Artificial Intelligence Lab Report

Submitted by

AFIFAH KHAN (1BM20CS195)
Batch: D2

Course: Artificial Intelligence
Course Code: 20CS5PCAIP
Sem & Section: 5D

BACHELOR OF ENGINEERING
in
COMPUTER SCIENCE AND ENGINEERING

B. M. S. COLLEGE OF ENGINEERING
(Autonomous Institution under VTU)
BENGALURU-560019
2022-2023

Table of contents

9/11/22 **Program 1 - Tic Tac Toe**

Algorithm:

Tic Tac Toe algorithm:

def printBoard(board):
    to print current state of board

def user1Turn(board):
    enter user 1's position.

def user2Turn(board):
    enter user 2's position.

def Minmax(board, player):
    x = analyze board(board)
    if (x) = 0:
        return (x * player)
    if pos = -1.
        return 0.

def compTurn(board):
    to find the position the
    computer's turn.

def analyzeboard(board):
    function for analyzing board.

def Main():
    main function.

Code:

```
import random
#two player tic tac toe
board = ["-","-","-",
        "-","-","-",
        "-","-","-"]

game_running = True
current_player = "X"
winner = None
```

```
#making the board
def print_board(board):
    print(board[0]+"|"+board[1]+"|"+board[2]+"|")
    print("------")
    print(board[3]+"|"+board[4]+"|"+board[5]+"|")
    print("------")
    print(board[6]+"|"+board[7]+"|"+board[8]+"|")
    print("------")

#taking player input
def player_input(board):
    inp = int(input("Enter position between 1-9 : "))
    if inp>=1 and inp<=9 and board[inp-1] == "-":
        board[inp-1] = current_player

    else:
        print("Player already in that spot")


#check for win and tie

def check_horizontal(board):
    global winner
    if board[0] == board[1] == board[2] and board[0] !="-":
        winner = board[0]
        return True
    elif board[3] == board[4] == board[5] and board[3] !="-":
        winner = board[3]
        return True
    elif board[6] == board[7] == board[8] and board[6] !="-":
        winner = board[6]
        return True

def check_col(board):
    global winner
    if board[0] == board[3] == board[6]and board[0] !="-":
        winner = board[0]
        return True
    elif board[1] == board[4] == board[7] and board[1] !="-":
        winner = board[0]
```

