping between the current index of 0 with its right element on the board
                new_state[x], new_state[x+1] = new_state[x+1], new_state[x]
            children.append(Puzzle(new_state,self,action))
        #TODO: return the children
        return children

    #method to find the solution
    def find_solution(self):
        solution = []
        solution.append(self.action)
        path = self
        while path.parent != None:
            path = path.parent
            print("  |  ")
            print(path)
            solution.append(path.action)
        solution = solution[:-1]
        solution.reverse()
        return solution

#method for breadth first search
#TODO: pass the initial_state as parameter to the breadth_first_search method
def breadth_first_search(initial_state):
    start_node = Puzzle(initial_state, None, None)
    print("Initial state:")
    print(start_node)
    print("STATES OF THR BOARD")
    if start_node.goal_test():
        return start_node.find_solution()
    q = Queue()
    #TODO: put start_node into the Queue
    q.put(start_node)
    #TODO: create an empty list of explored nodes
```

```
    explored=[]
    #TODO: Iterate the queue until empty. Use the empty() method of Queue
    while not(q.empty):
        #TODO: get the current node of a queue. Use the get() method of Queue
        node=q.get
        #TODO: Append the state of node in the explored list as node.state
        explored.append(node.state)
        #TODO: call the generate_child method to generate the child nodes of current node
        children= node.generate_child(node)
        #TODO: Iterate over each child node in children
        for child in children:
            if child.state not in explored:
                if child.goal_test():
                    return child.find_solution()
                q.put(child)
    return

#Start executing the 8-puzzle with setting up the initial state
#Here we have considered 3 initial state intitalized using state variable
state=[ 1, 2, 5,
        3, 4, 0,
        6, 7, 8],

#Iterate over number of initial_state
for i in range(len(state)):
    #TODO: Initialize the num_of_instances to zero
    Puzzle.num_of_instances=0
    #Set t0 to current time
    t0=time()
    bfs=breadth_first_search(state[i])
    #Get the time t1 after executing the breadth_first_search method
    t1=time()-t0
    print('BFS:', bfs)
    print('space:',Puzzle.num_of_instances)
    print('time:',t1)
```

Output

```
Current state:  1
[1, 2, 5, 3, 4, 0, 6, 7, 8]
Initial state:
[1, 2, 5]
```

```
[3, 4, 0]
[6, 7, 8]

STATES OF THE BOARD
Current state:  2
[1, 2, 0, 3, 4, 5, 6, 7, 8]
Current state:  3
[1, 2, 5, 3, 4, 8, 6, 7, 8]
Current state:  4
[1, 2, 5, 3, 0, 4, 6, 7, 8]
Current state:  5
[1, 2, 5, 3, 4, 0, 6, 7, 8]
Current state:  6
[1, 0, 2, 3, 4, 5, 6, 7, 8]
Current state:  7
[1, 2, 5, 3, 4, 0, 6, 7, 8]
Current state:  8
[1, 2, 5, 3, 4, 8, 6, 0, 8]
Current state:  9
[1, 0, 5, 3, 2, 4, 6, 7, 8]
Current state:  10
[1, 2, 5, 3, 7, 4, 6, 7, 8]
Current state:  11
[1, 2, 5, 0, 3, 4, 6, 7, 8]
Current state:  12
[1, 2, 5, 3, 4, 0, 6, 7, 8]
Current state:  13
[1, 4, 2, 3, 0, 5, 6, 7, 8]
Current state:  14
[0, 1, 2, 3, 4, 5, 6, 7, 8]
Current state:  15
[1, 2, 0, 3, 4, 5, 6, 7, 8]
Current state:  16
[1, 2, 5, 3, 0, 8, 6, 4, 7]
Current state:  17
[1, 2, 5, 3, 4, 8, 0, 6, 7]
Current state:  18
[1, 2, 5, 3, 4, 8, 6, 7, 0]
Current state:  19
[1, 2, 5, 3, 0, 4, 6, 7, 8]
Current state:  20
[0, 1, 5, 3, 2, 4, 6, 7, 8]
Current state:  21
[1, 5, 0, 3, 2, 4, 6, 7, 8]
```

```
Current state:  22
[1, 2, 5, 3, 0, 4, 6, 7, 8]
Current state:  23
[1, 2, 5, 3, 7, 4, 0, 6, 8]
Current state:  24
[1, 2, 5, 3, 7, 4, 6, 8, 0]
Current state:  25
[0, 2, 5, 1, 3, 4, 6, 7, 8]
Current state:  26
[1, 2, 5, 6, 3, 4, 0, 7, 8]
Current state:  27
[1, 2, 5, 3, 0, 4, 6, 7, 8]
Current state:  28
[1, 0, 2, 3, 4, 5, 6, 7, 8]
Current state:  29
[1, 4, 2, 3, 7, 5, 6, 0, 8]
Current state:  30
[1, 4, 2, 0, 3, 5, 6, 7, 8]
Current state:  31
[1, 4, 2, 3, 5, 0, 6, 7, 8]

BFS:[ 'U', 'L', 'D', 'R']
Space: 31
Time: 0.12287497520446777
```

Criteria	Total Marks	Marks Obtained	Comments
Concept(A)	2		
Implementation(B)	2		
Performance	2		
Total	6 (to be scaled down to 1)		

Lab 6

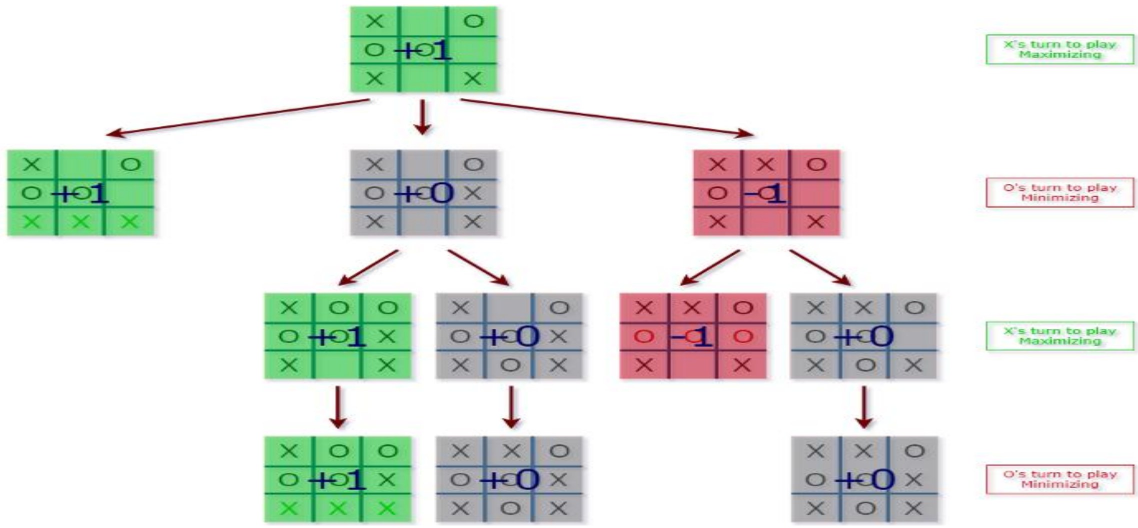
Aim- To implement Tic-Tac-Toe game using MiniMax Algorithm.

Theory

Minimax is an artificial intelligence applied in two player games, such as tic-tac-toe, checkers, chess and go. This games are known as zero-sum games, because in a mathematical representation: one player wins (+1) and other player loses (-1) or both of anyone not to win (0).

Minimax is a type of adversarial search algorithm for generating and exploring game trees. It is mostly used to solve zero-sum games where one side's gain is equivalent to other side's loss, so adding all gains and subtracting all losses end up being zero.

Adversarial search differs from conventional searching algorithms by adding opponents into the mix. Minimax algorithm keeps playing the turns of both player and the opponent optimally to figure out the best possible move.



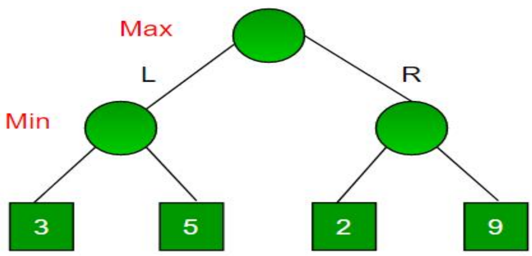
The algorithm search, recursively, the best move that leads the Max player to win or not lose (draw). It considers the current state of the game and the available moves at that state, then for each valid move it plays (alternating min and max) until it finds a terminal state (win, draw or lose).

Minimax is a kind of backtracking algorithm that is used in decision making and game theory to find the optimal move for a player, assuming that your opponent also plays optimally. It is widely used in two player turn-based games such as Tic-Tac-Toe, Backgammon, Mancala, Chess, etc.

In Minimax the two players are called maximizer and minimizer. The maximizer tries to get the highest score possible while the minimizer tries to do the opposite and get the lowest score possible.

Every board state has a value associated with it. In a given state if the maximizer has upper hand, then, the score of the board will tend to be some positive value. If the minimizer has the upper hand in that board state, then it will tend to be some negative value. The values of the board are calculated by some heuristics which are unique for every type of game.

Consider a game which has 4 final states and paths to reach final state are from root to 4 leaves of a perfect binary tree as shown below. Assume that maximizing player get the first chance to move and is at the root and opponent at next level.



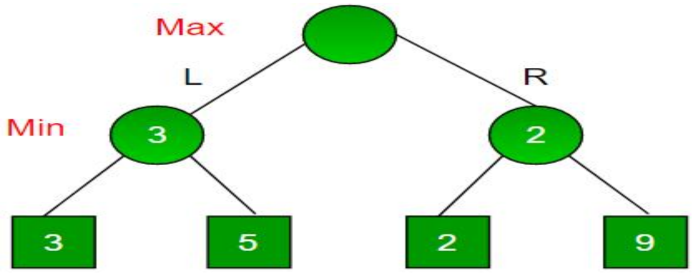
Since this is a backtracking-based algorithm, it tries all possible moves, then backtracks and makes a decision.

Maximizer goes LEFT: It is now the minimizers turn. The minimizer now has a choice between 3 and 5. Being the minimizer it will definitely choose the least among both, that is 3

Maximizer goes RIGHT: It is now the minimizers turn. The minimizer now has a choice between 2 and 9. He will choose 2 as it is the least among the two values.

Being the maximizer you would choose the larger value that is 3. Hence the optimal move for the maximizer is to go LEFT and the optimal value is 3.

Now the game tree looks like below :



The above tree shows two possible scores when maximizer makes left and right moves.

Code

