**INDEX**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **S.no.** | **Program** | **Date**  **Assigned** | **Date**  **Submission** | **Sign** |
| 1. | To implement Logic Gates in Python.  Create an empty set using python, add members to it and discard the element if it is present in the set already. | 06/01/2023 |  |  |
| 2. | Write a program to implement BFS for Water Jug Problem using python. | 13/01/2023 |  |  |
| 3. | Write a program using Python to implement DFS. | 20/01/2023 |  |  |
| 4. | Write a program to implement A\* algorithm using Python. | 27/01/2023 |  |  |
| 5. | Write a program to implement 8-puzzle problem using BFS using Python. | 03/02/2023 |  |  |
| 6. | Write a program using python to implement tic-tac-toe game using Mini-max algorithm. | 10/02/2023 |  |  |
| 7. | Write a program using python to implement tic-tac-toe game using alpha-beta pruning algorithm. | 24/02/2023 |  |  |
| 8. | Write a python program to implement graph coloring problem. | 03/03/2023 |  |  |
| 9. | Write a python script for a given cryptarithmetic problem,  (APPLE+LEMON=BANANA). | 10/03/2023 |  |  |
| 10 | Write a Python script to demonstrate the following:   * Tokenization of Word and Sentence * Stopword Removal * Stemming * Lemmatization * POS Tagging (Parsing | 24/03/2023 |  |  |

**LAB-1**

**AIM:** To implement Logic Gates in Python.

Create an empty set using python, add members to it and discard the element if it is present in the set already.

**THEORY:**  
Logic gates are fundamental building blocks of digital circuits and are used to perform logical operations on binary input values. There are several types of logic gates, including AND, OR, NOT, and NAND gates.

An AND gate takes two binary inputs and produces a binary output that is 1 only if both inputs are 1. An OR gate also takes two binary inputs and produces a binary output that is 1 if at least one input is 1. A NOT gate takes a single binary input and produces a binary output that is the opposite of the input (0 becomes 1 and 1 becomes 0).

NAND gate is a combination of an AND gate and a NOT gate. It takes two binary inputs and produces a binary output that is the opposite of the output of an AND gate. In other words, the output of a NAND gate is 0 only if both inputs are 1.

**Code:**

# Define logic gates

def AND\_gate(input1, input2):

if input1 == 1 and input2 == 1:

return 1

else:

return 0

def OR\_gate(input1, input2):

if input1 == 0 and input2 == 0:

return 0

else:

return 1

def NOT\_gate(input1):

if input1 == 0:

return 1

else:

return 0

def NAND\_gate(input1, input2):

output\_of\_AND = AND\_gate(input1, input2)

output\_of\_NOT = NOT\_gate(output\_of\_AND)

return output\_of\_NOT

# Create a set and add elements to it

my\_set = set()

my\_set.add(1)

my\_set.add(2)

my\_set.add(3)

my\_set.add(2) # This will not add a new element since 2 is already in the set

# Remove an element from the set

my\_set.discard(2)

# Test logic gates

print("AND gate:")

print(AND\_gate(0, 0)) # Expected output: 0

print(AND\_gate(0, 1)) # Expected output: 0

print(AND\_gate(1, 0)) # Expected output: 0

print(AND\_gate(1, 1)) # Expected output: 1

print("OR gate:")

print(OR\_gate(0, 0)) # Expected output: 0

print(OR\_gate(0, 1)) # Expected output: 1

print(OR\_gate(1, 0)) # Expected output: 1

print(OR\_gate(1, 1)) # Expected output: 1

print("NOT gate:")

print(NOT\_gate(0)) # Expected output: 1

print(NOT\_gate(1)) # Expected output: 0

print("NAND gate:")

print(NAND\_gate(0, 0)) # Expected output: 1

print(NAND\_gate(0, 1)) # Expected output: 1

print(NAND\_gate(1, 0)) # Expected output: 1

print(NAND\_gate(1, 1)) # Expected output: 0

print("Set:")

print(my\_set) # Expected output: {1, 3}

**Output:**

AND gate:

0

0

0

1

OR gate:

0

1

1

1

NOT gate:

1

0

NAND gate:

1

1

1

0

Set:

{1, 3}

**LAB-2**

**AIM:** Write a program using Python to implement BFS for Water Jug Problem using python.

**THEORY:**

You are given an m liter jug and a n liter jug. Both the jugs are initially empty. The jugs don’t have markings to allow measuring smaller quantities. You have to use the jugs to measure d liters of water where d is less than n.

(X, Y) corresponds to a state where X refers to the amount of water in Jug1 and Y refers to the amount of water in Jug2

Determine the path from e initial state (xi, yi) to the final state (xf, yf), where (xi, yi) is (0, 0) which indicates both Jugs are initially empty and (xf, yf) indicates a state which could be (0, d) or (d, 0).

