SRM Institute of Science & Technology, Delhi NCR Campus

Department of Computer Science & Engineering

Artificial Intelligence (18CSC307L)

Lab File

SRM Institute of Science & Technology, Delhi NCR Campus

Department of Computer Science & Engineering

LABORATORY FILE

Faculty Name	: Mr. Dharmendra	**Department**	: CSE
Course Name	: AI Lab	**Course Code**	: 18CSC307L
Year/Sem	: 3rd/6th	**Academic Year**	: 2022-23

Student Name	PALAK KHURANA
Registration No.	RA1911003030437
Section	B.TECH CSE - I

SRM Institute of Science & Technology, Ghaziabad
Department of Computer Science & Engineering

INDEX

Experiment No.	Experiment Name	Date of Conduction	Date of Submission	Faculty Signature
1	IMPLEMENTATION OF N-QUEEN PROBLEM.			
2	IMPLEMENTATION OF RIVER CROSSING PROBLEM.			
3	IMPLEMENTATION OF WATER JUG PROBLEM.			
4	IMPLEMENTATION OF DEPTH FIRST SEARCH ALGORITHM AND BREADTH FIRST SEARCH.			
5	IMPLEMENTATION OF MONKEY BANANA PROBLEM.			
6	IMPLEMENTATION OF A* ALGORITHM.			
7	IMPLEMENTATION OF TOWER OF HANOI PROBLEM.			
8	IMPLEMENTATION OF AGENT PROGRAMS FOR REAL-WORLD PROBLEMS (VACCUM CLEANER).			
9	IMPLEMENTATION OF CONSTRAINTS SATISFACTION PROBLEM (TRUCK PROBLEM).			
10	IMPLEMENTATION OF MINIMAX ALGORITHM.			
11	IMPLEMENTATION OF PROPOSITIONAL LOGIC IN REAL WORLD PROBLEMS.			
12	IMPLEMENTATION OF UNIFICATION AND RESOLUTION OF REAL-WORLD PROBLEMS.			

LIST OF EXPERIMENTS

Expt. No.	Title of experiment
1.	IMPLEMENTATION OF N-QUEEN PROBLEM.
2.	IMPLEMENTATION OF RIVER CROSSING PROBLEM.
3.	IMPLEMENTATION OF WATER JUG PROBLEM.
4.	IMPLEMENTATION OF DEPTH FIRST SEARCH ALGORITHM AND BREADTH FIRST SEARCH.
5.	IMPLEMENTATION OF MONKEY BANANA PROBLEM.
6.	IMPLEMENTATION OF A* ALGORITHM.
7.	IMPLEMENTATION OF TOWER OF HANOI PROBLEM.
8.	IMPLEMENTATION OF AGENT PROGRAMS FOR REAL-WORLD PROBLEMS (VACCUM CLEANER).
9.	IMPLEMENTATION OF CONSTRAINTS SATISFACTION PROBLEM (TRUCK PROBLEM).
10.	IMPLEMENTATION OF MINIMAX ALGORITHM.
11.	IMPLEMENTATION OF PROPOSITIONAL LOGIC IN REAL WORLD PROBLEMS.
12.	IMPLEMENTATION OF UNIFICATION AND RESOLUTION OF REAL-WORLD PROBLEMS.

GUIDELINES FOR LABORTORY RECORD PREPARATION:

While preparing the lab records, the student is required to adhere to the following guidelines:

Contents to be included in Lab Records:

1. Cover page
2. Index
3. Experiments-

Aim
Algorithm
Source code
Input-Output

Experiment No. 1

Aim : IMPLEMENTATION OF N-QUEEN PROBLEM.

Algorithm :

1) Start in the leftmost column
2) If all queens are placed
   return true
3) Try all rows in the current column. Do following
   for every tried row.
   a) If the queen can be placed safely in this row
      then mark this [row, column] as part of the
      solution and recursively check if placing
      queen here leads to a solution.
   b) If placing queen in [row, column] leads to a
      solution then return true.
   c) If placing queen doesn't lead to a solution
      then unmark this [row, column] (Backtrack)
      and go to step (a) to try other rows.
3) If all rows have been tried and nothing worked,
   return false to trigger backtracking.

Source Code :

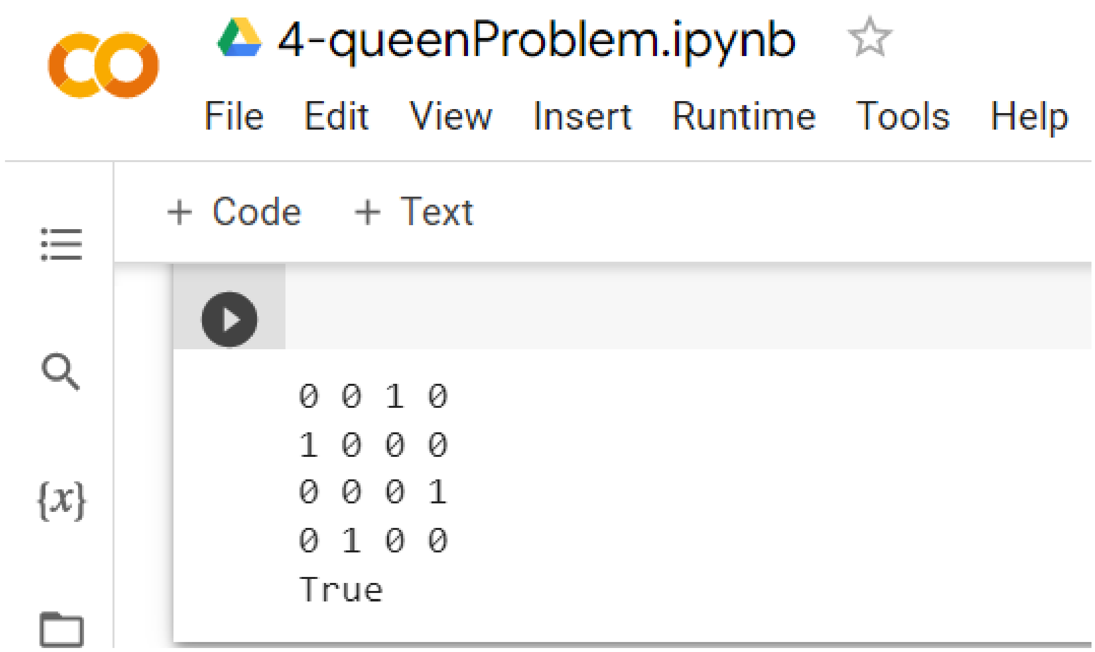
```
global N
N = 4

def printSolution(board):
    for i in range(N):
        for j in range(N):
            print (board[i][j], end = " ")
        print()

def isSafe(board, row, col):

    for i in range(col): # Check this row on left side
        if board[row][i] == 1:
            return False
```

```
for i, j in zip(range(row, -1, -1),        # Check upper diagonal on left side
                range(col, -1, -1)):
```

```
        if board[i][j] == 1:
            return False

    for i, j in zip(range(row, N, 1),       # Check lower diagonal on left side
                    range(col, -1, -1)):
        if board[i][j] == 1:
            return False

    return True

def solveNQUtil(board, col):

    if col >= N:
        return True

    for i in range(N):

        if isSafe(board, i, col):

            # Place this queen in board[i][col]
            board[i][col] = 1

            # recur to place rest of the queens
            if solveNQUtil(board, col + 1) == True:
                return True

            board[i][col] = 0

    return False

def solveNQ():
    board = [ [0, 0, 0, 0],
              [0, 0, 0, 0],
              [0, 0, 0, 0],
              [0, 0, 0, 0] ]

    if solveNQUtil(board, 0) == False:
        print ("Solution does not exist")
        return False

    printSolution(board)
    return True
```

solveNQ()

Output:



Experiment No. 2

Aim : IMPLEMENTATION OF RIVER CROSSING PROBLEM.

Algorithm :
Take the GOAT first.
Leave the LION with the CABBAGE.
Row back and pick up the LION.
Take the LION across and bring back the GOAT.
Leave the GOAT and take the CABBAGE cross.
Leave the LION with the CABBAGE.
Row back and bring the GOAT.
Everyone is now across safely!

Source code :

```
x=['M', 'L', 'G' , 'C']
y=[]
print("Before Process")
print("Element in the Left Side Bank ", x)
print("Element in the Right Side Bank ", y )
while True:
  print(x[1]," ", x[2]," ", x[3], " Select any one from the list")
  i=input("Enter the item :")
  i=i.upper()
  if x[1]==i and x[2]=='G' and x[3]=='C':
    print("Goat will eat cabbage :")
    break
  elif x[2]==i and x[3]!='C':
    y.append(x[2])
    if len(y)==2 and y[0]=='G':
      x[2]=y[0]
      y[0]=y[1]
      y.pop()
  elif x[1]==i and x[2]=='G':
   y.append(x[1])
   x[1]=x[2]
   x[2]=''
  elif x[1]==i and x[2]=='C':
   y.append(x[1])
   x[1]=x[2]
```

```
   x[2]=''
   if len(y)==2 and y[0]=='G':
     x[2]=y[0]
     y[0]=y[1]
     y.pop()
  elif x[1]==i and x[2]!='C' and x[2]!='G':
   y.append(x[1])
   y.append('M')
   x[1]=''
   x=[]
   print("Goal is reached ")
   break
  if x[2]==i  and x[3]=='C':
   y.append(x[2])
   x[2]=x[3]
   x[3]=''
  if x[3]==i:
   print("Lion will eat Goat ")
   break
print("After Process")
print("Element in the Left Side Bank ", x)
print("Element in the Right Side Bank ", y)
```

Output :

RiverCrossing.ipynb

File Edit View Insert Runtime Tools Help Last edited on March 10

+ Code + Text

```
Before Process
Element in the Left Side Bank  ['M', 'L', 'G', 'C']
Element in the Right Side Bank  []
L   G   C  Select any one from the list
Enter the item :g
L   C     Select any one from the list
Enter the item :c
L   G     Select any one from the list
Enter the item :l
G        Select any one from the list
Enter the item :l
G        Select any one from the list
Enter the item :c
G        Select any one from the list
Enter the item :g
Goal is reached
After Process
Element in the Left Side Bank  []
Element in the Right Side Bank  ['C', 'L', 'G', 'M']
```

Experiment No. 3

Aim : IMPLEMENTATION OF WATER JUG PROBLEM.