```
import numpy as np
from math import inf as infinity

#Set the Empty Board
game_state = [[' ',' ',' '],
              [' ',' ',' '],
              [' ',' ',' ']]
#Create the Two Players as 'X'/'O'
players = ['X','O']

#Method for checking the correct move on Tic-Tac-Toe
def play_move(state, player, block_num):
   if state[int((block_num-1)/3)][(block_num-1)%3] == ' ':
#TODO: Assign the player move on the current position of Tic-Tac-Toe if condition is True
      state[int((block_num-1)/3)][(block_num-1)%3] = player
   else:
      block_num = int(input("Block is not empty, ya blockhead! Choose again: "))
      play_move(state, player, block_num)
      #TODO: Recursively call the play_move

#Method to copy the current game state to new_state of Tic-Tac-Toe
def copy_game_state(state):
   new_state = [[' ',' ',' '],[' ',' ',' '],[' ',' ',' ']]
   for i in range(3):
      for j in range(3):
```

```
            #TODO: Copy the Tic-Tac-Toe state to new_state
            new_state[i][j] = state[i][j]
    #TODO: Return the new_state
    return new_state

#Method to check the current state of the Tic-Tac-Toe
def check_current_state(game_state):
    #TODO: Set the draw_flag to 0
    draw_flag = 0
    for i in range(3):
        for j in range(3):
            if game_state[i][j] == ' ':
                draw_flag = 1
    if draw_flag == 0:
        return None, "Draw"
    # Check horizontals in first row
    if (game_state[0][0]==game_state[0][1] and game_state[0][1]==game_state[0][2] and game_state[0][0] != ' '):
        return game_state[0][0], "Done"
    #TODO: Check horizontals in second row
    if (game_state[1][0]==game_state[1][1] and game_state[1][1]==game_state[1][2] and game_state[1][0] != ' '):
        return game_state[1][0], "Done"
    #TODO: Check horizontals in third row
    if (game_state[2][0]==game_state[2][1] and game_state[2][1]==game_state[2][2] and game_state[2][0] != ' '):
        return game_state[2][0], "Done"

    # Check verticals in first column
    if (game_state[0][0]==game_state[1][0] and game_state[1][0]==game_state[2][0] and game_state[0][0] != ' '):
        return game_state[0][0], "Done"
    # Check verticals in second column
    if (game_state[0][1]==game_state[1][1] and game_state[1][1]==game_state[2][1] and game_state[0][1] != ' '):
        return game_state[0][1], "Done"
    # Check verticals in third column
    if (game_state[0][2]==game_state[1][2] and game_state[1][2]==game_state[2][2] and game_state[0][2] != ' '):
        return game_state[0][2], "Done"

   # Check left diagonal
    if (game_state[0][0]==game_state[1][1] and game_state[1][1]==game_state[2][2] and game_state[0][0] != ' '):
        return game_state[1][1], "Done"
    # Check right diagonal
    if (game_state[2][0]==game_state[1][1] and game_state[1][1]==game_state[0][2] and game_state[2][0] != ' '):
        return game_state[1][1], "Done"

    return None, "Not Done"

#Method to print the Tic-Tac-Toe Board
```

```
def print_board(game_state):
    print('----------------')
    print('| ' + str(game_state[0][0]) + ' || ' + str(game_state[0][1]) + ' || ' + str(game_state[0][2]) + ' |')
    print('----------------')
    print('| ' + str(game_state[1][0]) + ' || ' + str(game_state[1][1]) + ' || ' + str(game_state[1][2]) + ' |')
    print('----------------')
    print('| ' + str(game_state[2][0]) + ' || ' + str(game_state[2][1]) + ' || ' + str(game_state[2][2]) + ' |')
    print('----------------')

#Method for implement the Minimax Algorithm
def getBestMove(state, player):
    #TODO: call the check_current_state method using state parameter
    winner_loser , done = check_current_state(state)
    #TODO:Check condition for winner if winner_loser is 'O' then Computer won
    #else if winner_loser is 'X' then You won else game is draw
    if done == "Done" and winner_loser == 'O': # If AI won
        return (1,0)
    elif done == "Done" and winner_loser == 'X': # If Human won
        return (-1,0)
    elif done == "Draw":    # Draw condition
        return (0,0)

    #TODO: set moves to empty list
    moves = []
    #TODO: set empty_cells to empty list
    empty_cells = []

    #Append the block_num to the empty_cells list
    for i in range(3):
        for j in range(3):
            if state[i][j] == ' ':
                empty_cells.append(i*3 + (j+1))

    #TODO:Iterate over all the empty_cells
    for empty_cell in empty_cells:
        #TODO: create the empty dictionary
        move = {}

        #TODO: Assign the empty_cell to move['index']
        move['index'] = empty_cell

        #Call the copy_game_state method
        new_state = copy_game_state(state)

        #TODO: Call the play_move method with new_state,player,empty_cell
```

```
        play_move(new_state, player, empty_cell)
        #if player is computer
        if player == 'O':
#TODO: Call getBestMove method with new_state and human player ('X') to make more depth tree for human
            result,_ = getBestMove(new_state, 'X')
            move['score'] = result
        else:
#TODO: Call getBestMove method with new_state and computer player('O') to make more depth tree for
computer
            result,_ = getBestMove(new_state, 'O')
            move['score'] = result

        moves.append(move)
  # Find best move
    best_move = None
    #Check if player is computer('O')
    if player == "O":
        #TODO: Set best as -infinity for computer
        best = -infinity
        for move in moves:
            #TODO: Check if move['score'] is greater than best
            if move['score'] > best:
                best = move['score']
                best_move = move['index']

    else:
        #TODO: Set best as infinity for human
        best = infinity
        for move in moves:
            #TODO: Check if move['score'] is less than best
            if move['score'] < best:
                best = move['score']
                best_move = move['index']
    return (best, best_move)

# Now PLaying the Tic-Tac-Toe Game
play_again = 'Y'
while play_again == 'Y' or play_again == 'y':
    #Set the empty board for Tic-Tac-Toe
    game_state = [[' ',' ',' '],
                  [' ',' ',' '],
                  [' ',' ',' ']]
    #Set current_state as "Not Done"
    current_state = "Not Done"
```

```
print("\nNew Game!")