The operations you can perform are:

Fill a Jug, (0, 0)->(X, 0) Fill Jug 1

Pour water from one jug to the other until one of the jugs is either empty or full, (X, Y) -> (X-d, Y+d)

Examples:

Input : 4 3 2

Output : {( 0,0),(0,3),(4,0),(4,3),(3,0),(1,3),(3,3),(4,2),(0,2)}

Recommended: Please try your approach on {IDE} first, before moving on to the solution.

We have discussed the optimal solution in The Two Water Jug Puzzle. In this post, a BFS based solution is discussed.

Here, we keep exploring all the different valid cases of the states of water in the jug simultaneously until and unless we reach the required target water.

As provided in the problem statement, at any given state we can do either of the following operations:

Fill a jug

Empty a jug

Transfer water from one jug to another until either of them gets completely filled or empty.

Examples:

Input: X = 4, Y = 3, Z = 2

Output: {(0, 0), (0, 3), (3, 0), (3, 3), (4, 2), (0, 2)}

Explanation:

Step 1:- First we will fill the 4 litre jug completely with water.

Step 2:- Then optimal approach would be to empty water from 4-litre jug into 3-litre (leaving 1L water in 4L jug and 3L completely full). Hence we got 1L water.

Step 3:- Now, Empty water from 3L.

Step 4:- Pour the water from 4L jug into 3L jug Now 4L container is completely empty and 1L water in present in 3L litre jug.

Step 5:- Fill the 4L jug with water completely again.

Step 6:- On transferring water from 4L jug to 3L jug, we will get 2L water in 4L jug which was our required quantity.

Input: X = 3, Y = 5, Z = 4

Output: 6

Explanation:

Step 1:- First we will fill the 5-litres jug to its maximum capacity.

Step 2:- Then optimal approach would be to transfer 3-litres from 5-litres jug to 3-litres jugs.

Step 3:- Now, Empty the 3-litres jug.

Step 4:- Transfer 2L from 5L jug to 3-L jug.

Step 5:- Now, Fill 5-litres jug to its maximum capacity.

Step 6:- On Pouring water from 5L jug to 3L jug until it’s full we will get 4L water in 5-litre jug which was our required quantity.

Running of the algorithm:

We start at an initial state in the queue where both the jugs are empty. We then continue to explore all the possible intermediate states derived from the current jug state using the operations provided.

We also, maintain a visited matrix of states so that we avoid revisiting the same state of jugs again and again.

Cases Jug 1 Jug 2 Is Valid

Case 1 Fill it Empty it ✔

Case 2 Empty it Fill it ✔

Case 3 Fill it Fill it Redundant case

Case 4 Empty it Empty it Already visited (Initial State)

Case 5 Unchanged Fill it ✔

Case 6 Fill it Unchanged ✔

Case 7 Unchanged Empty ✔

Case 8 Empty Unchanged ✔

Case 9 Transfer water from this Transfer water into this ✔

Case 10 Transfer water into this Transfer water from this ✔

From the table above, we can observe that the state where both the jugs are filled is redundant as we won’t be able to continue ahead / do anything with this state in any possible way.

So, we proceed, keeping in mind all the valid state cases (as shown in the table above) and we do a BFS on them.

In the BFS, we first skip the states which was already visited or if the amount of water in either of the jugs exceeded the jug quantity.

If we continue further, then we first mark the current state as visited and check if in this state, if we have obtained the target quantity of water in either of the jugs, we can empty the other jug and return the current state’s entire path.

But, if we have not yet found the target quantity, we then derive the intermediate states from the current state of jugs i.e. we derive the valid cases, mentioned in the table above (go through the code once if you have some confusion).

We keep repeating all the above steps until we have found our target or there are no more states left to proceed with.

**Code:**

import java.util.\*;

class Pair {

int j1, j2;

List<Pair> path;

Pair(int j1, int j2)

{

this.j1 = j1;

this.j2 = j2;

path = new ArrayList<>();

}

Pair(int j1, int j2, List<Pair> \_path)

{

this.j1 = j1;

this.j2 = j2;

path = new ArrayList<>();

path.addAll(\_path);

path.add(new Pair(this.j1, this.j2));

}

}

public class GFG {

public static void main(String[] args)

throws java.lang.Exception

{

int jug1 = 4;

int jug2 = 3;

int target = 2;

getPathIfPossible(jug1, jug2, target);

}

private static void

getPathIfPossible(int jug1, int jug2, int target)

{

boolean[][] visited

= new boolean[jug1 + 1][jug2 + 1];

Queue<Pair> queue = new LinkedList<>();

// Initial State: Both Jugs are empty so,

// initialise j1 j2 as 0 and put it in the path list

Pair initialState = new Pair(0, 0);

initialState.path.add(new Pair(0, 0));

queue.offer(initialState);

while (!queue.isEmpty()) {

Pair curr = queue.poll();