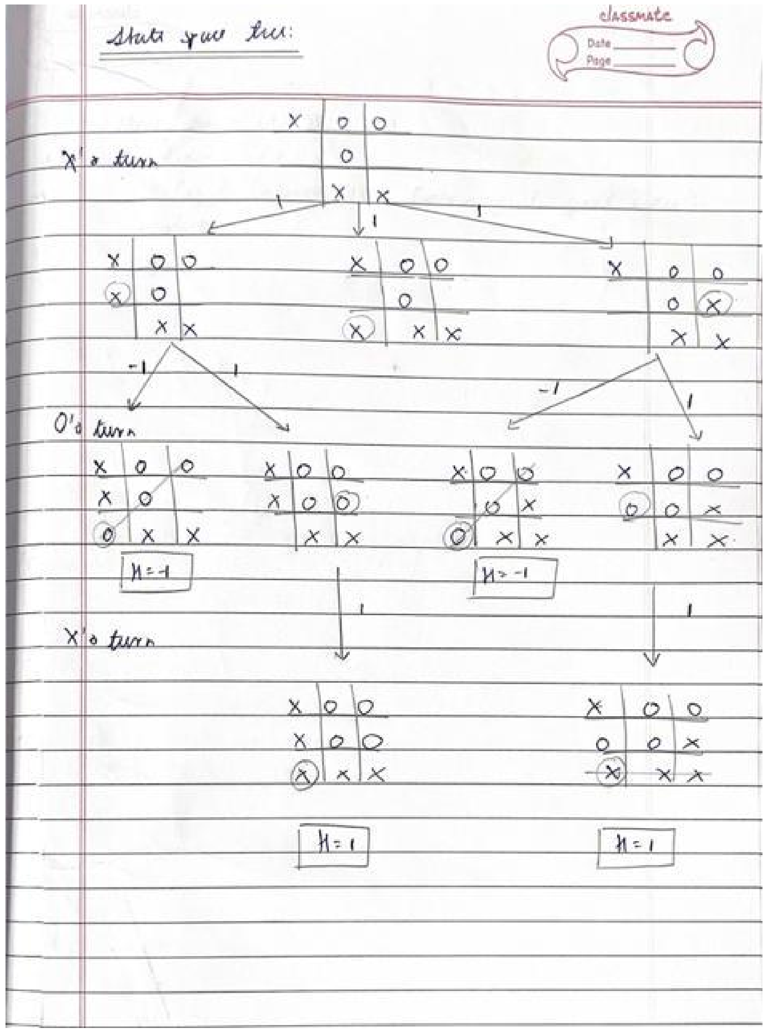
```
        return True

    elif board[2] == board[5] == board[8] and board[2] !="-":
        winner = board[0]
        return True


def check_diag(board):
    global winner
    if board[0] == board[4] == board[8] and board[0] !="-":
        winner = board[0]
        return True
    elif board[2] == board[4] == board[6] and board[2] !="-":
        winner = board[2]
        return True

def check_tie(board):
    global game_running
    if "-" not in board:
        print_board(board)
        print("it is a tie!")
        game_running = False


def check_win(board):
    global game_running
    if check_horizontal(board):
        print_board(board)
        print(f"The winner is {winner}!")
        game_running = False

    elif check_col(board):
        print_board(board)
        print(f"The winner is {winner}!")
        game_running = False

    elif check_diag(board):
        print_board(board)
        print(f"The winner is {winner}!")
        game_running = False
```

```
def switch_player():
    global current_player
    if current_player == "X":
        current_player = "O"
    else:
        current_player = "X"


#computer
def computer(board):
    global current_player
    while current_player == "O":
        position = random.randint(0,8)
        if board[position] == "-":
            board[position] = "O"
            switch_player()
    #    else:
    #       computer(board)


while game_running:
    print_board(board)
    player_input(board)
    check_win(board)
    check_tie(board)
    switch_player()
    computer(board)
    check_win(board)
    check_tie(board)
```

Output:

```
-|-|-|
------
-|-|-|
------
-|-|-|
------
Enter position between 1-9 : 1
X|-|-|
------
-|-|O|
------
-|-|-|
------
Enter position between 1-9 : 5
X|-|-|
------
-|X|O|
------
O|-|-|
------
Enter position between 1-9 : 9
X|-|-|
------
-|X|O|
------
O|-|X|
------
The winner is X!
X|-|O|
------
-|X|O|
------
O|-|X|
------
The winner is X!


...Program finished with exit code 0
Press ENTER to exit console.
```

State Space Diagram:



16/11/22 **Program 2 - 8 Puzzle Using BFS**

Algorithm:

6|11|16

8 puzzle BFS

Aim - To implement 8 puzzle game using BFS search algorithm.

Algorithm:

```
Breadth first search (initial state, goal state).
    return success or failure

frontier = Queue.new (initial state)
explored = Set.new()

while not frontier.isEmpty():
    state = frontier.dequeue()
    explored.add(state)

    if goalTest(state):
        return success(state)

    for neighbour in state.neighbours():
        if neighbour not in frontier ∪ explored:
            frontier.enqueue(neighbour)

return Failure.
```

Code:

```
import copy

inp=[[1,2,3],[4,-1,5],[6,7,8]]
```

```
out=[[1,2,3],[4,5,6],[7,8,-1]]

print("Enter input puzzle")
for i in range(3):
  for j in range(3):
    inp[i][j]=int(input("Enter number at "+str(i)+","+str(j)+" ->"))

def move(temp, movement):
  if movement=="up":
    for i in range(3):
      for j in range(3):
        if(temp[i][j]==-1):
          if i!=0:
            temp[i][j]=temp[i-1][j]
            temp[i-1][j]=-1
          return temp

  if movement=="down":
    for i in range(3):
      for j in range(3):
        if(temp[i][j]==-1):
          if i!=2:
            temp[i][j]=temp[i+1][j]
            temp[i+1][j]=-1
          return temp

  if movement=="left":
    for i in range(3):
      for j in range(3):
        if(temp[i][j]==-1):
          if j!=0:
            temp[i][j]=temp[i][j-1]
            temp[i][j-1]=-1
          return temp

  if movement=="right":
    for i in range(3):
      for j in range(3):
        if(temp[i][j]==-1):
          if j!=2:
```