#print the game_state
print_board(game_state)

#Select the player_choice to start the game
player_choice = input("Choose which player goes first - X (You) or O(Computer): ")

#Set winner as None
winner = None
#if player_choice is ('X' or 'x') for humans else for computer
if player_choice == 'X' or player_choice == 'x':
    #TODO: Set current_player_idx is 0
    current_player_idx = 0
else:
    #TODO: Set current_player_idx is 1
    current_player_idx = 1

while current_state == "Not Done":
    #For Human Turn
    if current_player_idx == 0:
        block_choice = int(input("\nYour turn please! Choose where to place (1 to 9): "))
#TODO: Call play_move with parameters as game_state ,players[current_player_idx], block_choice
        play_move(game_state ,players[current_player_idx], block_choice)
    else:   # Computer turn _,block_choice = getBestMove(game_state, players[current_player_idx])
#TODO: Call play_move with parameters as game_state ,players[current_player_idx], block_choice
        play_move(game_state ,players[current_player_idx], block_choice)
        print("\nAI plays move: " + str(block_choice))
    print_board(game_state)
#TODO: Call check_current_state function for game_state winner, current_state = check_current_state
(game_state)
    if winner is not None:
        print(str(winner) + " won!")
    else:
        current_player_idx = (current_player_idx + 1)%2
    if current_state == "Draw":
        print("Draw!")
play_again = input('Wanna try again?(Y/N) : ')
if play_again == 'N':
    print('Thank you for playing Tic-Tac-Toe Game!!!!!!!')
```

Output

```
*IDLE Shell 3.11.1*
File  Edit  Shell  Debug  Options  Window  Help
====== RESTART: C:\Users\sumit\Desktop\Sem 6\AI\py\tic tac toe Min max.py ======

New Game!
----------------
|   ||   ||   |
----------------
|   ||   ||   |
----------------
|   ||   ||   |
----------------
Choose which player goes first - X (You) or O(Computer): X

Your turn please! Choose where to place (1 to 9): 5
----------------
|   ||   ||   |
----------------
|   || X ||   |
----------------
|   ||   ||   |
----------------

AI plays move: 1
----------------
| O ||   ||   |
----------------
|   || X ||   |
----------------
|   ||   ||   |
----------------

Your turn please! Choose where to place (1 to 9): 8
----------------
| O ||   ||   |
----------------
|   || X ||   |
----------------
|   || X ||   |
----------------

AI plays move: 2
----------------
| O || O ||   |
----------------
|   || X ||   |
----------------
|   || X ||   |
----------------
```

```
Your turn please! Choose where to place (1 to 9): 3
----------------
| O || O || X |
----------------
|   || X ||   |
----------------
|   || X ||   |
----------------

AI plays move: 7
----------------
| O || O || X |
----------------
|   || X ||   |
----------------
| O || X ||   |
----------------

Your turn please! Choose where to place (1 to 9): 4
----------------
| O || O || X |
----------------
| X || X ||   |
----------------
| O || X ||   |
----------------

AI plays move: 6
----------------
| O || O || X |
----------------
| X || X || O |
----------------
| O || X ||   |
----------------