Algorithm:

The operations you can perform are:

1. Empty a Jug, (X, Y)->(0, Y) Empty Jug 1
2. Fill a Jug, (0, 0)->(X, 0) Fill Jug 1
3. Pour water from one jug to the other until one of the jugs is either empty or full, (X, Y) -> (X-d, Y+d)

Source code :

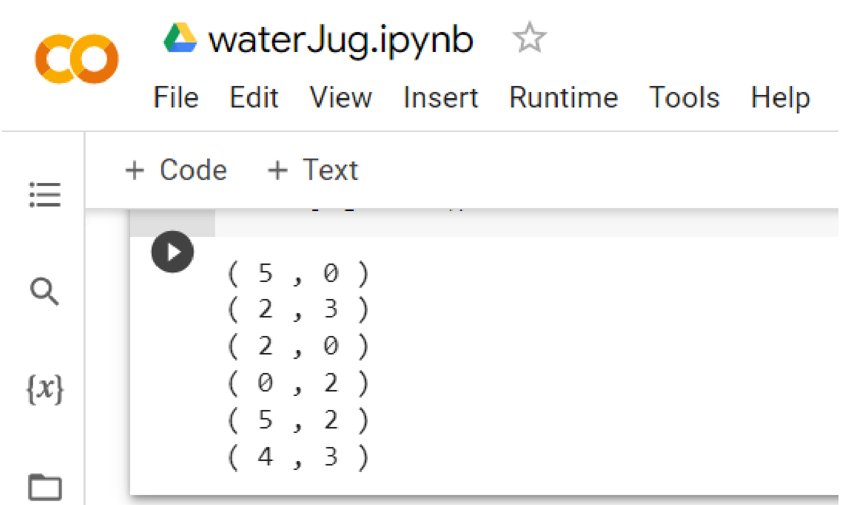
BY MINIMUM STEPS -

```
class Waterjug:

    def__init__(self,am,bm,a,b,g):
        self.a_max = am;
        self.b_max = bm;
        self.a = a;
        self.b = b;
        self.goal = g;

    def fillA(self):
        self.a = self.a_max;
        print ('(', self.a, ',',self.b, ')')

    def fillB(self):
        self.b = self.b_max;
        print ('(', self.a, ',', self.b, ')')

    def emptyA(self):
        self.a = 0;
        print ('(', self.a, ',', self.b, ')')

    def emptyB(self):
        self.b = 0;
        print ('(', self.a, ',', self.b, ')')
```

```
def transferAtoB(self):
    while (True):
```

```
            self.a = self.a - 1
            self.b = self.b + 1
            if (self.a == 0 or self.b == self.b_max):
                break
        print ('(', self.a, ',', self.b, ')')

    def main(self):
        while (True):

            if (self.a == self.goal or self.b == self.goal):
                 break
            if (self.a == 0):
             self.fillA()
            elif (self.a > 0 and self.b != self.b_max):
                self.transferAtoB()
            elif (self.a > 0 and self.b == self.b_max):
                self.emptyB()


waterjug=Waterjug(5,3,0,0,4);
waterjug.main();
```

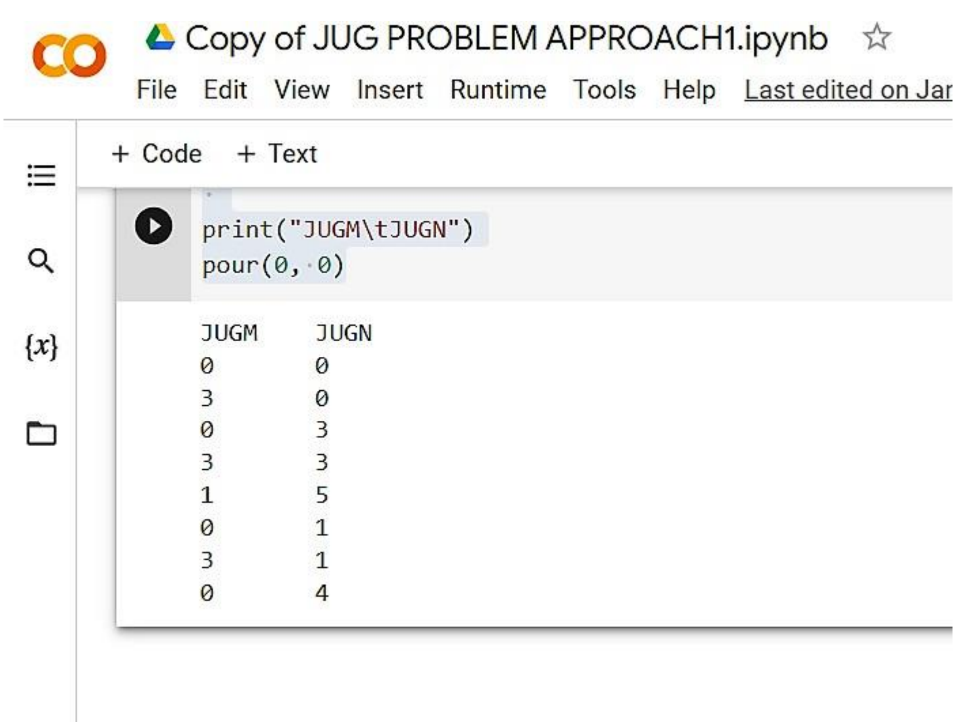
Output :



BY MAXIMUM STEPS –

```
def pour(jugM, jugN):
    A, B, fill = 3, 5, 4
    print("%d\t%d" % (jugM,jugN))
    if jugN is fill:
        return
    elif jugN is B:
        pour(0, jugM)
    elif jugM != 0 and jugN is 0:
        pour(0, jugM)
    elif jugM is fill:
        pour(jugM, 0)
    elif jugM < A:
        pour(A, jugN)
    elif jugM < (B-jugN):
        pour(0, (jugM+jugN))
    else:
        pour(jugM-(B-jugN), (B-jugN)+jugN)
 print("JUGM\tJUGN")
pour(0, 0)
```

Output :



Experiment No. 4

Aim : IMPLEMENTATION OF DEPTH FIRST SEARCH ALGORITHM AND BREADTH FIRST SEARCH

Algorithm:

BFS :

Consider G as a graph which we are going to traverse using the BFS algorithm.

Let S be the root/starting node of the graph.

- **Step 1:** Start with node S and enqueue it to the queue.
- **Step 2:** Repeat the following steps for all the nodes in the graph.
- **Step 3:** Dequeue S and process it.
- **Step 4:** Enqueue all the adjacent nodes of S and process them.
- [END OF LOOP]
- **Step 6:** EXIT

DFS:

- **Step 1:** Insert the root node or starting node of a tree or a graph in the stack.
- **Step 2:** Pop the top item from the stack and add it to the visited list.
- **Step 3:** Find all the adjacent nodes of the node marked visited and add the ones that are not yet visited, to the stack.
- **Step 4**: Repeat steps 2 and 3 until the stack is empty.

Source code:

BFS:

```
from queue import Queue

graph = {
    0 : [1,2,3],
    1 : [0,4],
    2 : [0,4],
    3 : [0,4],
    4 : [1,2,3]
    }
print("The adjacency List representing the graph is:")
print(graph)


def bfs(graph, source):
    Q = Queue()
    visited_vertices = set()
    Q.put(source)
    visited_vertices.update({1})
    while not Q.empty():
        vertex = Q.get()
        print(vertex, end=" ")
        for u in graph[vertex]:
            if u not in visited_vertices:
                Q.put(u)
                visited_vertices.update({u})

print("BFS traversal of graph with source 1 is:")
bfs(graph, 1)
```

Output :

bfs.ipynb

File Edit View Insert Runtime Tools Help Last edited on February 9

+ Code + Text

```
The adjacency List representing the graph is:
{0: [1, 2, 3], 1: [0, 4], 2: [0, 4], 3: [0, 4], 4: [1, 2, 3]}
BFS traversal of graph with source 1 is:
1  0  4  2  3
```

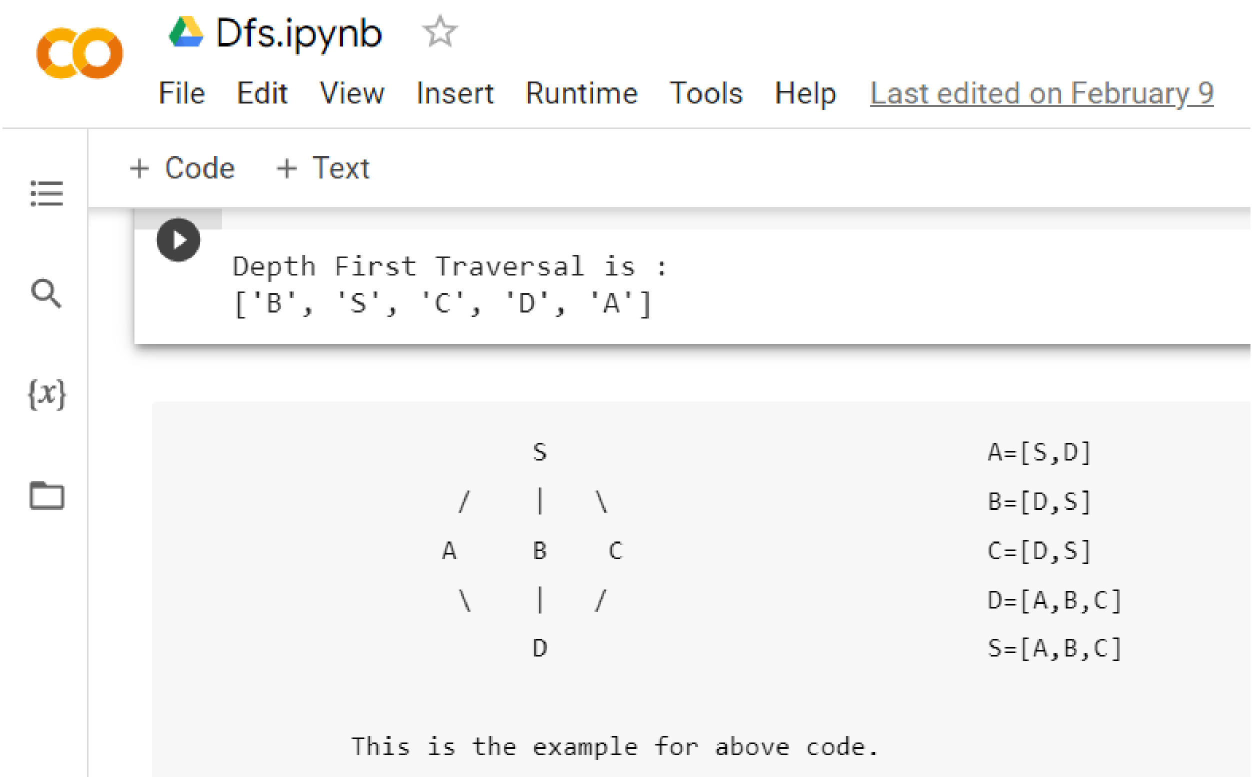
```
          0                    0=[1,2,3]
      /   |   \                1=[0,4]
     1    2    3               2=[0,4]
      \   |   /                3=[0,4]
          4                    4=[1,2,3]
  This is the example for above code.
```