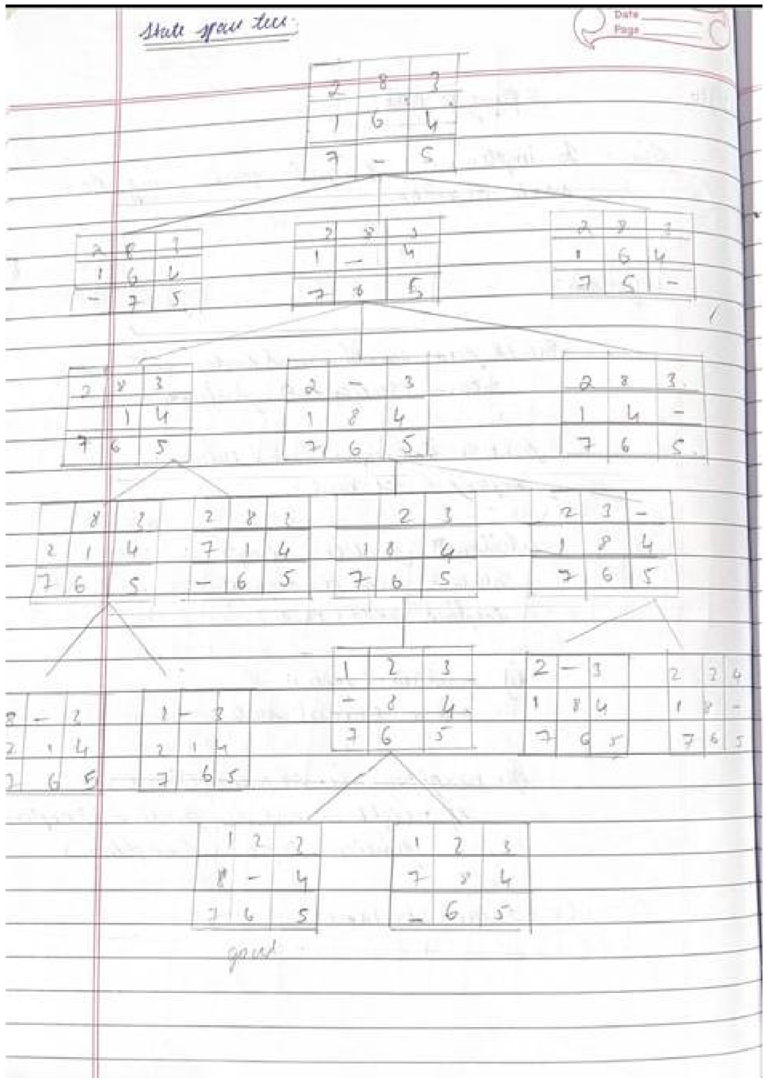
```
      temp[i][j]=temp[i][j+1]
      temp[i][j+1]=-1
     return temp

def bfs():
  global inp
  global out

  pathcost=0

  queue=[]
  inpx=[inp,"none"]
  queue.append(inpx)
  while(True):
    puzzle=queue.pop()
    pathcost=pathcost+1
    print(str(puzzle[1])+" --> "+str(puzzle[0]))
    if(puzzle[0]==out):
      print("Found")
      print('Path cost-> '+str(pathcost-1))
      break
    else:
      if(puzzle[1]!="down"):
        temp=copy.deepcopy(puzzle[0])
        up=move(temp, "up")
        upx=[up,"up"]
        queue.insert(0, upx)

      if(puzzle[1]!="right"):
        temp=copy.deepcopy(puzzle[0])
        left=move(temp, "left")
        leftx=[left,"left"]
        queue.insert(0, leftx)

      if(puzzle[1]!="up"):
        temp=copy.deepcopy(puzzle[0])
        down=move(temp, "down")
        downx=[down,"down"]
        queue.insert(0, downx)
```

```
    if(puzzle[1]!="left"):
      temp=copy.deepcopy(puzzle[0])
      right=move(temp, "right")
      rightx=[right,"right"]
      queue.insert(0, rightx)

print('~~~~~~~~~~~~~ BFS ~~~~~~~~~~~~~')
bfs()
```