Your turn please! Choose where to place (1 to 9): 9
----------------
| O || O || X |
----------------
| X || X || O |
----------------
| O || X || X |
----------------
Draw!
Wanna try again?(Y/N) : N
Thank you for playing Tic-Tac-Toe Game!!!!!!!
```

Lab 6B

Aim- To implement Tic-Tac-Toe game using Alpha Beta pruning Algorithm.

Theory

Alpha-beta pruning is a modified version of the minimax algorithm. It is an optimization technique for the minimax algorithm.

Pruning is a technique by which without checking each node of the game tree we can compute the correct minimax decision. This involves two threshold parameter Alpha and beta for future expansion, so it is called alpha-beta pruning. It is also called as Alpha-Beta Algorithm.

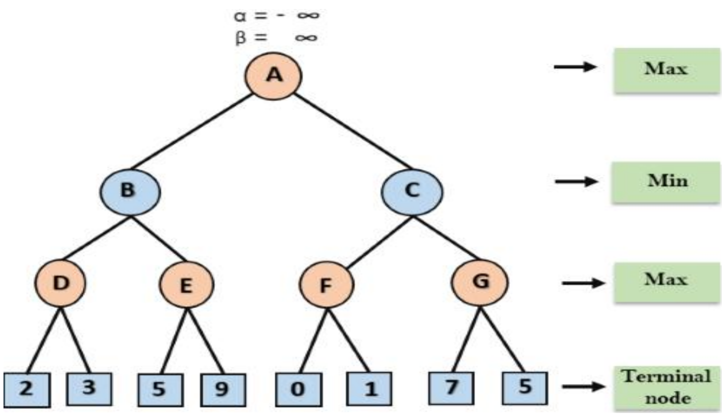
Alpha- The best (highest-value) choice we have found so far at any point along the path of Maximizer. The initial value of alpha is -∞.

Beta- The best (lowest-value) choice we have found so far at any point along the path of Minimizer. The initial value of beta is +∞.

Condition for Alpha-beta pruning

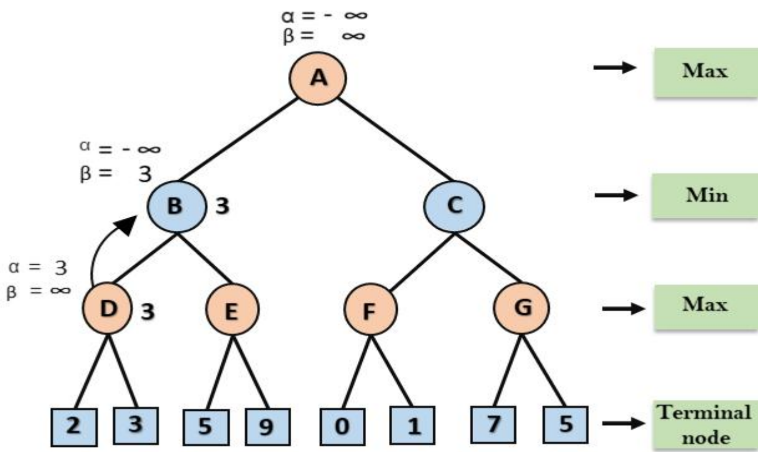
$$\alpha >= \beta$$

Step 1: Max player will start first move from node A where α= -∞ and β= +∞, these value of alpha and beta passed down to node B where again α= -∞ and β= +∞, and Node B passes the same value to its child D.

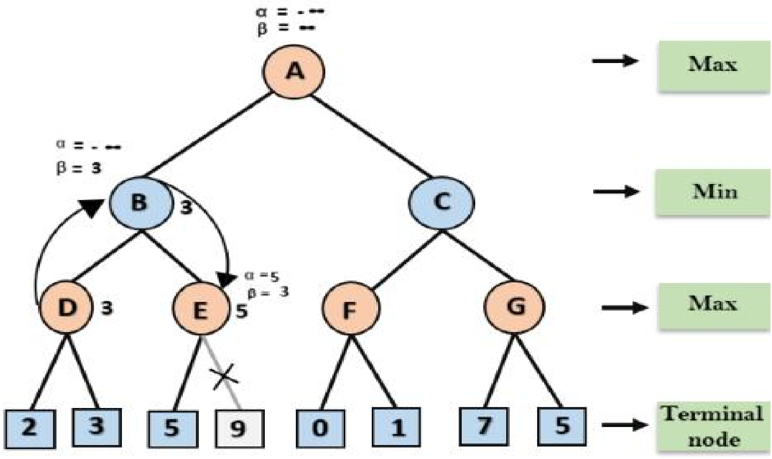


Step 2: At Node D, the value of α will be calculated as its turn for Max. The value of α is compared with firstly 2 and then 3, and the max (2, 3) = 3 will be the value of α at node D and node value will also be 3.

Step 3: Now algorithm backtrack to node B, where the value of β will change as this is a turn of Min, Now β= +∞, will compare with the available subsequent nodes value, i.e. min (∞, 3) = 3, hence at node B now α= -∞, and β= 3.

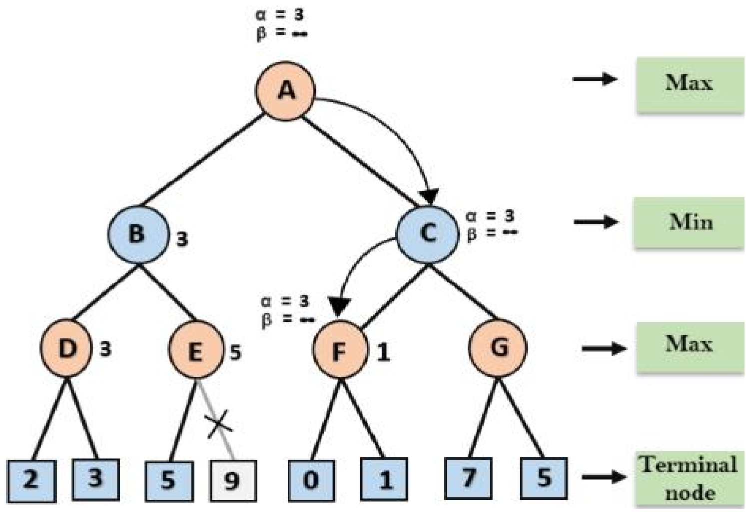


Step 4: At node E, Max will take its turn, and the value of alpha will change. The current value of alpha will be compared with 5, so max (-∞, 5) = 5, hence at node E α= 5 and β= 3, where α>=β, so the right successor of E will be pruned, and algorithm will not traverse it, and the value at node E will be 5.

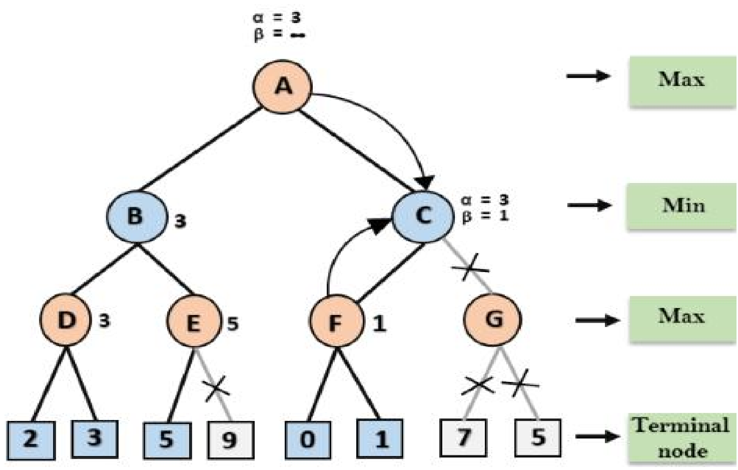


Step 5: Algorithm again backtrack the tree, from node B to node A. At node A, the value of alpha will be changed the maximum available value is 3 as max (-∞, 3)= 3, and β= +∞, these two values now passes to right successor of A which is Node C. At node C, α=3 and β= +∞, and the same values will be passed on to node F.

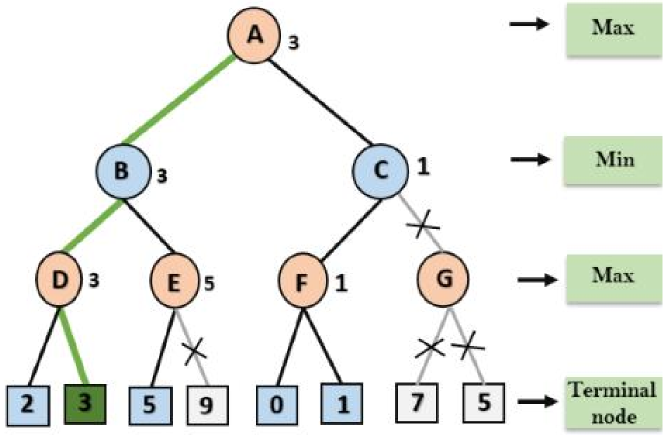
Step 6: At node F, again the value of α will be compared with left child which is 0, and max(3,0)= 3, and then compared with right child which is 1, and max(3,1)= 3 still α remains 3, but the node value of F will become 1.



Step 7: Node F returns the node value 1 to node C, at C α= 3 and β= +∞, here the value of beta will be changed, it will compare with 1 so min (∞, 1) = 1. Now at C, α=3 and β= 1, and again it satisfies the condition α>=β, so the next child of C which is G will be pruned, and the algorithm will not compute the entire sub-tree G.



Step 8: Node C now returns the value of 1 to A here the best value for A is max (3, 1) = 3. Following is the final game tree which is the showing the nodes which are computed and nodes which has never computed. Hence the optimal value for the maximizer is 3.



Code

```
import numpy as np
from math import inf as infinity

#Set the Empty Board
game_state = [[' ',' ',' '],
              [' ',' ',' '],
              [' ',' ',' ']]
#Create the Two Players as 'X'/'O'
players = ['X','O']
pruned=0

#Method for checking the correct move on Tic-Tac-Toe
def play_move(state, player, block_num):
    if state[int((block_num-1)/3)][(block_num-1)%3] == ' ':
        #TODO: Assign the player move on the current position of Tic-Tac-Toe if condition is True
        state[int((block_num-1)/3)][(block_num-1)%3] = player
    else:
        block_num = int(input("Block is not empty, ya blockhead! Choose again: "))
        play_move(state, player, block_num)
        #TODO: Recursively call the play_move
#Method to copy the current game state to new_state of Tic-Tac-Toe
def copy_game_state(state):
    new_state = [[' ',' ',' '],[' ',' ',' '],[' ',' ',' ']]
    for i in range(3):
        for j in range(3):
            #TODO: Copy the Tic-Tac-Toe state to new_state
            new_state[i][j] = state[i][j]
    #TODO: Return the new_state
```

```
    return new_state

#Method to check the current state of the Tic-Tac-Toe
def check_current_state(game_state):
    #TODO: Set the draw_flag to 0
    draw_flag = 0
    for i in range(3):
        for j in range(3):
            if game_state[i][j] == ' ':
                draw_flag = 1

    if draw_flag == 0:
        return None, "Draw"

    # Check horizontals in first row
    if (game_state[0][0]==game_state[0][1] and game_state[0][1]==game_state[0][2] and game_state[0][0] != ' '):
        return game_state[0][0], "Done"
    #TODO: Check horizontals in second row
    if (game_state[1][0]==game_state[1][1] and game_state[1][1]==game_state[1][2] and game_state[1][0] != ' '):
        return game_state[1][0], "Done"
    #TODO: Check horizontals in third row
    if (game_state[2][0]==game_state[2][1] and game_state[2][1]==game_state[2][2] and game_state[2][0] != ' '):
        return game_state[2][0], "Done"

    # Check verticals in first column
    if (game_state[0][0]==game_state[1][0] and game_state[1][0]==game_state[2][0] and game_state[0][0] != ' '):
        return game_state[0][0], "Done"
    # Check verticals in second column
    if (game_state[0][1]==game_state[1][1] and game_state[1][1]==game_state[2][1] and game_state[0][1] != ' '):
        return game_state[0][1], "Done"
    # Check verticals in third column
    if (game_state[0][2]==game_state[1][2] and game_state[1][2]==game_state[2][2] and game_state[0][2] != ' '):
        return game_state[0][2], "Done"

    # Check left diagonal
    if (game_state[0][0]==game_state[1][1] and game_state[1][1]==game_state[2][2] and game_state[0][0] != ' '):
        return game_state[1][1], "Done"
    # Check right diagonal
    if (game_state[2][0]==game_state[1][1] and game_state[1][1]==game_state[0][2] and game_state[2][0] != ' '):
        return game_state[1][1], "Done"

    return None, "Not Done"

#Method to print the Tic-Tac-Toe Board
def print_board(game_state):
    print('----------------')
```