// Skip already visited states and overflowing

// water states

if (curr.j1 > jug1 || curr.j2 > jug2

|| visited[curr.j1][curr.j2])

continue;

// mark current jugs state as visited

visited[curr.j1][curr.j2] = true;

// Check if current state has already reached

// the target amount of water or not

if (curr.j1 == target || curr.j2 == target) {

if (curr.j1 == target) {

// If in our current state, jug1 holds

// the required amount of water, then we

// empty the jug2 and push it into our

// path.

curr.path.add(new Pair(curr.j1, 0));

}

else {

// else, If in our current state, jug2

// holds the required amount of water,

// then we empty the jug1 and push it

// into our path.

curr.path.add(new Pair(0, curr.j2));

}

int n = curr.path.size();

System.out.println(

"Path of states of jugs followed is :");

for (int i = 0; i < n; i++)

System.out.println(

curr.path.get(i).j1 + " , "

+ curr.path.get(i).j2);

return;

}

// If we have not yet found the target, then we

// have three cases left I. Fill the jug and

// Empty the other II. Fill the jug and let the

// other remain untouched III. Empty the jug and

// let the other remain untouched

// IV. Transfer amounts from one jug to another

// Please refer to the table attached above to

// understand the cases that we are taking into

// consideration

// Now,

// I. Fill the jug and Empty the other

queue.offer(new Pair(jug1, 0, curr.path));

queue.offer(new Pair(0, jug2, curr.path));

// II. Fill the jug and let the other remain

// untouched

queue.offer(new Pair(jug1, curr.j2, curr.path));

queue.offer(new Pair(curr.j1, jug2, curr.path));

// III. Empty the jug and let the other remain

// untouched

queue.offer(new Pair(0, curr.j2, curr.path));

queue.offer(new Pair(curr.j1, 0, curr.path));

// IV. Transfer water from one to another until

// one jug becomes empty or until one jug

// becomes full in this process

// Transferring water form jug1 to jug2

int emptyJug = jug2 - curr.j2;

int amountTransferred

= Math.min(curr.j1, emptyJug);

int j2 = curr.j2 + amountTransferred;

int j1 = curr.j1 - amountTransferred;

queue.offer(new Pair(j1, j2, curr.path));

// Tranferring water form jug2 to jug1

emptyJug = jug1 - curr.j1;

amountTransferred = Math.min(curr.j2, emptyJug);

j2 = curr.j2 - amountTransferred;

j1 = curr.j1 + amountTransferred;

queue.offer(new Pair(j1, j2, curr.path));

}

System.out.println("Not Possible to obtain target");

}

}

**Output:**

Path of states of jugs followed is :

0 , 0

0 , 3

3 , 0

3 , 3

4 , 2

0 , 2

**LAB-3**

**AIM:** Write a program using Python to implement DFS.

**THEORY:**

The graph is represented as a dictionary of sets. Each key in the dictionary represents a vertex in the graph, and its corresponding value is a set of vertices that are adjacent to it. The DFS function takes a graph and a starting vertex as input, and returns a set of all the vertices that are visited during the DFS traversal.

**Code:**

# DFS using a stack

def DFS(graph, start):

stack = []

visited = set()

stack.append(start)

while stack:

vertex = stack.pop()

if vertex not in visited:

visited.add(vertex)

stack.extend(graph[vertex] - visited)

return visited

# Example usage

graph = {

'A': set(['B', 'C']),

'B': set(['A', 'D', 'E']),

'C': set(['A', 'F']),

'D': set(['B']),

'E': set(['B', 'F']),

'F': set(['C', 'E'])