Output:

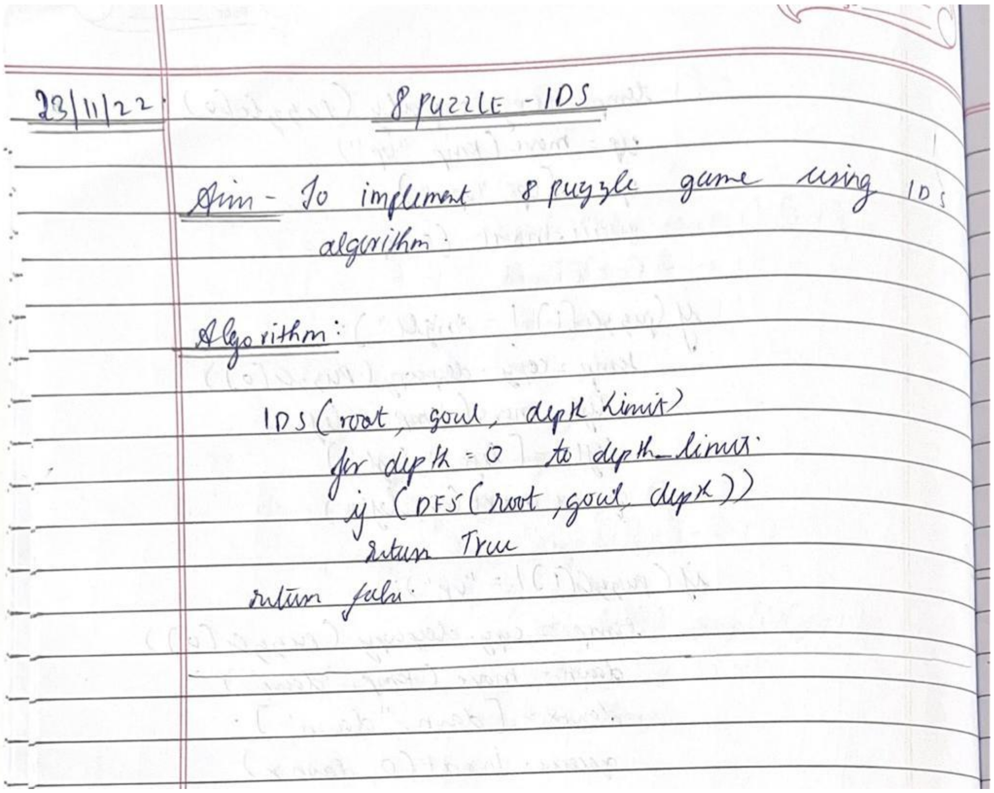
```
main.py
Enter input puzzle
Enter number at 0,0 ->1
Enter number at 0,1 ->2
Enter number at 0,2 ->3
Enter number at 1,0 ->4
Enter number at 1,1 ->5
Enter number at 1,2 ->6
Enter number at 2,0 ->-1
Enter number at 2,1 ->7
Enter number at 2,2 ->8
~~~~~~~~~~~~~ BFS ~~~~~~~~~~~~~
none --> [[1, 2, 3], [4, 5, 6], [-1, 7, 8]]
up --> [[1, 2, 3], [-1, 5, 6], [4, 7, 8]]
left --> [[1, 2, 3], [4, 5, 6], [-1, 7, 8]]
down --> [[1, 2, 3], [4, 5, 6], [-1, 7, 8]]
right --> [[1, 2, 3], [4, 5, 6], [7, -1, 8]]
up --> [[-1, 2, 3], [1, 5, 6], [4, 7, 8]]
left --> [[1, 2, 3], [-1, 5, 6], [4, 7, 8]]
right --> [[1, 2, 3], [5, -1, 6], [4, 7, 8]]
up --> [[1, 2, 3], [-1, 5, 6], [4, 7, 8]]
left --> [[1, 2, 3], [4, 5, 6], [-1, 7, 8]]
down --> [[1, 2, 3], [4, 5, 6], [-1, 7, 8]]
left --> [[1, 2, 3], [4, 5, 6], [-1, 7, 8]]
down --> [[1, 2, 3], [4, 5, 6], [-1, 7, 8]]
right --> [[1, 2, 3], [4, 5, 6], [7, -1, 8]]
up --> [[1, 2, 3], [4, -1, 6], [7, 5, 8]]
down --> [[1, 2, 3], [4, 5, 6], [7, -1, 8]]
right --> [[1, 2, 3], [4, 5, 6], [7, 8, -1]]
Found
Path cost-> 16