```
    print('| ' + str(game_state[0][0]) + ' || ' + str(game_state[0][1]) + ' || ' + str(game_state[0][2]) + ' |')
    print('----------------')
    print('| ' + str(game_state[1][0]) + ' || ' + str(game_state[1][1]) + ' || ' + str(game_state[1][2]) + ' |')
    print('----------------')
    print('| ' + str(game_state[2][0]) + ' || ' + str(game_state[2][1]) + ' || ' + str(game_state[2][2]) + ' |')
    print('----------------')

#Method for implement the alpha beta pruning function
def alphabeta(state,depth,alpha,beta,player):
    global pruned
    #TODO: call the check_current_state method using state parameter
    winner_loser , done = check_current_state(state)
    #TODO:Check condition for winner if winner_loser is 'O' then Computer won
    #else if winner_loser is 'X' then You won else game is draw
    if done == "Done" and winner_loser == 'O': # If AI won
        return (1,0)
    elif done == "Done" and winner_loser == 'X': # If Human won
        return (-1,0)
    elif done == "Draw":    # Draw condition
        return (0,0)
    #TODO: set moves to empty list
    moves = []
    #TODO: set empty_cells to empty list
    empty_cells = []
    #Append the block_num to the empty_cells list
    for i in range(3):
        for j in range(3):
            if state[i][j] == ' ':
                empty_cells.append(i*3 + (j+1))

    #TODO:Iterate over all the empty_cells
    for empty_cell in empty_cells:
        #TODO: create the empty dictionary
        move = {}

        #TODO: Assign the empty_cell to move['index']
        move['index'] = empty_cell

        #Call the copy_game_state method
        new_state = copy_game_state(state)

        #TODO: Call the play_move method with new_state,player,empty_cell
        play_move(new_state, player, empty_cell)
        #if player is computer
```

```
        if player == 'O':
#TODO: Call getBestMove method with new_state and human player ('X') to make more depth tree for human
            result = alphabeta(new_state, depth-1, alpha, beta,False)[1]
            move['score'] = result
        else:
#TODO:Call getBestMove method with new_state and computer player('O') to make more depth tree for computer
            result = alphabeta(new_state, depth-1, alpha, beta,True)[1]
            move['score'] = result

        moves.append(move)

# Find best move
    best_move = None
    #Check if player is computer('O')
    if player == "O":
        #TODO: Set best as -infinity for computer
        best = - infinity
        for move in moves:
            #TODO: Check if move['score'] is greater than best
            if move['score'] > best:
                best = move['score']
                best_move = move['index']
            alpha=max(alpha, best)
            if alpha >= beta:
                pruned+=(3-depth)**2
                # Increment pruned counter
                break
    else:
        #TODO: Set best as infinity for human
        best = infinity
        for move in moves:
        #TODO: Check if move['score'] is less than best
            if move['score'] < best:
                best = move['score']
                best_move = move['index']
            beta=min(alpha, best)
            if alpha >= beta:
                pruned+=(3-depth)**2
                break

    return (best, best_move,pruned)

# Now PLaying the Tic-Tac-Toe Game
```

```
play_again = 'Y'
while play_again == 'Y' or play_again == 'y':
   depth = 9
   #Set the empty board for Tic-Tac-Toe
   game_state = [[' ',' ',' '],
                 [' ',' ',' '],
                 [' ',' ',' ']]
   pruned=0
   #Set current_state as "Not Done"
   current_state = "Not Done"
   print("\nNew Game!")
   #print the game_state
   print_board(game_state)
   #Select the player_choice to start the game
   player_choice = input("Choose which player goes first - X (You) or O(Computer): ")
   #Set winner as None
   winner = None
   #if player_choice is ('X' or 'x') for humans else for computer
   if player_choice == 'X' or player_choice == 'x':
      #TODO: Set current_player_idx is 0
      current_player_idx = 0
   else:
      #TODO: Set current_player_idx is 1
      current_player_idx = 1
   while current_state == "Not Done":
      #For Human Turn
      if current_player_idx == 0:
         block_choice = int(input("Your turn please! Choose where to place (1 to 9): "))
         #TODO: Call the play_move with parameters as game_state ,players[current_player_idx], block_choice
         play_move(game_state ,players[current_player_idx], block_choice)
      else:
         best_move, best_score, pruned = alphabeta(game_state, depth, float('-inf'), float('inf'),True)
         play_move(game_state ,players[current_player_idx], best_move)
         print(f"Best move: {best_move}, score: {best_score}, pruned: {pruned}")
         print("\nAI plays move: " + str(best_move))

# Computer turn block_choice = getBestMove(game_state,float('inf'), float('inf'), players[current_player_idx], pruned_states)
#TODO: Call the play_move with parameters as game_state ,players[current_player_idx], block_choice

      print_board(game_state)
      #TODO: Call the check_current_state function for game_state
      winner, current_state = check_current_state(game_state)
```

```
        if winner is not None:
            print(str(winner) + " won!")
        else:
            current_player_idx = (current_player_idx + 1)%2

        if current_state == "Draw":
            print("Draw!")

    play_again = input('Wanna try again?(Y/N) : ')
    if play_again == 'N':
        print('Thank you for playing Tic-Tac-Toe Game!!!!!!!')
```

<u>Output</u>

```
IDLE Shell 3.11.1
File  Edit  Shell  Debug  Options  Window  Help
======= RESTART: C:/Users/sumit/Desktop/tic tac toe alpha beta pruning.py ======

New Game!
----------------
|   ||   ||   |
----------------
|   ||   ||   |
----------------
|   ||   ||   |
----------------
Choose which player goes first - X (You) or O(Computer): O
Best move: 2, score: 1, pruned: 1666125

AI plays move: 2
----------------
|   || O ||   |
----------------
|   ||   ||   |
----------------
|   ||   ||   |
----------------
Your turn please! Choose where to place (1 to 9): 7
----------------
|   || O ||   |
----------------
|   ||   ||   |
----------------
| X ||   ||   |
----------------
Best move: 3, score: 1, pruned: 1674778

AI plays move: 3
----------------
|   || O || O |
----------------
|   ||   ||   |
----------------
| X ||   ||   |
----------------
Your turn please! Choose where to place (1 to 9): 1
----------------
| X || O || O |
----------------
|   ||   ||   |
----------------
| X ||   ||   |
----------------
```

```
Best move: 5, score: 4, pruned: 1676279

AI plays move: 5
----------------
| X || O || O |
----------------
|   || O ||   |
----------------
| X ||   ||   |
----------------
Your turn please! Choose where to place (1 to 9): 8
----------------
| X || O || O |
----------------
|   || O ||   |
----------------
| X || X ||   |
----------------
Best move: 6, score: 4, pruned: 1676486

AI plays move: 6
----------------
| X || O || O |
----------------
|   || O || O |
----------------
| X || X ||   |
----------------
Your turn please! Choose where to place (1 to 9): 4
----------------
| X || O || O |
----------------
| X || O || O |
----------------
| X || X ||   |
----------------
X won!
Wanna try again?(Y/N) : N
Thank you for playing Tic-Tac-Toe Game!!!!!!!
>>>
```

Lab 7

Aim- Write a python program for the cryptarithmetic problem APPLE + LEMON = BANANA.

Theory

Cryptarithmetic Problem is a type of constraint satisfaction problem where the game is about digits and its unique replacement either with alphabets or other symbols. In cryptarithmetic problem, the digits (0-9) get substituted by some possible alphabets or symbols. The task in cryptarithmetic problem is to substitute each digit with an alphabet to get the result arithmetically correct.

We can perform all the arithmetic operations on a given cryptarithmetic problem.

The rules or constraints on a cryptarithmetic problem are as follows:

- 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)

Consider the equation APPLE + LEMON = BANANA. Assume that each letter actually represents a digit from 0 to 9. Some conditions are imposed. The leftmost letter can't be zero in any word. There must be a one-to-one mapping between letters and digits. In other words, if you choose the digit 5 for the letter E, then all of the E's in the equation must be 5 and no other letter can be a 5. No digit can be repeated.

	1	1	1		
	A	P	P	L	E
	6	7	7	9	4
+	L	E	M	O	N
	9	4	8	3	2
B	A	N	A	N	A
1	6	2	6	2	6

Code