DFS :

```
class Stack:

    def__init__(self):
        self.list = []

    def push(self, item):
        self.list.append(item)

    def pop(self):
        return self.list.pop()

    def top(self):
        return self.list[-1]

    def is_empty(self):
        return len(self.list) == 0

def depth_first_search(graph, start):
    stack = Stack()
    stack.push(start)
    path = []

    while not stack.is_empty():
        vertex = stack.pop()
        if vertex in path:
            continue
        path.append(vertex)
        for neighbor in graph[vertex]:
            stack.push(neighbor)

    return path

def main():
    adjacency_matrix = {
        'S' : ['A','B','C'],
    'A' : ['S','D'],
    'B' : ['D','S'],
    'C' : ['D','S'],
```

```
    'D' : ['A','B','C']
    }
    dfs_path = depth_first_search(adjacency_matrix, 'B')
    print("Depth First Traversal is : ")
    print(dfs_path)

if__name__== '__main__':
    main()
```

Output :



Experiment No. 5

Aim: IMPLEMENTATION OF MONKEY BANANA PROBLEM.

Algorithm :

If the monkey is clever enough, he can come to the block, drag the block to the center, climb on it, and get the banana. Below are few observations in this case −

- Monkey can reach the block, if both of them are at the same level. From the above image, we can see that both the monkey and the block are on the floor.
- If the block position is not at the center, then monkey can drag it to the center.
- If monkey and the block both are on the floor, and block is at the center, then the monkey can climb up on the block. So the vertical position of the monkey will be changed.
- When the monkey is on the block, and block is at the center, then the monkey can get the bananas.

Source code :

```
import pdb
from random import shuffle

class State:
    def __init__(self):
        self.properties = self.generateStates()

    def __eq__(self, other):
        return self.properties == other.properties

    def generateStates(self):
        properties = set()
        posit = [0, 1, 2]
        elements = ["Monkey", "Banana", "Box"]

        for el in elements:
            for pos in posit:
                properties.add(el + "At" + str(pos))
```

```
        properties.add('MonkeyLevelUp')
        properties.add('MonkeyLevelDown')
        properties.add('haveBanana')
        return properties

    def setProperties(self, properties):
        self.properties = properties

    def __str__(self):
        return str(self.properties)
    def __repr__(self):
        return str(self.properties)



class Operation:
    def __init__(self, name):
        self.name = name
        self.PC = set()
        self.A = set()
        self.E = set()

    def __str__(self):
        return "{}".format(self.name)

    def __eq__(self, other):
        return self.name == other.name

    def __hash__(self):
        return hash(self.name)

    def canApply(self, state):

        return self.PC.intersection(state.properties) == self.PC

    def apply(self, state):

        print("Apply {} to state".format(self.name))
        s = State()
        properties = set()
        if self.canApply(state):
```

```
            properties = state.properties.union(self.A)
            properties = properties.difference(self.E)
        else:
            print("Cannot apply {} to state {}".format(self.name, state))
        s.setProperties(properties)
        return s

    def show(self):
        """
        Show actions verbose
        :return:
        """
        print(self.name)
        print("PC: {}".format(self.PC))
        print("A: {}".format(self.A))
        print("E: {}".format(self.E))

class Move(Operation):
    def __init__(self, object, x, y):
        self.name = "Move{}({},{})".format(object, x, y)
        self.object = object
        self.x = x
        self.y = y
        self.PC = self.getPreconditions()
        self.A = self.getA()
        self.E = self.getE()

    def getPreconditions(self):
        p = set()
        p.add(self.object + "At" + str(self.x))
        if self.object == 'Monkey':
            p.add('MonkeyLevelDown')
        return p

    def __repr__(self):
        return Operation.__str__(self)

    def getA(self):
        p = set()
        p.add(self.object + "At" + str(self.y))
        return p
```

```
    def getE(self):
        p = set()
        p.add(self.object + "At" + str(self.x))
        return p

class PushBox(Operation):
    def __init__(self, x, y):
        self.name = "PushBox({},{})".format(x, y)
        self.x = x
        self.y = y
        self.PC = self.getPreconditions()
        self.A = self.getA()
        self.E = self.getE()

    def __repr__(self):
        return Operation.__str__(self)

    def getPreconditions(self):
        p = set()
        p.add("BoxAt" + str(self.x))
        p.add("MonkeyAt" + str(self.x))
        p.add("MonkeyLevelDown")
        return p

    def getA(self):
        p = set()
        p.add("BoxAt" + str(self.y))
        p.add("MonkeyAt" + str(self.y))
        return p

    def getE(self):
        p = set()
        p.add("BoxAt" + str(self.x))
        p.add("MonkeyAt" + str(self.x))
        return p

class ClimbBox(Operation):
    def __init__(self, x, updown):
        self.name = "ClimbBox{}(at {})".format(updown, x)
        self.x = x
```

```
        self.updown = updown
        self.PC = self.getPreconditions()
        self.A = self.getA()
        self.E = self.getE()

    def __repr__(self):
        return Operation.__str__(self)

    def getPreconditions(self):
        p = set()
        p.add("BoxAt" + str(self.x))
        p.add("MonkeyAt" + str(self.x))
        if self.updown == 'Up':
            p.add("MonkeyLevelDown")
        else:
            p.add("MonkeyLevelUpAt{}".format(self.x))
        return p

    def getA(self):
        p = set()
        if self.updown == 'Up':
            p.add("MonkeyLevelUpAt{}".format(self.x))
        else:
            p.add("MonkeyLevelDown".format(self.x))
            p.add("MonkeyAt{}".format(self.x))
        return p

    def getE(self):
        p = set()
        if self.updown == 'Up':
            p.add("MonkeyLevelDown")
        else:
            p.add("MonkeyLevelUpAt{}".format(self.x))
        return p

class HaveBanana(Operation):
    def __init__(self, x):
        self.name = "GetBanana(at {})".format(x)
        self.x = x
        self.PC = self.getPreconditions()
        self.A = self.getA()
```

```
        self.E = self.getE()

    def __repr__(self):
        return Operation.__str__(self)

    def getPreconditions(self):
        p = set()
        p.add("BoxAt" + str(self.x))
        p.add("MonkeyAt" + str(self.x))
        p.add("BananaAt" + str(self.x))
        p.add("MonkeyLevelUpAt{}".format(self.x))
        return p

    def getA(self):
        p = set()
        p.add("haveBanana")
        return p

    def getE(self):
        p = set()
        return p

def generateOperations(initial):

    operations = list()
    s = State()
    properties = s.generateStates()

    elements = ['Banana', 'Monkey', 'Box']
    movement = ['Up', 'Down']
    positions = [0, 1, 2]

    for x in positions:
        for y in positions:
            if x!=y:
                operations.append(Move('Monkey', x, y))
                operations.append(PushBox(x,y))


        for direction in movement:
            operations.append(ClimbBox(x, direction))
```

```
    for item in list(initial.properties):
        if 'Banana' in item:
            operations.append(HaveBanana(item.split('At')[1]))
    return operations


def selectOperation(cs, gs):

    return list(set(filter(lambda x: len(x.A.intersection(gs)) > 0, generateOperations(cs))))

def isFinalState(state, goal):

    return state.properties.intersection(goal.properties) == goal.properties

def STRIPSiter(state, goal):

    plan = []
    stack = list(goal.properties)
    while len(stack) > 0:
        target = stack[0]

        if type(target) in [ClimbBox, HaveBanana, PushBox, Move]:
            state = target.apply(state)
            plan.append(target)
            stack.remove(target)

        elif target in state.properties:
            stack.remove(target)
        elif type(target) == str:

            operations = list(selectOperation(state, {target}))

            if len(operations) > 0:

                if 'MonkeyAt' in target:
                    operations = list(filter(lambda x: type(x) == Move, operations))


                shuffle(operations)
```

```
            operation = operations[0]
            stack = [operation] + stack

            stack = list(operation.PC) + stack

        else:
            stack = []
            plan = False
    return plan


initial = State()

initial.setProperties({"MonkeyAt0", "BananaAt1", "BoxAt2", "MonkeyLevelDown"})

goal = State()
goal.setProperties({'haveBanana'})

print("Initially:\n{}\n".format(initial))
print("Goal state:\n{}\n".format(goal))
print("STRIPS START\n=============")
plan = STRIPSiter(initial, goal)
if plan:
    print("\nFinally the steps are: \n{}".format(plan))
else:
    print("Could not find a solution")
```

Output :

Finally the steps are:

[MoveMonkey(0,1), MoveMonkey(1,0), MoveMonkey(0,1), MoveMonkey(1,2), MoveMonkey(2,1), MoveMonkey(1,0), MoveMonkey(0,2), PushBox(2,1), PushBox(1,0), PushBox(0,1), ClimbBoxUp(at 1), GetBanana(at 1)]

Experiment No. 6

Aim : IMPLEMENTATION OF A* ALGORITHM.