...Program finished with exit code 0
Press ENTER to exit console.
```

State Space Tree:



23/11/22 **Program 3 - 8 puzzle using IDDFS**

Algorithm:



23/11/22 8 PUZZLE - IDS

Aim - To implement 8 puzzle game using IDS algorithm.

Algorithm:

IDS(root, goal, depth limit)
for depth = 0 to depth_limit
if (DFS(root, goal, depth))
return True
return false.

Code:

```
import copy

inp=[[1,2,3],[4,5,6],[-1,7,8]]
out=[[1,2,3],[4,5,6],[7,8,-1]]

print("Enter input puzzle")
for i in range(3):
  for j in range(3):
    inp[i][j]=int(input("Enter number at "+str(i)+","+str(j)+" ->"))

def move(temp, movement):
  if movement=="up":
    for i in range(3):
      for j in range(3):
        if(temp[i][j]==-1):
          if i!=0:
```

```
          temp[i][j]=temp[i-1][j]
          temp[i-1][j]=-1
        return temp

  if movement=="down":
    for i in range(3):
      for j in range(3):
        if(temp[i][j]==-1):
          if i!=2:
            temp[i][j]=temp[i+1][j]
            temp[i+1][j]=-1
          return temp

  if movement=="left":
    for i in range(3):
      for j in range(3):
        if(temp[i][j]==-1):
          if j!=0:
            temp[i][j]=temp[i][j-1]
            temp[i][j-1]=-1
          return temp

  if movement=="right":
    for i in range(3):
      for j in range(3):
        if(temp[i][j]==-1):
          if j!=2:
            temp[i][j]=temp[i][j+1]
            temp[i][j+1]=-1
          return temp

def ids():
  global inp
  global out
  global flag

  for limit in range(100):
    print('LIMIT -> '+str(limit))
    stack=[]
    inpx=[inp,"none"]
```

```
stack.append(inpx)
level=0
while(True):
  if len(stack)==0:
    break
  puzzle=stack.pop(0)
  if level<=limit:
    print(str(puzzle[1])+" --> "+str(puzzle[0]))
    if(puzzle[0]==out):
      print("Found")
      print('Path cost='+str(level))
      flag=True
      return
    else:
      level=level+1
      if(puzzle[1]!="down"):
        temp=copy.deepcopy(puzzle[0])
        up=move(temp, "up")
        if(up!=puzzle[0]):
          upx=[up,"up"]
          stack.insert(0, upx)

      if(puzzle[1]!="right"):
        temp=copy.deepcopy(puzzle[0])
        left=move(temp, "left")
        if(left!=puzzle[0]):
          leftx=[left,"left"]
          stack.insert(0, leftx)

      if(puzzle[1]!="up"):
        temp=copy.deepcopy(puzzle[0])
        down=move(temp, "down")
        if(down!=puzzle[0]):
          downx=[down,"down"]
          stack.insert(0, downx)