```
def find_value(word, assigned):
  num = 0
  for char in word:
    num = num * 10
    num += assigned[char]
```

```
  return num
def is_valid_assignment(word1, word2, result, assigned):
  # First letter of any word cannot be zero.
  if assigned[word1[0]] == 0 or assigned[word2[0]] == 0 or assigned[result[0]] == 0:
    return False
  return True

def _solve(word1, word2, result, letters, assigned, solutions):
  if not letters:
    if is_valid_assignment(word1, word2, result, assigned):
      num1 = find_value(word1, assigned)
      num2 = find_value(word2, assigned)
      num_result = find_value(result, assigned)
      if num1 + num2 == num_result:
        solutions.append((f'{num1} + {num2} = {num_result}', assigned.copy()))
    return

  for num in range(10):
    if num not in assigned.values():
      cur_letter = letters.pop()
      assigned[cur_letter] = num
      _solve(word1, word2, result, letters, assigned, solutions)
      assigned.pop(cur_letter)
      letters.append(cur_letter)

def solve(word1, word2, result):
  letters = sorted(set(word1) | set(word2) | set(result))
  if len(result) > max(len(word1), len(word2)) + 1 or len(letters) > 10:
    print('0 Solutions!')
    return

  solutions = []
```

```
    _solve(word1, word2, result, letters, {}, solutions)
    if solutions:
        print('\nSolutions:')
        for soln in solutions:
            print(f'{soln[0]}\t{soln[1]}')

if __name__ == '__main__':
    print('CRYPTARITHMETIC PUZZLE SOLVER')
    print('WORD1 + WORD2 = RESULT')
    word1 = input('Enter WORD 1: ').upper()
    word2 = input('Enter WORD 2: ').upper()
    result = input('Enter RESULT: ').upper()

    if not word1.isalpha() or not word2.isalpha() or not result.isalpha():
        raise TypeError('\nInputs should only consist of alphabets.')

    solve(word1, word2, result)
```

Output

```
IDLE Shell 3.11.1
File Edit Shell Debug Options Window Help
Python 3.11.1 (tags/v3.11.1:a7a450f, Dec  6 2022, 19:58:39) [MSC v.1934 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
==================== RESTART: C:/Users/sumit/Desktop/Lab 7.py ====================
CRYPTARITHMETIC PUZZLE SOLVER
WORD1 + WORD2 = RESULT
Enter WORD 1: APPLE
Enter WORD 2: LEMON
Enter RESULT: BANANA

Solutions:
67794 + 94832 = 162626  {'P': 7, 'O': 3, 'N': 2, 'M': 8, 'L': 9, 'E': 4, 'B': 1, 'A': 6}
>>>
```

Lab 8

Aim- To implement graph colouring problem in python.

Theory

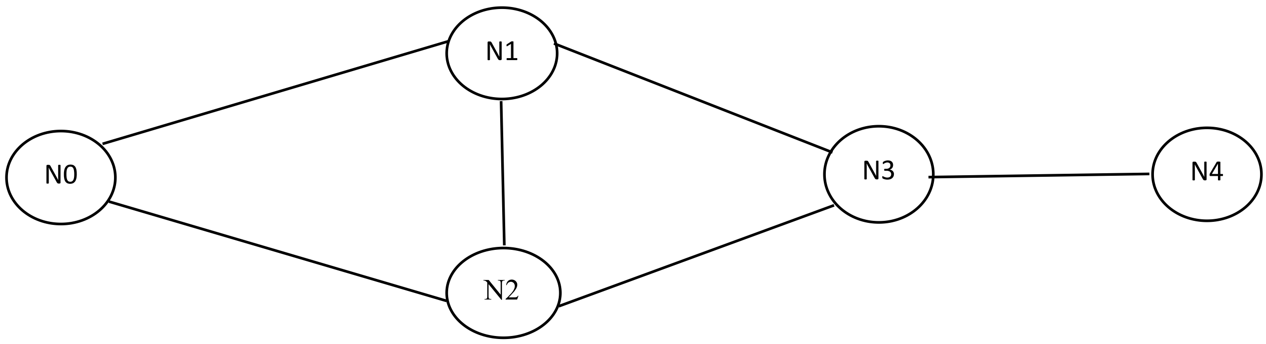
The graph colouring problem is a classical problem in computer science and mathematics that involves assigning colours to the vertices of a graph in such a way that no two adjacent vertices have the same colour. The greedy algorithm is a simple algorithm that can be used to solve this problem.

The greedy algorithm for the graph colouring problem works as follows:

1. Sort the vertices of the graph in some order.
2. Initialize an array of colours, where the colour of each vertex is initially set to 0 (i.e., no colour).
3. For each vertex v in the sorted order:
    a. Consider the colours of its neighbours.
    b. Choose the smallest colour that is not used by any of its neighbours.
    c. Assign that colour to v.
4. Return the array of colours.

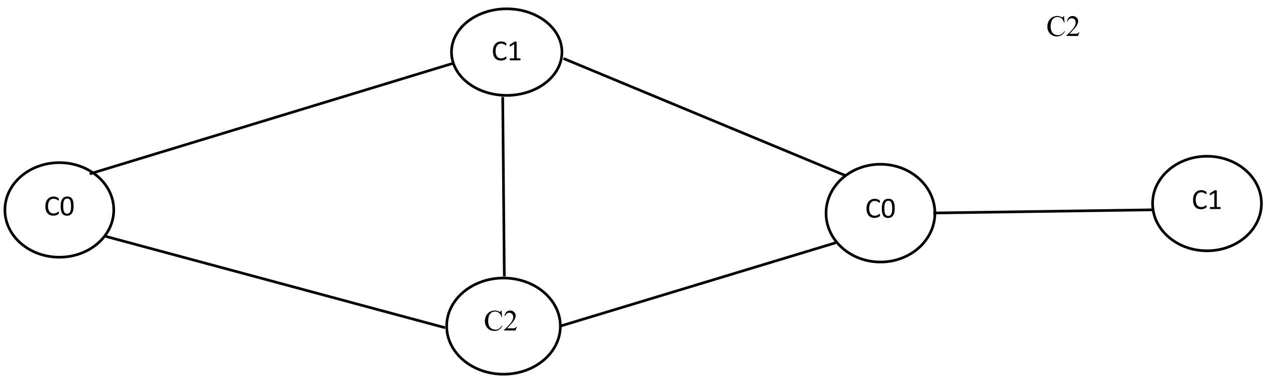
The backtracking algorithm for the graph colouring problem works as follows:

1. Choose an uncoloured vertex v.
2. For each possible colour c that can be assigned to v:
    a. Check if c is a valid colour for v (i.e., no adjacent vertex of v has colour c).
    b. If c is a valid colour for v, assign colour c to v and recursively apply the same steps to the next uncoloured vertex.
    c. If no valid colour can be found for v, backtrack (i.e., undo the colour assignment to v) and try a different colour for the previous vertex.
3. If all vertices are coloured, the algorithm has found a valid colour assignment.



Graph used for the program

Colours used:
C0
C1
C2



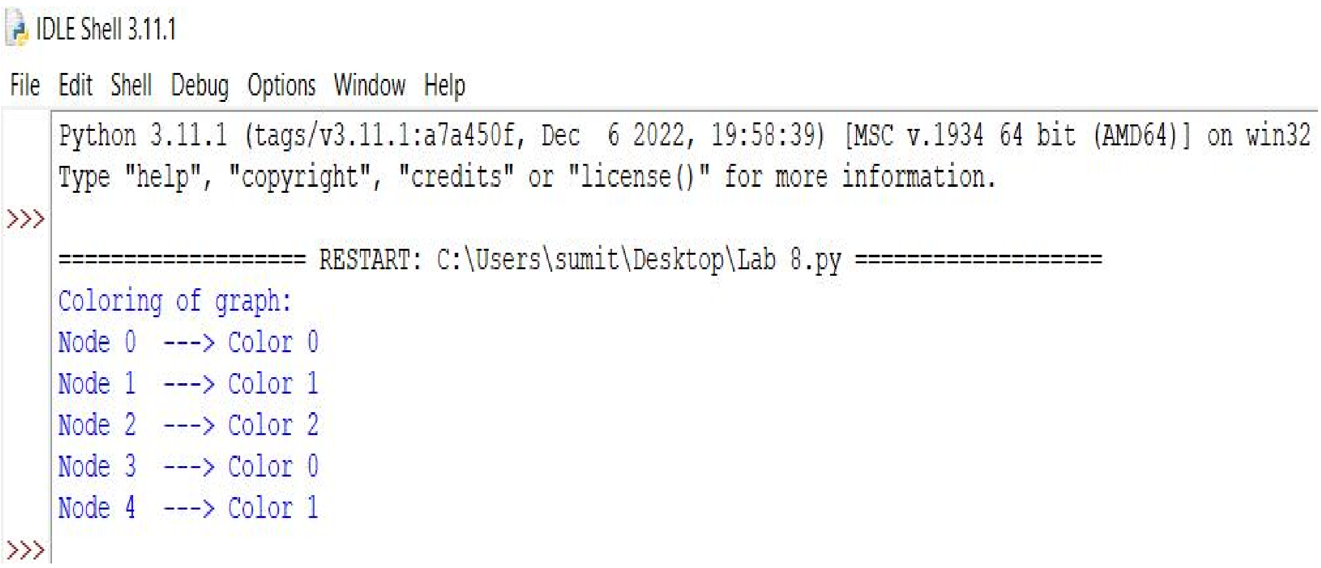
Graph after colours are assigned

Code