Algorithm :
pseudocode –
// A* (star) Pathfinding
// Initialize both open and closed list,let the openList equal empty list of nodes,let the closedList equal empty list of nodes
// Add the start node,put the startNode on the openList (leave it's f at zero)
// Loop until you find the end,while the openList is not empty
// Get the current node,let the currentNode equal the node with the least f value, remove the currentNode from the openList, add the currentNode to the closedList
// Found the goal,if currentNode is the goal You've found the end! Backtrack to get path
// Generate children,let the children of the currentNode equal the adjacent nodesfor each child in the children
// Child is on the closedList,if child is in the closedList,continue to beginning of for loop
// Create the f, g, and h values child.g = currentNode.g + distance between child and current child.h = distance from child to end child.f = child.g + child.h
// Child is already in openList,if child.position is in the openList's nodes positions, if the child.g is higher than the openList node's g , continue to beginning of for loop
// Add the child to the openList, add the child to the openList

Source code :

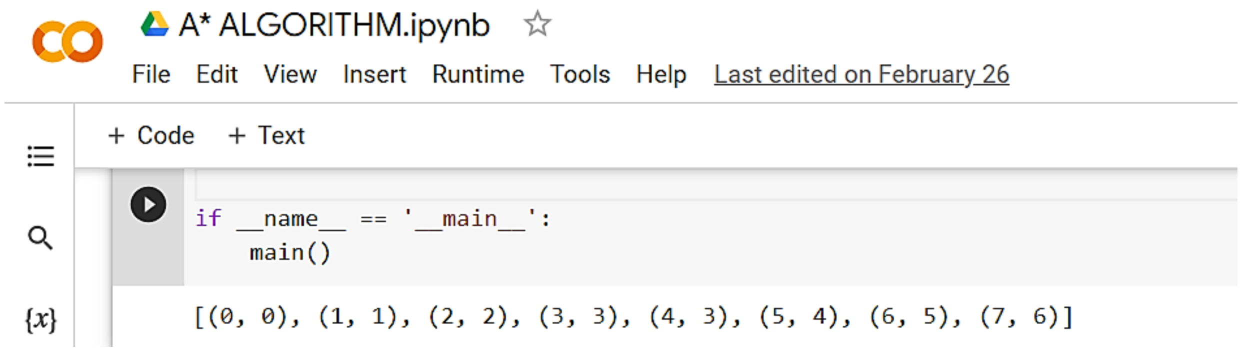
```
class Node():
    """A node class for A* Pathfinding"""

    def__init__(self, parent=None, position=None):
        self.parent = parent
        self.position = position

        self.g = 0
        self.h = 0
        self.f = 0
```

```
    def__eq__(self, other):
        return self.position == other.position

def astar(maze, start, end):
    """Returns a list of tuples as a path from the given start to the given end in the gi
ven maze"""

    # Create start and end node
    start_node = Node(None, start)
    start_node.g = start_node.h = start_node.f = 0
    end_node = Node(None, end)
    end_node.g = end_node.h = end_node.f = 0

    # Initialize both open and closed list
    open_list = []
    closed_list = []

    # Add the start node
    open_list.append(start_node)

    # Loop until you find the end
    while len(open_list) > 0:

        # Get the current node
        current_node = open_list[0]
        current_index = 0
        for index, item in enumerate(open_list):
            if item.f < current_node.f:
                current_node = item
                current_index = index

        # Pop current off open list, add to closed list
        open_list.pop(current_index)
        closed_list.append(current_node)

        # Found the goal
        if current_node == end_node:
            path = []
            current = current_node
            while current is not None:
```

```
            path.append(current.position)
            current = current.parent
        return path[::-1] # Return reversed path

    # Generate children
    children = []
    for new_position in [(0, -1), (0, 1), (-1, 0), (1, 0), (-1, -1), (-1, 1), (1, -
1), (1, 1)]: # Adjacent squares

        # Get node position
        node_position = (current_node.position[0] + new_position[0], current_node
.position[1] + new_position[1])

        # Make sure within range
        if node_position[0] > (len(maze) -
 1) or node_position[0] < 0 or node_position[1] > (len(maze[len(maze)-1]) -
1) or node_position[1] < 0:
            continue

        # Make sure walkable terrain
        if maze[node_position[0]][node_position[1]] != 0:
            continue

        # Create new node
        new_node = Node(current_node, node_position)

        # Append
        children.append(new_node)

    # Loop through children
    for child in children:

        # Child is on the closed list
        for closed_child in closed_list:
            if child == closed_child:
                continue

        # Create the f, g, and h values
        child.g = current_node.g + 1
        child.h = ((child.position[0] -
```

```
 end_node.position[0]) ** 2) + ((child.position[1] - end_node.position[1]) ** 2)
            child.f = child.g + child.h

            # Child is already in the open list
            for open_node in open_list:
                if child == open_node and child.g > open_node.g:
                    continue

            # Add the child to the open list
            open_list.append(child)

def main():

    maze = [[0, 0, 0, 0, 1, 0, 0, 0, 0, 0],
            [0, 0, 0, 0, 1, 0, 0, 0, 0, 0],
            [0, 0, 0, 0, 1, 0, 0, 0, 0, 0],
            [0, 0, 0, 0, 1, 0, 0, 0, 0, 0],
            [0, 0, 0, 0, 1, 0, 0, 0, 0, 0],
            [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
            [0, 0, 0, 0, 1, 0, 0, 0, 0, 0],
            [0, 0, 0, 0, 1, 0, 0, 0, 0, 0],
            [0, 0, 0, 0, 1, 0, 0, 0, 0, 0],
            [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]]

    start = (0, 0)
    end = (7, 6)

    path = astar(maze, start, end)
    print(path)

if__name__== '__main__':
    main()
```

Output-


A* ALGORITHM.ipynb
File Edit View Insert Runtime Tools Help Last edited on February 26
+ Code + Text
if __name__ == '__main__':
    main()
[(0, 0), (1, 1), (2, 2), (3, 3), (4, 3), (5, 4), (6, 5), (7, 6)]

Experiment No. 7

Aim : IMPLEMENTATION OF TOWER OF HANOI PROBLEM.

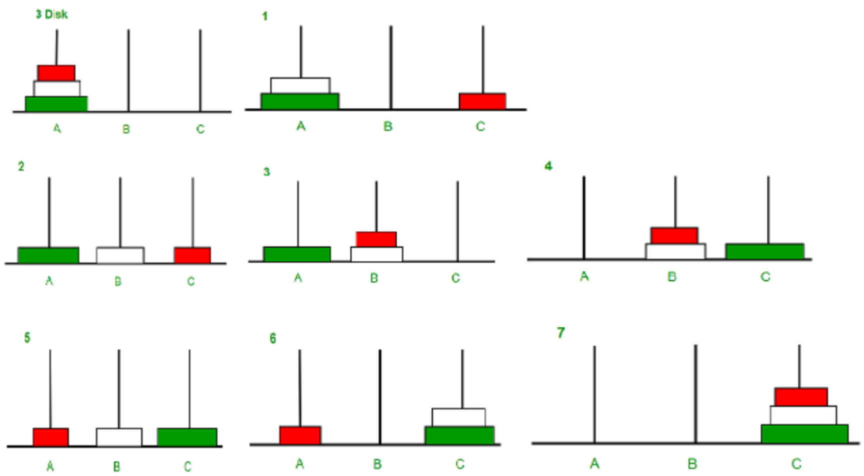
Algorithm :

1) Only one disk can be moved at a time.
2) Each move consists of taking the upper disk from one of the stacks and placing it on top of another stack i.e. a disk can only be moved if it is the uppermost disk on a stack.
3) No disk may be placed on top of a smaller disk. Note: Transferring the top n-1 disks from source rod to Auxiliary rod can again be thought of as a fresh problem and can be solved in the same manner.

Source code :

```
def TowerOfHanoi(n , source, destination, auxiliary):
  if n==1:
    print ("Move disk 1 from source",source,"to destination",destination)
    return
  TowerOfHanoi(n-1, source, auxiliary, destination)
  print ("Move disk",n,"from source",source,"to destination",destination)
  TowerOfHanoi(n-1, auxiliary, destination, source)

n = 3
TowerOfHanoi(n,'A','B','C')
```

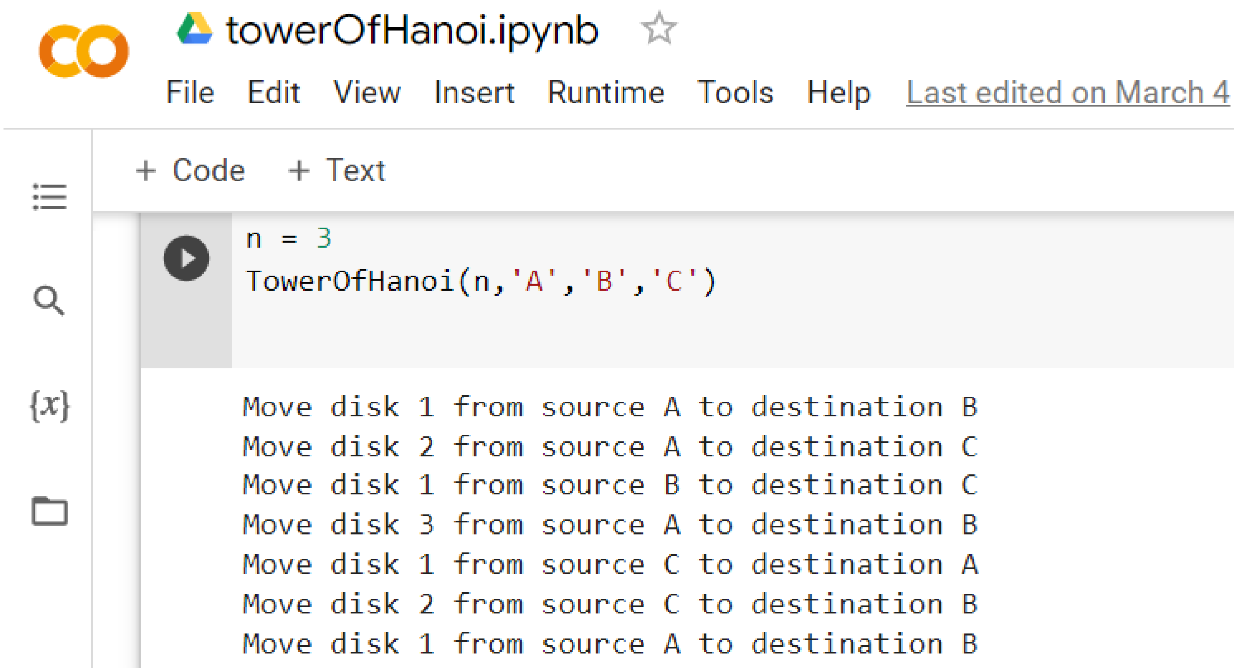
\+ Code + Text



No. of disks = 3

No. of towers = 3

Output :


towerOfHanoi.ipynb
File Edit View Insert Runtime Tools Help Last edited on March 4
+ Code + Text
n = 3
TowerOfHanoi(n,'A','B','C')
Move disk 1 from source A to destination B
Move disk 2 from source A to destination C
Move disk 1 from source B to destination C
Move disk 3 from source A to destination B
Move disk 1 from source C to destination A
Move disk 2 from source C to destination B
Move disk 1 from source A to destination B

Experiment No. 8

`

Aim : IMPLEMENTATION OF AGENT PROGRAMS FOR REAL-WORLD PROBLEMS (VACUUM CLEANER)

Algorithm :

Write a program to find the performance measurement in the vacuum cleaner problem given anumber of problem states.