}

DFS(graph, 'A') # returns {'E', 'D', 'F', 'A', 'C', 'B'}

**Output:**

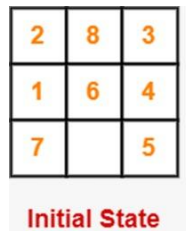
****

**LAB-4**

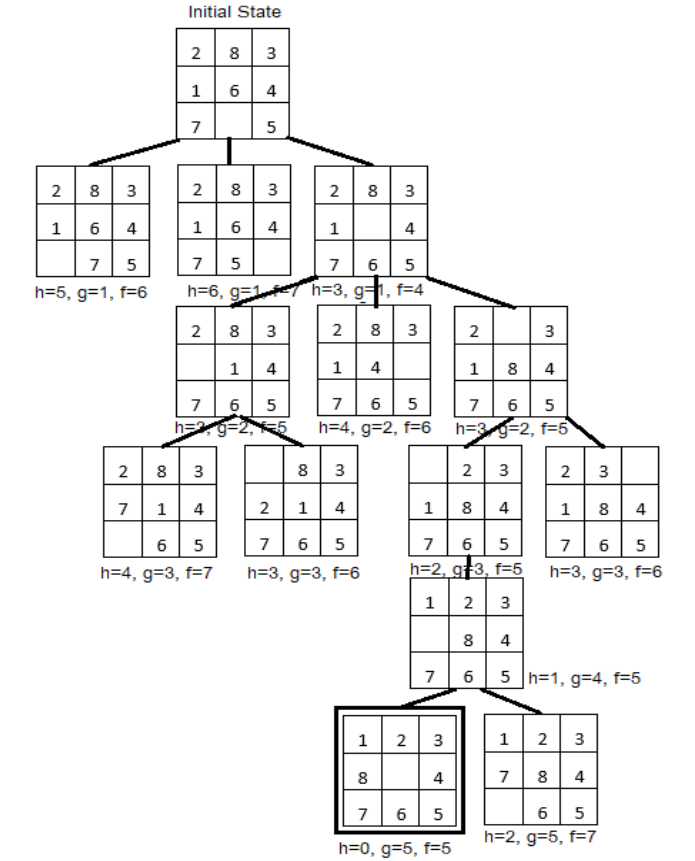
**AIM:** Write a program to implement 8-tile puzzle using A\* algorithm using python.

**THEORY:**

A\* is a pathfinding algorithm that finds the shortest path between two points in a graph. It uses a combination of a heuristic estimate of the remaining distance to the goal, and the actual cost to reach a node. The algorithm prioritizes visiting nodes that are likely to lead to the goal, making it an efficient way to find the optimal solution. The algorithm works by maintaining an open list of nodes to visit, and a closed list of nodes that have already been visited. At each step, the algorithm selects the node with the lowest estimated total cost (the sum of the cost so far and the heuristic estimate) from the open list and adds it to the closed list. The algorithm continues until either the goal node is reached or there are no more nodes on the open list to visit. A\* algorithm can be used to solve the 8-tile puzzle by finding the shortest path from the initial state to the goal state. It uses a combination of the heuristic function and the cost function to determine the next best move. The heuristic function provides an estimate of the minimum number of moves required to reach the goal state, while the cost function is used to keep track of the number of moves made so far. By combining these two functions, A\* can determine the optimal solution.

** **

**State Space Tree:**

****

**Goal State**

**Code:**

class Node:

def \_\_init\_\_(self, data, level, fval):

self.data = data

self.level = level

self.fval = fval

def generate\_child(self):

x, y = self.find(self.data, '\_')

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):

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):

temp = []

for i in root:

t = []

for j in i:

t.append(j)

temp.append(t)

return temp

def find(self, puz, x):

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):

self.n = size

self.open = []

self.closed = []

def accept(self):

puz = []

for i in range(0, self.n):

temp = input().split(" ")

puz.append(temp)

return puz

def f(self, start, goal):

return self.h(start.data, goal)+start.level

def h(self, start, goal):

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):

print("Enter the start state matrix: ")

start = self.accept()

print("Enter the goal state matrix: ")

goal = self.accept()

start = Node(start, 0, 0)

start.fval = self.f(start, goal)

self.open.append(start)

while True:

cur = self.open[0]

print("")

print(" | ")

print(" \\\'/ ")

for i in cur.data:

for j in i:

print(j, end = " ")

print("")

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]

self.open.sort(key=lambda x: x.fval, reverse=False)

puz = Puzzle(3)

puz.process()

**Output:**

**Graphical user interface, text, application, email

Description automatically generated**

**  **

**RESULT:**

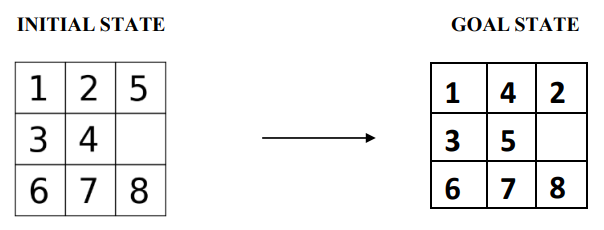
The 8-tile puzzle has been implemented successfully using A\* algorithm in python.

**LAB-5**

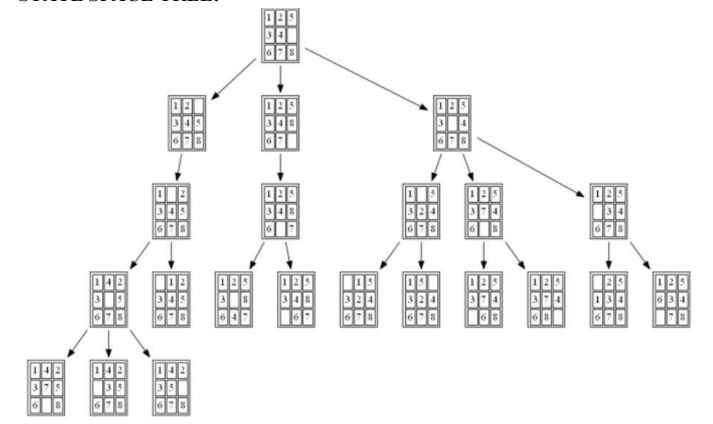
**AIM:** To implement 8-puzzle single player game using Breadth First Search.