```
def addEdge(adj, v, w):
        adj[v].append(w)
        adj[w].append(v)                    # The graph is undirected
        return adj

# Assigns colors (starting from 0) to all vertices and prints the assignment of colors
def greedyColoring(adj, V):
        result = [-1] * V
        result[0] = 0                       # Assign the first color to first vertex

# A temporary array to store the available colors.
# True value of available[cr] would mean that the color cr is assigned to one of its adjacent vertices
        available = [False] * V

# Assign colors to remaining V-1 vertices
        for u in range(1, V):

                # Process all adjacent vertices and flag their colors as unavailable
                for i in adj[u]:
                        if (result[i] != -1):
                                available[result[i]] = True

                # Find the first available color
                cr = 0
                while cr < V:
                        if (available[cr] == False):
                                break
                        cr += 1

                # Assign the found color
                result[u] = cr
```

```
        # Reset the values back to false for the next iteration
        for i in adj[u]:
            if (result[i] != -1):
                available[result[i]] = False

    # Print the result
    for u in range(V):
        print("Vertex", u, " ---> Color", result[u])

# Driver Code
if __name__ == '__main__':
    g1 = [[] for i in range(5)]
    g1 = addEdge(g1, 0, 1)
    g1 = addEdge(g1, 0, 2)
    g1 = addEdge(g1, 1, 2)
    g1 = addEdge(g1, 1, 3)
    g1 = addEdge(g1, 2, 3)
    g1 = addEdge(g1, 3, 4)
    print("Coloring of graph: ")
    greedyColoring(g1, 5)
```

Output



Lab 9

Aim- Write a program for tokenization of word and sentence using NLTK package in python.

Also perform:

1. Stop word removal
2. Stemming
3. Lemmatization
4. POS tagging (parsing)
5. Parsing of a sentence

Theory

The Natural Language Toolkit (NLTK) is a popular open-source Python library used for natural language processing (NLP) tasks such as tokenization, stemming, lemmatization, part-of-speech tagging, named entity recognition, parsing, and semantic analysis. NLTK provides a comprehensive set of tools and resources for processing and analysing human language data.

Tokenization is essentially splitting a phrase, sentence, paragraph, or an entire text document into smaller units, such as individual words or terms. Each of these smaller units are called tokens. The tokens could be words, numbers or punctuation marks. In tokenization, smaller units are created by locating word boundaries. These are the ending point of a word and the beginning of the next word.

Stop words are commonly used words that are filtered out from natural language processing tasks like text analysis and information retrieval. These words are considered insignificant and do not add meaning to the content of a document or sentence. Stop words typically include pronouns, prepositions, conjunctions, and other frequently occurring words in a language, such as "the", "and", "a", "an", "in", "on", "of", "to", etc.

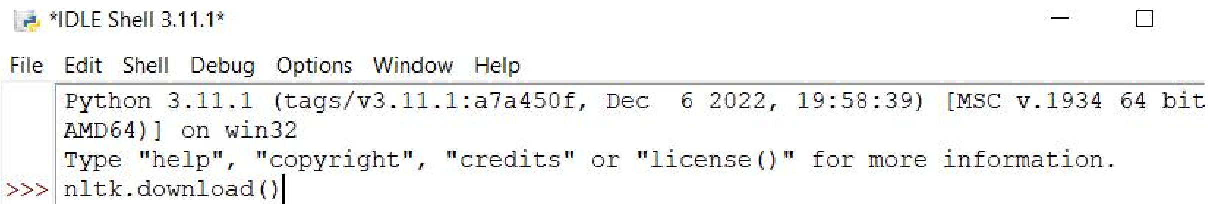
Stemming is a text processing task in which words can be reduce to their root, which is the core part of a word. For example, the words “helping” and “helper” share the root “help.” Stemming allows you to zero in on the basic meaning of a word rather than all the details of how it’s being used. Types of stemming:

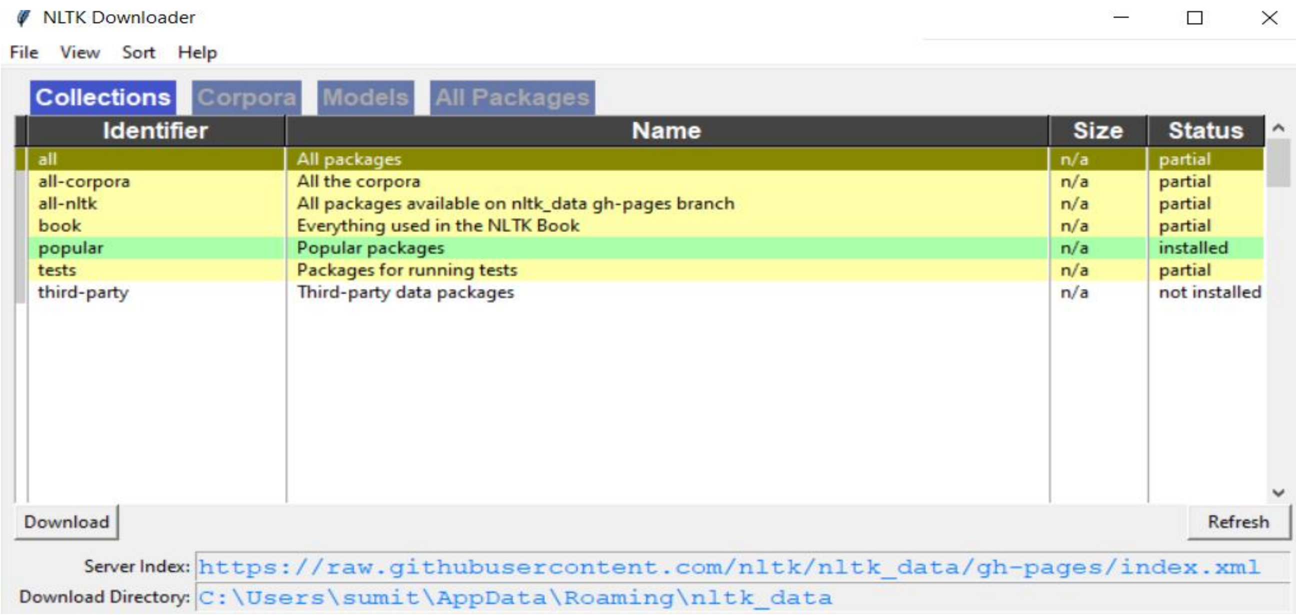
1. **Understemming** happens when two related words should be reduced to the same stem but aren’t. This is a false negative.
2. **Overstemming** happens when two unrelated words are reduced to the same stem even though they shouldn’t be. This is a false positive.

Lemmatization reduces words to their core meaning, but it will give a complete English word that makes sense on its own instead of just a fragment of a word like 'discoveri'.

Lemma is a word that represents a whole group of words, and that group of words is called a **lexeme**. For example, if you were to look up the word “blending” in a dictionary, then you’d need to look at the entry for “blend,” but you would find “blending” listed in that entry. In this example, “blend” is the lemma, and “blending” is part of the lexeme.

Parts of Speech Tagging(POS) is a process to mark up the words in text format for a particular part of a speech based on its definition and context. It is responsible for text reading in a language and assigning some specific token (Parts of Speech) to each word. It is also called grammatical tagging.





Code

#Code for Tokenization of words and sentence

import nltk

from nltk.tokenize import sent_tokenize, word_tokenize

text = "Cryptarithmetic Problem is a type of constraint satisfaction problem where the game is about digits and its unique replacement " + \

"either with alphabets or other symbols. In cryptarithmetic problem, the digits (0-9) get substituted by some possible alphabets or symbols. " + \

"The task in cryptarithmetic problem is to substitute each digit with an alphabet to get the result arithmetically correct. "

nltk_tokens = nltk.sent_tokenize(text)

print(word_tokenize(text))