1. Initialize goal_state to {'A':'0','B':'0'} where 0 indicates Clean and 1 indicates
2. Initialize cost to 0.
3. Accept the input from the user namely location of vacuum and status of rooms.
4. If the location with vacuum is A, check the status of both A and B, modify the state andcost accordingly.
5. Else the location with vacuum is B, check the status of the rooms, modify the state andcost.
6. Display the goal state and the performance measurement stored in cost.

Source code :

```
def vacuum_world():
        # initializing goal_state
        # 0 indicates Clean and 1 indicates Dirty
    goal_state = {'A': '0', 'B': '0'}
    cost = 0

    location_input = input("Enter Location of Vacuum") #user_input of location vac
uum is placed
    status_input = input("Enter status of " + location_input) #user_input if location i
s dirty or clean
    status_input_complement = input("Enter status of other room")
    print("Initial Location Condition" + str(goal_state))

    if location_input == 'A':
        # Location A is Dirty.
        print("Vacuum is placed in Location A")
        if status_input == '1':
```

```
print("Location A is Dirty.")
# suck the dirt  and mark it as clean
```

```
        goal_state['A'] = '0'
        cost += 1                      #cost for suck
        print("Cost for CLEANING A " + str(cost))
        print("Location A has been Cleaned.")

        if status_input_complement == '1':
            # if B is Dirty
            print("Location B is Dirty.")
            print("Moving right to the Location B. ")
            cost += 1                       #cost for moving right
            print("COST for moving RIGHT" + str(cost))
            # suck the dirt and mark it as clean
            goal_state['B'] = '0'
            cost += 1                       #cost for suck
            print("COST for SUCK " + str(cost))
            print("Location B has been Cleaned. ")
        else:
            print("No action" + str(cost))
            # suck and mark clean
            print("Location B is already clean.")

    if status_input == '0':
        print("Location A is already clean ")
        if status_input_complement == '1':# if B is Dirty
            print("Location B is Dirty.")
            print("Moving RIGHT to the Location B. ")
            cost += 1                       #cost for moving right
            print("COST for moving RIGHT " + str(cost))
            # suck the dirt and mark it as clean
            goal_state['B'] = '0'
            cost += 1                       #cost for suck
            print("Cost for SUCK" + str(cost))
            print("Location B has been Cleaned. ")
        else:
            print("No action " + str(cost))
            print(cost)
            # suck and mark clean
            print("Location B is already clean.")

else:
```

```
print("Vacuum is placed in location B")
# Location B is Dirty.
if status_input == '1':
    print("Location B is Dirty.")
    # suck the dirt and mark it as clean
    goal_state['B'] = '0'
    cost += 1  # cost for suck
    print("COST for CLEANING " + str(cost))
    print("Location B has been Cleaned.")

    if status_input_complement == '1':
        # if A is Dirty
        print("Location A is Dirty.")
        print("Moving LEFT to the Location A. ")
        cost += 1  # cost for moving right
        print("COST for moving LEFT" + str(cost))
        # suck the dirt and mark it as clean
        goal_state['A'] = '0'
        cost += 1  # cost for suck
        print("COST for SUCK " + str(cost))
        print("Location A has been Cleaned.")

else:
    print(cost)
    # suck and mark clean
    print("Location B is already clean.")

    if status_input_complement == '1': # if A is Dirty
        print("Location A is Dirty.")
        print("Moving LEFT to the Location A. ")
        cost += 1 # cost for moving right
        print("COST for moving LEFT " + str(cost))
        # suck the dirt and mark it as clean
        goal_state['A'] = '0'
        cost += 1 # cost for suck
        print("Cost for SUCK " + str(cost))
        print("Location A has been Cleaned. ")
    else:
        print("No action " + str(cost))
        # suck and mark clean
```

```
            print("Location A is already clean.")

    # done cleaning
    print("GOAL STATE: ")
    print(goal_state)
    print("Performance Measurement: " + str(cost))

vacuum_world()
```

Output :

vacuumCleaner.ipynb

File Edit View Insert Runtime Tools Help Last edited on March 11

+ Code + Text

```
Enter Location of Vacuum: B
Enter status of B: 1
Enter status of other room: 1
Initial Location Condition: {'A': '0', 'B': '0'}
Vacuum is placed in location B
Location B is Dirty.
COST for CLEANING: 1
Location B has been Cleaned.
Location A is Dirty.
Moving LEFT to the Location A.
COST for moving LEFT: 2
COST for SUCK 3
Location A has been Cleaned.
GOAL STATE:
{'A': '0', 'B': '0'}
Performance Measurement: 3
```

Experiment No. 9

Aim : IMPLEMENTATION OF CONSTRAINTS SATISFACTION PROBLEM(GRAPH COLORING PROBLEM)

Algorithm:

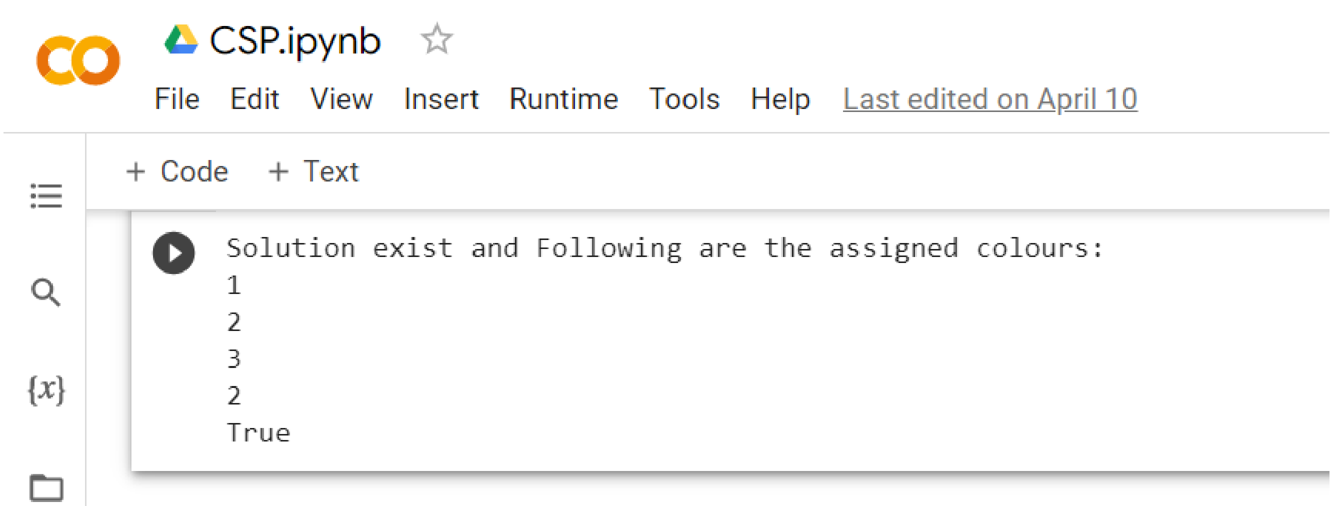
1. Create a recursive function that takes the graph, current index, number of vertices, andoutput color array.
2. If the current index is equal to the number of vertices. Print the color configuration in the output array.
3. Assign a color to a vertex (1 to m).
4. For every assigned color, check if the configuration is safe, (i.e. check if the adjacent vertices do not have the same color) recursively call the function with next index and number of vertices
5. If any recursive function returns true, break the loop and return true.
6. If no recursive function returns true then return false.

Source 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, colour, c):
    for i in range(self.V):
      if self.graph[v][i] == 1 and colour[i] == c:
        return False
    return True

  def graphColourUtil(self, m, colour, v):
    if v == self.V:
      return True

    for c in range(1, m + 1):
      if self.isSafe(v, colour, c) == True:
```

```
        colour[v] = c
        if self.graphColourUtil(m, colour, v + 1) == True:
          return True
        colour[v] = 0

  def graphColouring(self, m):
    colour = [0] * self.V
    if self.graphColourUtil(m, colour, 0) == None:
      return False

    print ("Solution exist and Following are the assigned colours:")
    for c in colour:
      print (c)
    return True