**THEORY:**

An instance of the n-puzzle game consists of a board holding n2 -1 distinct movable tiles, plus an empty space. The tiles are numbers from the set 1….. n2 -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 of 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 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.

****

**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=[1,4,2,3,5,0,6,7,8]

num\_of\_instances=0

#constructor to initialize the class members

#works like the default constructor

def \_\_init\_\_(self,state,parent,action): # value passed in it is (self,0,none,none) second pass ([1,2,3,8,0,4,7,6,5],self,0)

self.parent=parent #-> 0 -> self

self.state=state #state is the current state of the board -> set 0 -> [1,2,3,8,0,4,7,6,5]

self.action=action # -> 0 -> 0

#action is the action that is used to generate the current state from the parent state

#TODO: incrementing the number of instance by 1

# We are incrementing the number of instance by 1 to keep track of the number of nodes generated

Puzzle.num\_of\_instances +=1 # -> 1 -> 2

print("Current state: ",Puzzle.num\_of\_instances)

print(self.state)

#function used to display a state of 8-puzzle

def \_\_str\_\_(self):

# state tree such that 0-2 in first row, 3-5 in second row and 6-8 in third row

return str(self.state[0:3])+'\n'+str(self.state[3:6])+'\n'+str(self.state[6:9]) # divided the single list into 3 rows and 3 columns

#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()

print("Goal state found and printed in reverse order: ")

print( ) print(str(Puzzle.goal\_state[0:3])+'\n'+str(Puzzle.goal\_state[3:6])+'\n'+str(Puzzle.goal\_state[6:9]))

#Puzzle.goal\_state

return True

else:

return False

#static method to find the legal action based on the current board position

#static method can be called without creating an object and can be called using the class name

@staticmethod

# i and j are the row and column of the board

def find\_legal\_actions(i,j):

legal\_action = ['U', 'D', 'L', 'R'] # these are the actions performed on the board

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=[] #empty list created

x = self.state.index(0) #index(0) returns the row and column of the board -> 7

# the value 0 in the board is the empty space

# x is the index of the empty space in the board its stores the row and column of the board in a single variable by using the formula x = i\*3 + j

i = int(x / 3) # divided by 3 to get the row of the board -> 2

j = int(x % 3) # mod 3 to get the column of the board -> 1

#example if x=4 then i=1 and j=1

#TODO: call the method to find the legal actions based on i and j values

legal\_actions= Puzzle.find\_legal\_actions(i,j) #(2,1) passed to find\_legal\_action

#legal\_action = ['U', 'L', 'R']

#TODO:Iterate over all legal actions

for action in legal\_actions:

new\_state = self.state.copy() #coping the current state of the board to new\_state

#if the legal action is UP

if action == 'U':

#Swapping between current index of 0 with its up element on the board

# It is going to take the value of the element above the 0 that is 6 in the first step and swap it with the 0

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) # call the class puzzle default constructor with pass (0,none,none) as parameter

print("Initial state:")

print(start\_node) # # in form of 3 lists -> call the \_\_str\_\_ method of puzzle class

print()

print("STATES OF THE BOARD ")

if start\_node.goal\_test(): # check if goal state == initial state

return start\_node.find\_solution()

q = Queue() # created a queue object

#TODO: put start\_node into the Queue

q.put(start\_node) # startnode is the string having 3 lists in it ... so putting that string into the queue

#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()): # currently as the queue is not empty so it will go inside the loop

#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()

#Jump to the generate\_child method of Puzzle class

#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]) # state[i] is at the first run is 0

#Get the time t1 after executing the breadth\_first\_search method

t1=time()-t0

print()

print('BFS:', bfs)

print('space:',Puzzle.num\_of\_instances)

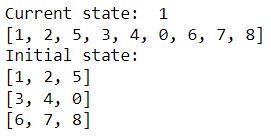
print()

print('time:',t1)

print()

print('------------------------------------------')

**Output:**

**Text, letter

Description automatically generated**

**Text, letter

Description automatically generatedText, letter

Description automatically generated**

**Text, letter

Description automatically generatedText, table

Description automatically generated with medium confidence**

**Table

Description automatically generated with low confidence**

**LAB-6**

**AIM:** Write a program using python to implement tic-tac-toe game using Mini-max algorithm.

**THEORY:**

The algorithm search, recursively, the best move that leads the Max player to win or not lose (draw). It consider 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.