```
print("\n")
print(sent_tokenize(text))
```

Output

```
IDLE Shell 3.11.1
File Edit Shell Debug Options Window Help
Python 3.11.1 (tags/v3.11.1:a7a450f, Dec  6 2022, 19:58:39) [MSC v.1934 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
================== RESTART: C:\Users\sumit\Desktop\Lab 9.py ==================
['Cryptarithmetic', 'Problem', 'is', 'a', 'type', 'of', 'constraint', 'satisfaction', 'problem',
 'where', 'the', 'game', 'is', 'about', 'digits', 'and', 'its', 'unique', 'replacement', 'either
', 'with', 'alphabets', 'or', 'other', 'symbols', '.', 'In', 'cryptarithmetic', 'problem', ',',
'the', 'digits', '(', '0-9', ')', 'get', 'substituted', 'by', 'some', 'possible', 'alphabets', '
or', 'symbols', '.', 'The', 'task', 'in', 'cryptarithmetic', 'problem', 'is', 'to', 'substitute'
, 'each', 'digit', 'with', 'an', 'alphabet', 'to', 'get', 'the', 'result', 'arithmetically', 'co
rrect', '.']


['Cryptarithmetic Problem is a type of constraint satisfaction problem where the game is about d
igits and its unique replacement either with alphabets or other symbols.', 'In cryptarithmetic p
roblem, the digits (0-9) get substituted by some possible alphabets or symbols.', 'The task in c
ryptarithmetic problem is to substitute each digit with an alphabet to get the result arithmetic
ally correct.']
```

```
#Code for stop word removal
import nltk
from nltk.corpus import stopwords
nltk.download('stopwords')
from nltk.tokenize import word_tokenize
text = "Nick likes to play football, however he is not too fond of tennis."
text_tokens = word_tokenize(text)
tokens_without_sw = [word for word in text_tokens if not word in stopwords.words()]
print("After removing stop words:")
print(tokens_without_sw)
```

Output

```
IDLE Shell 3.11.1
File  Edit  Shell  Debug  Options  Window  Help
    Python 3.11.1 (tags/v3.11.1:a7a450f, Dec  6 2022, 19:58:39) [MSC v.1934 64 bit (AMD64)] on win32
    Type "help", "copyright", "credits" or "license()" for more information.
>>>
    ================== RESTART: C:\Users\sumit\Desktop\Lab 9.py ===================
    After removing stop words:
    ['Nick', 'likes', 'play', 'football', ',', 'fond', 'tennis', '.']
```

```
#Code for stemming
from nltk.stem import PorterStemmer
from nltk.tokenize import word_tokenize
ps = PorterStemmer()
words = ["program", "programs", "programmer", "programming", "programmers"]
sentence = "Stemming is a text processing task where words can be reduced to roots"
print("\nWord Stemming")
for w in words:
        print(w, " : ", ps.stem(w))

print("\nSentence Stemming")
words = word_tokenize(sentence)

for w in words:
   print(w, " : ", ps.stem(w))
```

Output

```
IDLE Shell 3.11.1
File  Edit  Shell  Debug  Options  Window  Help
    Python 3.11.1 (tags/v3.11.1:a7a450f, Dec  6 2022, 19:58:39)
    Type "help", "copyright", "credits" or "license()" for more
>>>
    ==================== RESTART: C:\Users\sumit\Desktop\Lab 9.py
    Word Stemming
    program  :  program
    programs  :  program
    programmer  :  programm
    programming  :  program
    programmers  :  programm

    Sentence Stemming
    Stemming  :  stem
    is  :  is
    a  :  a
    text  :  text
    processing  :  process
    task  :  task
    where  :  where
    words  :  word
    can  :  can
    be  :  be
    reduced  :  reduc
    to  :  to
    roots  :  root
```

```
#Code for Lemmatization
import nltk
from nltk.stem import WordNetLemmatizer
wordnet_lemmatizer = WordNetLemmatizer()
text = "The children were running and laughing in the playground."
tokenization = nltk.word_tokenize(text)
print("\n")
for w in tokenization:
   print("Lemma for {} is {}".format(w, wordnet_lemmatizer.lemmatize(w)))
```

Output

```
IDLE Shell 3.11.1
File Edit Shell Debug Options Window Help
Python 3.11.1 (tags/v3.11.1:a7a450f, Dec  6 2022, 19:58:39)
Type "help", "copyright", "credits" or "license()" for more
>>>
=================== RESTART: C:\Users\sumit\Desktop\Lab 9.py
Lemma for The is The
Lemma for children is child
Lemma for were is were
Lemma for running is running
Lemma for and is and
Lemma for laughing is laughing
Lemma for in is in
Lemma for the is the
Lemma for playground is playground
Lemma for . is .
```

```
#Code for POS tagging
import nltk
from nltk.corpus import stopwords
from nltk.tokenize import word_tokenize, sent_tokenize
stop_words = set(stopwords.words('english'))
txt ="Parts of Speech Tagging is a process to mark up the words in text " \
    "format for a particular part of a speech based on its definition " \
    "It is responsible for text reading in a language " \
    "It is also called grammatical tagging. "
tokenized = sent_tokenize(txt)
for i in tokenized:
  wordsList = nltk.word_tokenize(i)
#Using a POS Tagger.
tagged = nltk.pos_tag(wordsList)
print("\nPOS tagged text:")
print(tagged)
```

Output

```
POS tagged text:
[('Parts', 'NNS'), ('of', 'IN'), ('Speech', 'NNP'), ('Tagging', 'NNP'), ('is', 'VBZ'), ('
a', 'DT'), ('process', 'NN'), ('to', 'TO'), ('mark', 'VB'), ('up', 'RP'), ('the', 'DT'),
('words', 'NNS'), ('in', 'IN'), ('text', 'JJ'), ('format', 'NN'), ('for', 'IN'), ('a', 'D
T'), ('particular', 'JJ'), ('part', 'NN'), ('of', 'IN'), ('a', 'DT'), ('speech', 'NN'), (
'based', 'VBN'), ('on', 'IN'), ('its', 'PRP$'), ('definition', 'NN'), ('It', 'PRP'), ('is
', 'VBZ'), ('responsible', 'JJ'), ('for', 'IN'), ('text', 'JJ'), ('reading', 'NN'), ('in'
, 'IN'), ('a', 'DT'), ('language', 'NN'), ('It', 'PRP'), ('is', 'VBZ'), ('also', 'RB'), (
'called', 'VBN'), ('grammatical', 'JJ'), ('tagging', 'NN'), ('.', '.')]
```

```
#Code for Parsing
import nltk
```

```
from nltk import word_tokenize
from nltk.parse import RecursiveDescentParser
sentence = "the cat sat on the mat"
tokens = word_tokenize(sentence)
# Define a grammar for parsing the sentence
grammar = nltk.CFG.fromstring("""
                S -> NP VP
                NP -> DT NN
                VP -> V PP
                PP -> IN NP
                DT -> 'the'
                NN -> 'cat' | 'mat'
                V -> 'sat'
                IN -> 'on'
                """)
# Create a parser object
parser = RecursiveDescentParser(grammar)
print("\nAfter parsing: ")
for tree in parser.parse(tokens):
    print(tree)
```

Output