g = Graph(4)
g.graph = [[0, 1, 1, 1], [1, 0, 1, 0], [1, 1, 0, 1], [1, 0, 1, 0]]
m = 3
g.graphColouring(m)
```

Output :



Experiment No. 10

Aim : IMPLEMENTATION OF MINIMAX ALGORITHM.

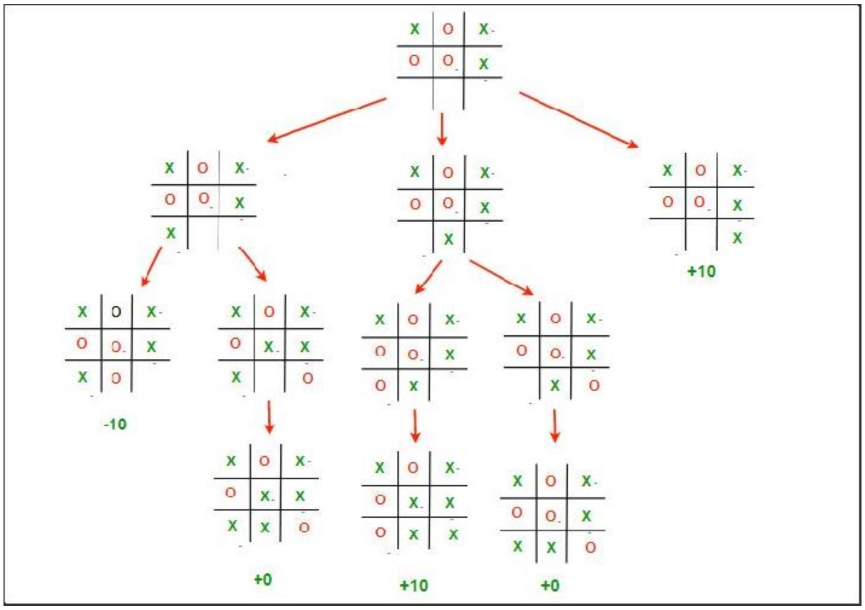
Algorithm :

We will be implementing the minimax algorithm for Tic-Tac-Toe. The game is to be played between two people. One of the players chooses 'O' and the other 'X' to mark their respectivecells. The game starts with one of the players and the game ends when either of the players hasone whole row/ column/ diagonal filled with his/her respective character ('O' or 'X').

Minimax algorithm is a recursive or backtracking algorithm which is used in decision-making and game theory. It provides an optimal move for the player assuming that the opponent is also playing optimally. It uses recursion to search through the game-tree. Here, we will recursively generate the game tree by exploring all possible moves for each board state and upon reaching aterminal state, we will assign a value of :

- 1 for winning,
- -1 for losing
- 0 for draw.

Then based on these terminal states, for each explored turn either maximizer or minimizer willpick the most appropriate move.



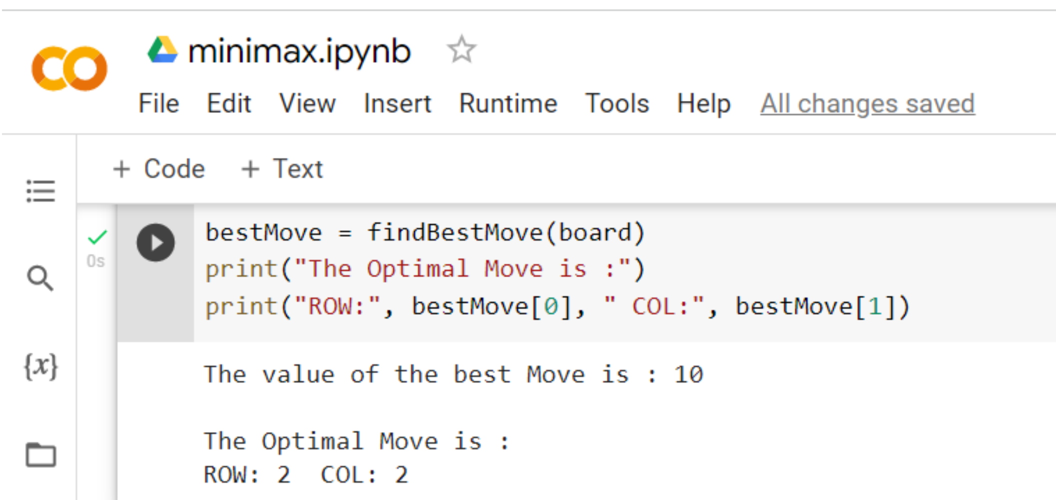
Source code :

```
player, opponent = 'x', 'o'
def isMovesLeft(board) :
  for i in range(3) :
    for j in range(3) :
      if (board[i][j] == '_') :
        return True
  return False

def evaluate(b) :
  for row in range(3) :
    if (b[row][0] == b[row][1] and b[row][1] == b[row][2]) :
      if (b[row][0] == player) :
        return 10

  for col in range(3) :
    if (b[0][col] == b[1][col] and b[1][col] == b[2][col]) :
      if (b[0][col] == player) :
        return 10


  if (b[0][0] == b[1][1] and b[1][1] == b[2][2]) :
    if (b[0][0] == player) :
      return 10


  if (b[0][2] == b[1][1] and b[1][1] == b[2][0]) :
    if (b[0][2] == player) :
      return 10
  return 0

def minimax(board, depth, isMax) :
  score = evaluate(board)
```

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

  else :
    best = 1000
    for i in range(3) :
      for j in range(3) :
        if (board[i][j] == '_') :
          board[i][j] = opponent
          best = min(best, minimax(board, depth + 1, not isMax))
          board[i][j] = '_'
    return best

def findBestMove(board) :
  bestVal = -1000
  bestMove = (-1, -1)
  for i in range(3) :
    for j in range(3) :
      if (board[i][j] == '_') :
        board[i][j] = player
        moveVal = minimax(board, 0, False)
        board[i][j] = '_'
        if (moveVal > bestVal) :
          bestMove = (i, j)
          bestVal = moveVal

  print("The value of the best Move is :", bestVal)
```

```
  print()
  return bestMove

board = [
  [ 'x', 'o', 'x' ],
  [ 'o', 'o', 'x' ],
  [ '_', '_', '_' ]
]

bestMove = findBestMove(board)
print("The Optimal Move is :")
print("ROW:", bestMove[0], " COL:", bestMove[1])
```

Output :



<u>Experiment No. 11</u>

Aim : IMPLEMENTATION OF PROPOSITIONAL LOGIC IN REAL WORLD PROBLEMS.

Algorithm :

```
Collecting formula
  Downloading formula-2.0.1.tar.gz (24.3 MB)
     |████████████████████████████████| 24.3 MB 17.1 MB/s
Collecting pybind11>=2.4
  Using cached pybind11-2.9.2-py2.py3-none-any.whl (213 kB)
Building wheels for collected packages: formula
  Building wheel for formula (setup.py) ... done
  Created wheel for formula: filename=formula-2.0.1-cp37-cp37m-
linux_x86_64.whl size=1104718
sha256=493f62805544ac7316f0f1f5ea58d6a00c74594b1918e6b5350b57d5625d3d
4a
  Stored in directory:
/root/.cache/pip/wheels/dd/ed/a9/3962025d76b8dbf796a5d02985ffa66a3848cf2add
259869f7
Successfully built formula
Installing collected packages: pybind11, formula
Successfully installed formula-2.0.1 pybind11-2.9.2
```

Source code :

```
 !pip install formula


class Formula:
  def___invert__(self):
    return Not(self)
  def___and__(self, other):
  return And(self, other)
  def___or__(self, other):
  return Or(self, other)
  def___rshift__(self, other):
    return Implies(self, other)
```

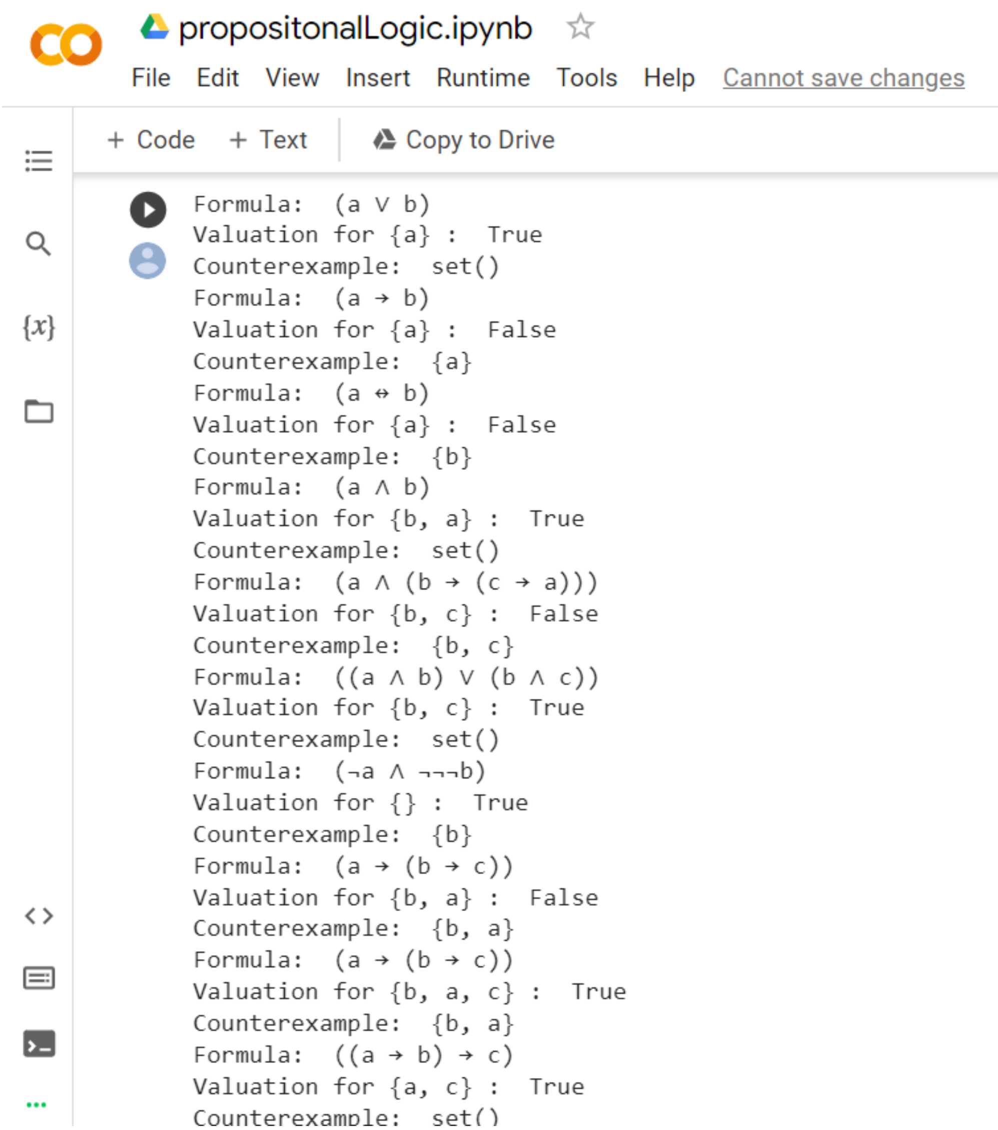
```
def__lshift__(self, other):
  return Iff(self, other)
def__eq__(self, other):
  return self.__class__== other.__class__and self.eq(other)
def v(self, v):
  raise NotImplementedError("Plain formula can not be valuated")
def _t(self, left, right):
  while True:
    found = True
    for item in left:
      if item in right:
        return None
      if not isinstance(item, Atom):
        left.remove(item)
        tup = item._tleft(left, right)
        left, right = tup[0]
        if len(tup) > 1:
          v = self._t(*tup[1])
          if v is not None:
          return v
        found = False
        break
    for item in right:
      if item in left:
        return None
      if not isinstance(item, Atom):
        right.remove(item)
        tup = item._tright(left, right)
        left, right = tup[0]
        if len(tup) > 1:
          v = self._t(*tup[1])
          if v is not None:
          return v
        found = False
        break
    if found:
      return set(left)
def t(self):
  return self._t([], [self])
```

```
class BinOp(Formula):
  def__init__(self, lchild, rchild):
    self.lchild = lchild
  self.rchild  =  rchild
  def_str_(self):
    return '(' + str(self.lchild) + ' ' + self.op+ ' ' + str(self.rchild) + ')'
  def eq(self, other):
    return self.lchild == other.lchild and self.rchild == other.rchild

class And(BinOp):
  op = 'Λ'
  def v(self, v):
    return self.lchild.v(v) and self.rchild.v(v)
  def _tleft(self, left, right):
    return (left + [self.lchild, self.rchild], right),
  def _tright(self, left, right):
    return (left, right + [self.lchild]), (left, right + [self.rchild])

class Or(BinOp):
  op = 'V'
  def v(self, v):
    return self.lchild.v(v) or self.rchild.v(v)
  def _tleft(self, left, right):
    return (left + [self.lchild], right), (left + [self.rchild], right)
  def _tright(self, left, right):
    return (left, right + [self.lchild, self.rchild]),

class Implies(BinOp):
  op = '→'
  def v(self, v):
    return not self.lchild.v(v) or self.rchild.v(v)
  def _tleft(self, left, right):
    return (left + [self.rchild], right), (left, right + [self.lchild])
  def _tright(self, left, right):
    return (left + [self.lchild], right + [self.rchild]),

class Iff(BinOp):
  op = '↔'
  def v(self, v):
    return self.lchild.v(v) is self.rchild.v(v)
```

```
  def _tleft(self, left, right):
    return (left + [self.lchild, self.rchild], right), (left, right + [self.lchild, self.rchild])
  def _tright(self, left, right):
    return (left + [self.lchild], right + [self.rchild]), (left + [self.rchild], right + [self.l
child])

class Not(Formula):
  def___init__(self, child):
    self.child = child
  def v(self, v):
    return not self.child.v(v)
  def___str__(self):
    return '¬' + str(self.child)
  def eq(self, other):
    return self.child == other.child
  def _tleft(self, left, right):
    return (left, right + [self.child]),
  def _tright(self, left, right):
  return (left + [self.child], right),

class Atom(Formula):
  def___init__(self, name):
    self.name = name
  def___hash__(self):
    return hash(self.name)
  def v(self, v):
    return self in v
  def___str__(self):
    return str(self.name)
  __repr___=___str___
  def eq(self, other):
    return self.name == other.name

a = Atom('a')
b = Atom('b')
c = Atom('c')

def dop(f, e):
  print("Formula: ", f)
  print("Valuation for", e, ": ", f.v(e))
```

```
  print("Counterexample: ", f.t())

dop(a | b, {a})
dop(a >> b, {a})
dop(a << b, {a})
dop(a & b, {a,b})
dop(a & b >> (c >> a), {b,c})
dop(a & b | b & c, {b,c})
dop(~a & ~~~b, {})
dop(a >> (b >> c), {a, b})
dop(a >> (b >> c), {a, b, c})
dop(a >> b >> c, {a, c})
dop(((c | ~b) >> (b | c)) >> (b | c), {a, c})
dop(a | ~a, {})
dop(a >> a, {a})
dop(a << a, {})
dop((a >> b) | (b >> a), {})
dop((~a | b) | (~b | a), {})
dop((~a | a) | (~b | b), {})
```