**Code:**

#Import the necessary libraries

from time import time

from queue import Queue

#Creating a class Puzzle

class Puzzle:

#Import the necessary libraries

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 = {}

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("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: # Computer turn

\_,block\_choice = getBestMove(game\_state, players[current\_player\_idx])

#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)

print("AI plays move: " + str(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":

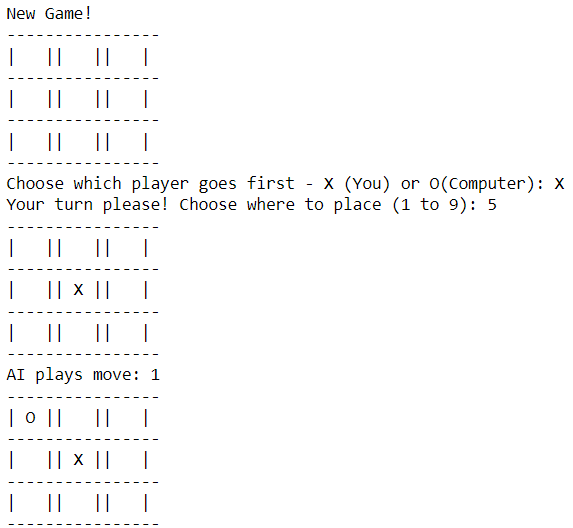
print("Draw!")

play\_again = input('Wanna try again?(Y/N) : ')

if play\_again == 'N':

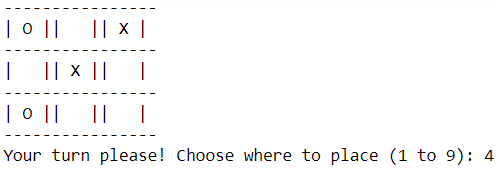
print('Thank you for playing Tic-Tac-Toe Game!!!!!!!')

**Output:**



Graphical user interface, text

Description automatically generated with medium confidence



Table

Description automatically generated

Graphical user interface

Description automatically generated with medium confidence



**LAB-7**

**AIM:** Write a program using python to implement tic-tac-toe game using alpha-beta pruning algorithm.

**Code:**

#Import the necessary libraries

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("AI 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":

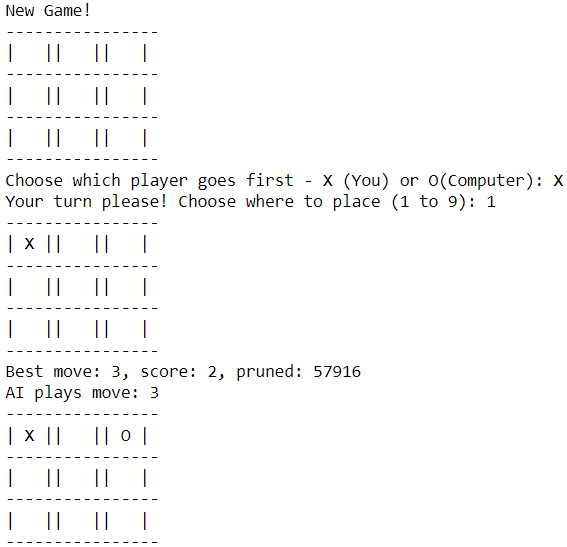
print("Draw!")

play\_again = input('Wanna try again?(Y/N) : ')

if play\_again == 'N':

print('Thank you for playing Tic-Tac-Toe Game!!!!!!!')

**Output:**



Table

Description automatically generated

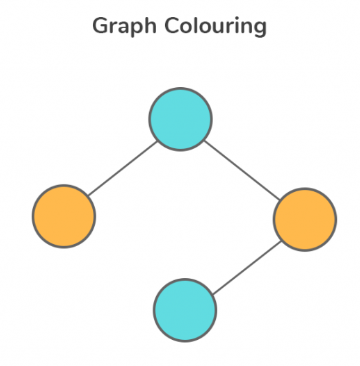
**LAB-8**

**AIM:** Write a python program to implement graph coloring problem.

**THEORY:**

Graph coloring problem involves assigning colors to certain elements of a graph subject to certain restrictions and constraints. In other words, the process of assigning colors to the vertices such that no two adjacent vertexes have the same color is caller Graph Coloring.

This is also known as vertex coloring.

****

**Code:**

class Graph:

def \_\_init\_\_(self, vertices):

self.V = vertices

self.graph = [[0 for column in range(vertices)] for row in range(vertices)]

def isSafe(self, v, color, c):

for i in range(self.V):

if self.graph[v][i] == 1 and color[i] == c:

return False

return True

def graphColorUtil(self, m, color, v):

if v == self.V:

return True

for c in range(1, m+1):

if self.isSafe(v, color, c) == True:

color[v] = c

if self.graphColorUtil(m, color, v+1) == True:

return True

color[v] = 0

def graphColoring(self, m):

color = [0] \* self.V

if self.graphColorUtil(m, color, 0) == False:

return False

print("Graph can be colored with at most", m, "colors.")