Output-


propositonalLogic.ipynb
File Edit View Insert Runtime Tools Help Cannot save changes
+ Code + Text
Copy to Drive
Formula: (a ∨ b)
Valuation for {a} : True
Counterexample: set()
Formula: (a → b)
Valuation for {a} : False
Counterexample: {a}
Formula: (a ↔ b)
Valuation for {a} : False
Counterexample: {b}
Formula: (a ∧ b)
Valuation for {b, a} : True
Counterexample: set()
Formula: (a ∧ (b → (c → a)))
Valuation for {b, c} : False
Counterexample: {b, c}
Formula: ((a ∧ b) ∨ (b ∧ c))
Valuation for {b, c} : True
Counterexample: set()
Formula: (¬a ∧ ¬¬¬b)
Valuation for {} : True
Counterexample: {b}
Formula: (a → (b → c))
Valuation for {b, a} : False
Counterexample: {b, a}
Formula: (a → (b → c))
Valuation for {b, a, c} : True
Counterexample: {b, a}
Formula: ((a → b) → c)
Valuation for {a, c} : True
Counterexample: set()

propositonalLogic.ipynb

File Edit View Insert Runtime Tools Help Cannot save changes

+ Code + Text Copy to Drive

```
Counterexample:  {b, a}
[ ] Formula:  ((a → b) → c)
    Valuation for {a, c} :  True
    Counterexample:  set()
    Formula:  (((c ∨ ¬b) → (b ∨ c)) → (b ∨ c))
    Valuation for {a, c} :  True
    Counterexample:  None
    Formula:  (a ∨ ¬a)
    Valuation for {} :  True
    Counterexample:  None
    Formula:  (a → a)
    Valuation for {a} :  True
    Counterexample:  None
    Formula:  (a ↔ a)
    Valuation for {} :  True
    Counterexample:  None
    Formula:  ((a → b) ∨ (b → a))
    Valuation for {} :  True
    Counterexample:  None
    Formula:  ((¬a ∨ b) ∨ (¬b ∨ a))
    Valuation for {} :  True
    Counterexample:  None
    Formula:  ((¬a ∨ a) ∨ (¬b ∨ b))
    Valuation for {} :  True
    Counterexample:  None
```

Experiment No.12

Aim : IMPLEMENTATION OF UNIFICATION AND RESOLUTION OF REAL-WORLD PROBLEMS.

Algorithm :

UNIFICATION-

Let Ψ1 and Ψ2 be two atomic sentences and σ be a unifier such that, **Ψ1**σ **= Ψ2**σ, then it can beexpressed as **UNIFY(Ψ1, Ψ2).**

Step. 1: If Ψ1 or Ψ2 is a variable or constant, then:

- a) If Ψ1 or Ψ2 are identical, then return NIL.
- b) Else if Ψ1is a variable,
  - a. then if Ψ1 occurs in Ψ2, then return FAILURE
  - b. Else return { (Ψ2/ Ψ1)}.
- c) Else if Ψ2 is a variable,
  - a. If Ψ2 occurs in Ψ1 then return FAILURE,
  - b. Else return {(Ψ1/ Ψ2)}.
- d) Else return FAILURE.

Step.2: If the initial Predicate symbol in Ψ1 and Ψ2 are not same, then return FAILURE.

Step. 3: IF Ψ1 and Ψ2 have a different number of arguments, then return FAILURE.

Step. 4: Set Substitution set(SUBST) to NIL.

Step. 5: For i=1 to the number of elements in Ψ1.

- a) Call Unify function with the ith element of Ψ1 and ith element of Ψ2, and put the result into S.
- b) If S = failure then returns Failure
- c) If S ≠ NIL then do,
  - a. Apply S to the remainder of both L1 and L2.
  - b. SUBST= APPEND(S, SUBST).

Step.6: Return SUBST.

RESOLUTION:

Steps for Resolution:

1. Conversion of facts into first-order logic.
2. Convert FOL statements into CNF
3. Negate the statement which needs to prove (proof by contradiction)
4. Draw resolution graph (unification).

Source code :

UNIFICATION :