print("The coloring scheme is as follows:")

for i in range(self.V):

print("Vertex", i, "is colored with", color[i])

g = Graph(4)

g.graph = [[0, 1, 1, 1],

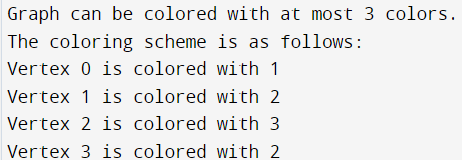
[1, 0, 1, 0],

[1, 1, 0, 1],

[1, 0, 1, 0]]

g.graphColoring(3)

**Output:**

****

**LAB-9**

**AIM:** Write a python script for a given cryptarithmetic problem (APPLE+LEMON= BANANA).

**THEORY:**

A = {5, 6, 7, 8}  
Proof:  
E + N = A , remind E > N  
0 is impossible because E(0) + N(0) = A(0)  
1 is impossible too because E(1) + N(0) = A(1) and main reason is digit 1 is for B  
2 is impossible too because E(2) + N(0) = A(2)  
A cannot be 3, 4 and 9 too.  
So A = {5, 6, 7 ,8}, E + N = A , remind E > N  
if A = 5 means E + N can be either 0 + 5, 5 + 0, 4 + 1, 1 + 4, 2 + 3 or 3 + 2.  
E(5) + N(0) = A(5) impossible!  
4 + 1 impossible too because digit1 is for B  
so left 3 + 2  
E(3) + N(2) = A(5)  
L(9) + O(?) = N(2) so ? = 3 therefore 3 is repeated in E and O so 3 + 2 is wrong.  
Try and error for the following:  
A = {5, 6, 7 ,8}  
A = E + N  
5 = 3 +2  
6 = 4 + 2  
7 = 5 + 2  
7 = 4 + 3  
8 = 5 + 3  
8 = 6 + 2  
Therefore, APPLE + LEMON = BANANA  
 67794 + 94832 = 162626

**Code:**

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 WORD1: ').upper()

word2 = input('Enter WORD2: ').upper()

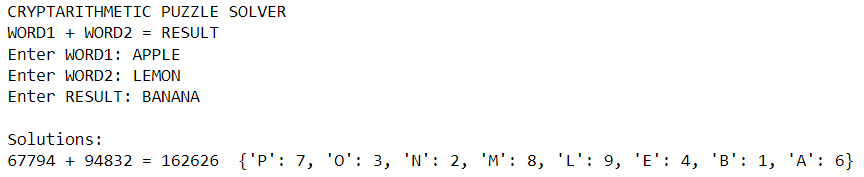
result = input('Enter RESULT: ').upper()

if not word1.isalpha() or not word2.isalpha() or not result.isalpha():

raise TypeError('Inputs should ony consists of alphabets.')

solve(word1, word2, result)

**Output:**



**LAB-10**

**AIM:** Write a Python script to demonstrate the following:

* Tokenization of Word and Sentence
* Stopword Removal
* Stemming
* Lemmatization
* POS Tagging (Parsing)

**THEORY:**

* **Tokenization:** Tokenization is the first step in any NLP pipeline. It has an important effect on the rest of your pipeline. A tokenizer breaks unstructured data and natural language text into chunks of information that can be considered as discrete elements. The token occurrences in a document can be used directly as a vector representing that document. Tokenization can separate sentences, words, characters, or subwords. When we split the text into sentences, we call it sentence tokenization. For words, we call it word tokenization.
* **Stopword Removal:** Stop words are common words like ‘the’, ‘and’ , ‘I’, etc. that are very frequent in text, and so don’t convey insights into the specific topic of a document. We can remove these stop words from the text in a given corpus to clean up the data, and identify words that are more rare and potentially more relevant to what we’re interested in. Stop words can be filtered from the text to be processed. There is no universal list of stop words in nlp research, however the nltk module contains a list of stop words.
* **Stemming:** Stemming is a natural language processing technique that lowers inflection in words to their root forms, hence aiding in the preprocessing of text, words, and documents for text normalization.
* **Lemmatization:** Lemmatization is one of the text normalization techniques that reduce words to their base forms. However, lemmatization is more context-sensitive and linguistically informed, lemmatization uses a dictionary or a corpus to find the lemma or the canonical form of each word. Lemmatization also takes into account the part-of-speech tag of each word, which can affect its meaning and lemma. For instance, the word "saw" can be lemmatized differently depending on whether it is a noun or a verb in a sentence.
* **POS tagging :** POS Tagging (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 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. Some NLTK POS tagging examples are: CC, CD, EX, JJ, MD, NNP, PDT, PRP, TO, etc.

**CODE:**

**Tokenization:**

import nltk

#nltk.download('punkt')

from nltk.tokenize import word\_tokenize, sent\_tokenize

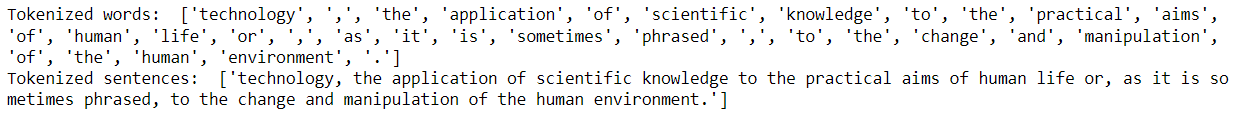
text = "technology, the application of scientific knowledge to the practical aims of human life or, as it is sometimes phrased, to the change and manipulation of the human environment."

words = word\_tokenize(text)

sentences = sent\_tokenize(text)

print("Tokenized words: ", words)

print("Tokenized sentences: ", sentences)

****

**Stopword Removal:**

#nltk.download('stopwords')

from nltk.corpus import stopwords

stop\_words = set(stopwords.words('english'))

filtered\_words = [word for word in words if not word.lower() in stop\_words]

print("Filtered words: ", filtered\_words)



**Stemming:**

from nltk.stem import PorterStemmer, SnowballStemmer, LancasterStemmer

porter = PorterStemmer()

stemmed\_words\_porter = [porter.stem(word) for word in words]

print("Porter stemmed words: ", stemmed\_words\_porter)

snowball = SnowballStemmer('english')

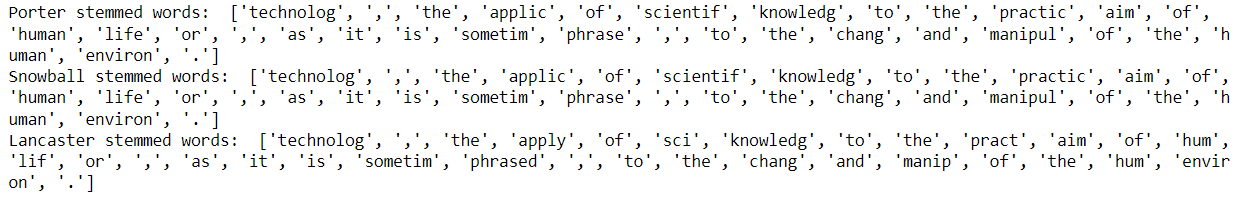
stemmed\_words\_snowball = [snowball.stem(word) for word in words]

print("Snowball stemmed words: ", stemmed\_words\_snowball)

lancaster = LancasterStemmer()

stemmed\_words\_lancaster = [lancaster.stem(word) for word in words]

print("Lancaster stemmed words: ", stemmed\_words\_lancaster)



**Lemmatization:**

#nltk.download('wordnet')

from nltk.stem import WordNetLemmatizer

lemmatizer = WordNetLemmatizer()

lemmatized\_words\_wordnet = [lemmatizer.lemmatize(word) for word in words]

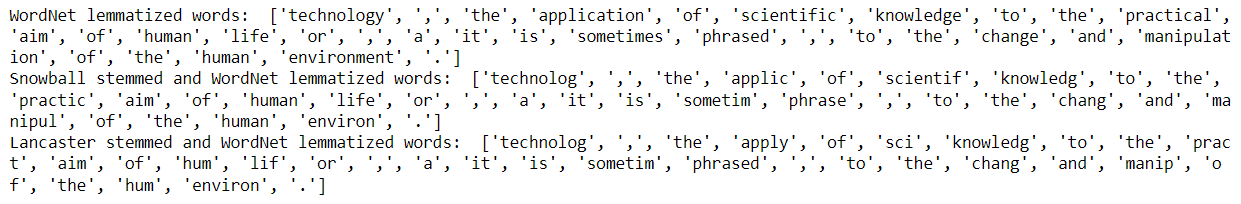
print("WordNet lemmatized words: ", lemmatized\_words\_wordnet)

lemmatized\_words\_snowball = [lemmatizer.lemmatize(word) for word in stemmed\_words\_snowball]

print("Snowball stemmed and WordNet lemmatized words: ", lemmatized\_words\_snowball)

lemmatized\_words\_lancaster = [lemmatizer.lemmatize(word) for word in stemmed\_words\_lancaster]

print("Lancaster stemmed and WordNet lemmatized words: ", lemmatized\_words\_lancaster)



**POS Tagging ( Parsing ):**

#import nltk

from nltk.tokenize import word\_tokenize

sentence = "I saw the man with the telescope"

grammar = nltk.CFG.fromstring("""

S -> NP VP

PP -> P NP

NP -> DT N | DT JJ N | PRP | NP PP

VP -> V | V NP | VP PP | V S

DT -> "the"

N -> "man" | "telescope"

JJ -> "with"

V -> "saw"

P -> "with"

PRP -> "I"

""")

words = word\_tokenize(sentence)

parser = nltk.ChartParser(grammar)

trees = parser.parse(words)

for tree in trees:

tree.pretty\_print()