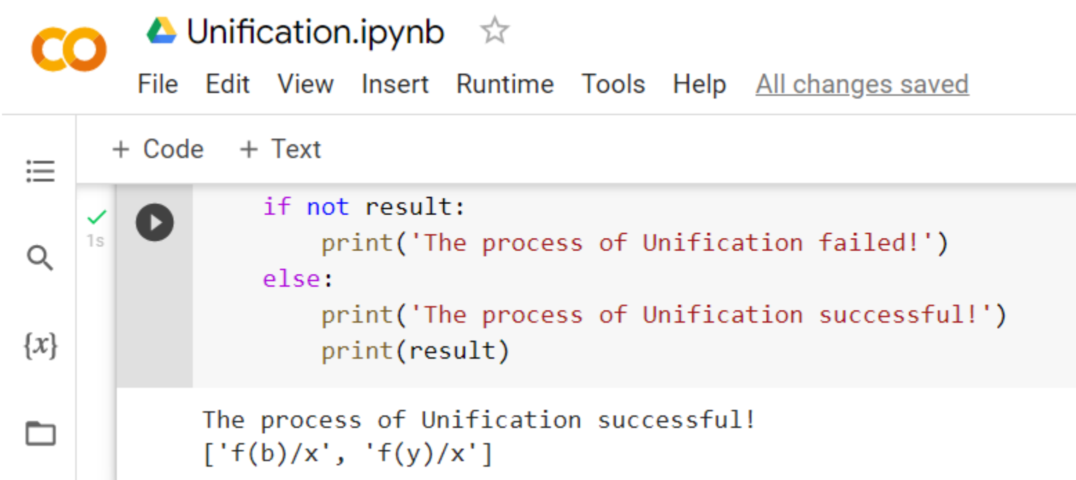
```
def get_index_comma(string):
    index_list = list()
    par_count = 0

    for i in range(len(string)):
        if string[i] == ',' and par_count == 0:
            index_list.append(i)
        elif string[i] == '(':
            par_count += 1
        elif string[i] == ')':
            par_count -= 1

    return index_list

def is_variable(expr):
    for i in expr:
```

```
        if i == '(' or i == ')':
            return False

    return True

def process_expression(expr):
    expr = expr.replace(' ', '')
    index = None
    for i in range(len(expr)):
        if expr[i] == '(':
            index = i
            break
    predicate_symbol = expr[:index]
    expr = expr.replace(predicate_symbol, '')
    expr = expr[1:len(expr) - 1]
    arg_list = list()
    indices = get_index_comma(expr)

    if len(indices) == 0:
        arg_list.append(expr)
    else:
        arg_list.append(expr[:indices[0]])
        for i, j in zip(indices, indices[1:]):
            arg_list.append(expr[i + 1:j])
        arg_list.append(expr[indices[len(indices) - 1] + 1:])

    return predicate_symbol, arg_list

def get_arg_list(expr):
    _, arg_list = process_expression(expr)

    flag = True
    while flag:
        flag = False

        for i in arg_list:
            if not is_variable(i):
                flag = True
                _, tmp = process_expression(i)
                for j in tmp:
```

```
                if j not in arg_list:
                    arg_list.append(j)
            arg_list.remove(i)

    return arg_list

def check_occurs(var, expr):
    arg_list = get_arg_list(expr)
    if var in arg_list:
        return True

    return False

def unify(expr1, expr2):

    if is_variable(expr1) and is_variable(expr2):
        if expr1 == expr2:
            return 'Null'
        else:
            return False
    elif is_variable(expr1) and not is_variable(expr2):
        if check_occurs(expr1, expr2):
            return False
        else:
            tmp = str(expr2) + '/' + str(expr1)
            return tmp
    elif not is_variable(expr1) and is_variable(expr2):
        if check_occurs(expr2, expr1):
            return False
        else:
            tmp = str(expr1) + '/' + str(expr2)
            return tmp
    else:
        predicate_symbol_1, arg_list_1 = process_expression(expr1)
        predicate_symbol_2, arg_list_2 = process_expression(expr2)

        # Step 2
        if predicate_symbol_1 != predicate_symbol_2:
            return False
        # Step 3
```

```
        elif len(arg_list_1) != len(arg_list_2):
            return False
        else:
            # Step 4: Create substitution list
            sub_list = list()

            # Step 5:
            for i in range(len(arg_list_1)):
                tmp = unify(arg_list_1[i], arg_list_2[i])

                if not tmp:
                    return False
                elif tmp == 'Null':
                    pass
                else:
                    if type(tmp) == list:
                        for j in tmp:
                            sub_list.append(j)
                    else:
                        sub_list.append(tmp)

            # Step 6
            return sub_list

if __name__ == '__main__':

    f1 = 'Q(a, g(x, a), f(y))'
    f2 = 'Q(a, g(f(b), a), x)'
    # f1 = input('f1 : ')
    # f2 = input('f2 : ')

    result = unify(f1, f2)
    if not result:
        print('The process of Unification failed!')
    else:
        print('The process of Unification successful!')
        print(result)
```

Output:



RESOLUTION :

```
import copy
import time

class Parameter:
    variable_count = 1

    def___init_(self, name=None):
        if name:
            self.type = "Constant"
            self.name = name
        else:
            self.type = "Variable"
            self.name = "v" + str(Parameter.variable_count)
            Parameter.variable_count += 1

    def isConstant(self):
        return self.type == "Constant"

    def unify(self, type_, name):
        self.type = type_
        self.name = name
```

```
    def __eq__(self, other):
        return self.name == other.name

    def __str__(self):
        return self.name

class Predicate:
    def __init__(self, name, params):
        self.name = name
        self.params = params

    def __eq__(self, other):
        return self.name == other.name and all(a == b for a, b in zip(self.params, other.params))

    def __str__(self):
        return self.name + "(" + ",".join(str(x) for x in self.params) + ")"

    def getNegatedPredicate(self):
        return Predicate(negatePredicate(self.name), self.params)

class Sentence:
    sentence_count = 0

    def __init__(self, string):
        self.sentence_index = Sentence.sentence_count
        Sentence.sentence_count += 1
        self.predicates = []
        self.variable_map = {}
        local = {}

        for predicate in string.split("|"):
            name = predicate[:predicate.find("(")]
            params = []

            for param in predicate[predicate.find("(") + 1: predicate.find(")")].split(","):
                if param[0].islower():
                    if param not in local: # Variable
                        local[param] = Parameter()
                        self.variable_map[local[param].name] = local[param]
                    new_param = local[param]
                else:
                    new_param = Parameter(param)
```

```
                    self.variable_map[param] = new_param

                params.append(new_param)

            self.predicates.append(Predicate(name, params))

    def getPredicates(self):
        return [predicate.name for predicate in self.predicates]

    def findPredicates(self, name):
        return [predicate for predicate in self.predicates if predicate.name == name]

    def removePredicate(self, predicate):
        self.predicates.remove(predicate)
        for key, val in self.variable_map.items():
            if not val:
                self.variable_map.pop(key)

    def containsVariable(self):
        return any(not param.isConstant() for param in self.variable_map.values())

    def __eq__(self, other):
        if len(self.predicates) == 1 and self.predicates[0] == other:
            return True
        return False

    def __str__(self):
        return "".join([str(predicate) for predicate in self.predicates])

class KB:
    def __init__(self, inputSentences):
        self.inputSentences = [x.replace(" ", "") for x in inputSentences]
        self.sentences = []
        self.sentence_map = {}

    def prepareKB(self):
        self.convertSentencesToCNF()
        for sentence_string in self.inputSentences:
            sentence =  Sentence(sentence_string)
            for predicate in sentence.getPredicates():
                self.sentence_map[predicate] = self.sentence_map.get(
                    predicate, []) + [sentence]
```

```
def convertSentencesToCNF(self):
    for sentenceIdx in range(len(self.inputSentences)):
        # Do negation of the Premise and add them as literal
        if "=>" in self.inputSentences[sentenceIdx]:
            self.inputSentences[sentenceIdx] = negateAntecedent(
                self.inputSentences[sentenceIdx])

def askQueries(self, queryList):
    results = []

    for query in queryList:
        negatedQuery = Sentence(negatePredicate(query.replace(" ", "")))
        negatedPredicate = negatedQuery.predicates[0]
        prev_sentence_map = copy.deepcopy(self.sentence_map)
        self.sentence_map[negatedPredicate.name] = self.sentence_map.get(
            negatedPredicate.name, []) + [negatedQuery]
        self.timeLimit = time.time() + 40

        try:
            result = self.resolve([negatedPredicate], [
                                  False]*(len(self.inputSentences) + 1))
        except:
            result = False

        self.sentence_map = prev_sentence_map

        if result:
            results.append("TRUE")
        else:
            results.append("FALSE")

    return results

def resolve(self, queryStack, visited, depth=0):
    if time.time() > self.timeLimit:
        raise Exception
    if queryStack:
        query = queryStack.pop(-1)
        negatedQuery = query.getNegatedPredicate()
        queryPredicateName = negatedQuery.name
        if queryPredicateName not in self.sentence_map:
            return False
        else:
```

```
                queryPredicate = negatedQuery
                for kb_sentence in self.sentence_map[queryPredicateName]:
                    if not visited[kb_sentence.sentence_index]:
                        for kbPredicate in kb_sentence.findPredicates(queryPredicateName):

                            canUnify, substitution = performUnification(
                                copy.deepcopy(queryPredicate), copy.deepcopy(kbPredicate))

                            if canUnify:
                                newSentence = copy.deepcopy(kb_sentence)
                                newSentence.removePredicate(kbPredicate)
                                newQueryStack = copy.deepcopy(queryStack)

                                if substitution:
                                    for old, new in substitution.items():
                                        if old in newSentence.variable_map:
                                            parameter = newSentence.variable_map[old]
                                            newSentence.variable_map.pop(old)
                                            parameter.unify(
                                                "Variable" if new[0].islower() else "Constant", new)
                                            newSentence.variable_map[new] = parameter

                                    for predicate in newQueryStack:
                                        for index, param in enumerate(predicate.params):
                                            if param.name in substitution:
                                                new = substitution[param.name]
                                                predicate.params[index].unify(
                                                    "Variable" if new[0].islower() else "Constant", new)

                                for predicate in newSentence.predicates:
                                    newQueryStack.append(predicate)

                                new_visited = copy.deepcopy(visited)
                                if kb_sentence.containsVariable() and len(kb_sentence.predicates) >
1:
                                    new_visited[kb_sentence.sentence_index] = True

                                if self.resolve(newQueryStack, new_visited, depth + 1):
                                    return True
                return False
        return True

def performUnification(queryPredicate, kbPredicate):
```

```
    substitution = {}
    if queryPredicate == kbPredicate:
        return True, {}
    else:
        for query, kb in zip(queryPredicate.params, kbPredicate.params):
            if query == kb:
                continue
            if kb.isConstant():
                if not query.isConstant():
                    if query.name not in substitution:
                        substitution[query.name] = kb.name
                    elif substitution[query.name] != kb.name:
                        return False, {}
                    query.unify("Constant", kb.name)
                else:
                    return False, {}
            else:
                if not query.isConstant():
                    if kb.name not in substitution:
                        substitution[kb.name] = query.name
                    elif substitution[kb.name] != query.name:
                        return False, {}
                    kb.unify("Variable", query.name)
                else:
                    if kb.name not in substitution:
                        substitution[kb.name] = query.name
                    elif substitution[kb.name] != query.name:
                        return False, {}
    return True, substitution

def negatePredicate(predicate):
    return predicate[1:] if predicate[0] == "~" else "~" + predicate

def negateAntecedent(sentence):
    antecedent = sentence[:sentence.find("=>")]
    premise = []

    for predicate in antecedent.split("&"):
        premise.append(negatePredicate(predicate))

    premise.append(sentence[sentence.find("=>") + 2:])
    return "|".join(premise)
```

```
def getInput(filename):
     with open(filename, "r") as file:
        noOfQueries = int(file.readline().strip())
        inputQueries = [file.readline().strip() for _ in range(noOfQueries)]
        noOfSentences = int(file.readline().strip())
        inputSentences =  [file.readline().strip()
                          for _ in range(noOfSentences)]
        return inputQueries, inputSentences

def printOutput(filename, results):
    print(results)
    with open(filename, "w") as file:
        for line in results:
            file.write(line)
            file.write("\n")
    file.close()

if__name__ == '_main__':
    inputQueries_, inputSentences_ = getInput('/New Text Document.txt')
    knowledgeBase = KB(inputSentences_)
    knowledgeBase.prepareKB()
    results_ = knowledgeBase.askQueries(inputQueries_)
    printOutput("/output.txt", results_)
```

Output :