      if(puzzle[1]!="left"):
        temp=copy.deepcopy(puzzle[0])
        right=move(temp, "right")
        if(right!=puzzle[0]):
```

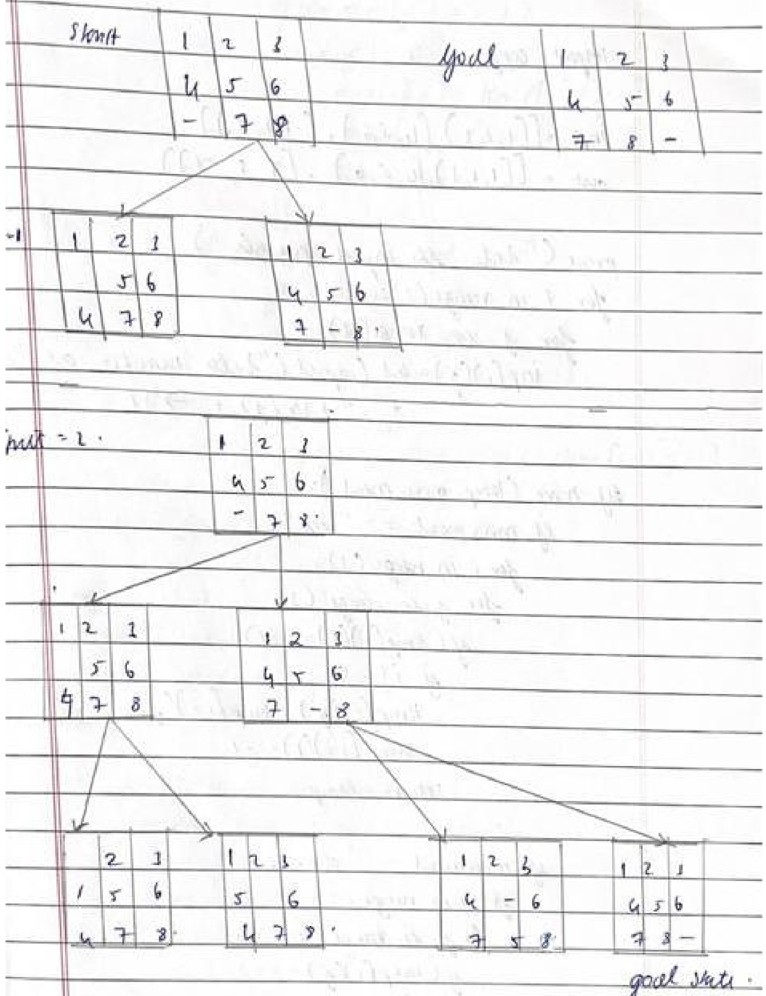
```
            rightx=[right,"right"]
            stack.insert(0, rightx)

print('~~~~~~~~~~~~ IDS ~~~~~~~~~~~~')
ids()
```

Output:

```
Enter input puzzle
Enter number at 0,0 ->1
Enter number at 0,1 ->2
Enter number at 0,2 ->3
Enter number at 1,0 ->4
Enter number at 1,1 ->5
Enter number at 1,2 ->6
Enter number at 2,0 ->-1
Enter number at 2,1 ->7
Enter number at 2,2 ->8
~~~~~~~~~~~~ IDS ~~~~~~~~~~~~
LIMIT -> 0
none --> [[1, 2, 3], [4, 5, 6], [-1, 7, 8]]
LIMIT -> 1
none --> [[1, 2, 3], [4, 5, 6], [-1, 7, 8]]
right --> [[1, 2, 3], [4, 5, 6], [7, -1, 8]]
LIMIT -> 2
none --> [[1, 2, 3], [4, 5, 6], [-1, 7, 8]]
right --> [[1, 2, 3], [4, 5, 6], [7, -1, 8]]
right --> [[1, 2, 3], [4, 5, 6], [7, 8, -1]]
Found
Path cost=2


...Program finished with exit code 0
Press ENTER to exit console.
```

State Space Tree:

State space tree



30/11/22 Program 04 - 8 Puzzle Using A* Algorithm

Algorithm:

Algorithm ::

$f(n) = g(n) + h(n)$.

$g(n)$ = sum of edge costs from start to n.

$h(n)$ = estimate of least cost path from n to goal.

$f(n)$ = actual distance so far + estimated distance remaining.

function A* search (problem) returns a solution or failure

node ← a node n with n.state = problem.initial state

frontier ← a priority Queue ordered by ascending g+h, only element n.

loop do

if empty ? (frontier) then return failure.

n ← pop (frontier)

if problem.goal Test (n.state) then return solution

for each action a in problem.actions (n.state)

do

n' ← child Node (problem, n, a)

insert (n', g(n') + h(n'), frontier)

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 \n")
        start = self.accept()
        print("Enter the goal state matrix \n")
        goal = self.accept()

        start = Node(start,0,0)
        start.fval = self.f(start,goal)
        self.open.append(start)

        while True:
            cur = self.open[0]
            print("")
            print("  | ")
            print("  | ")
```

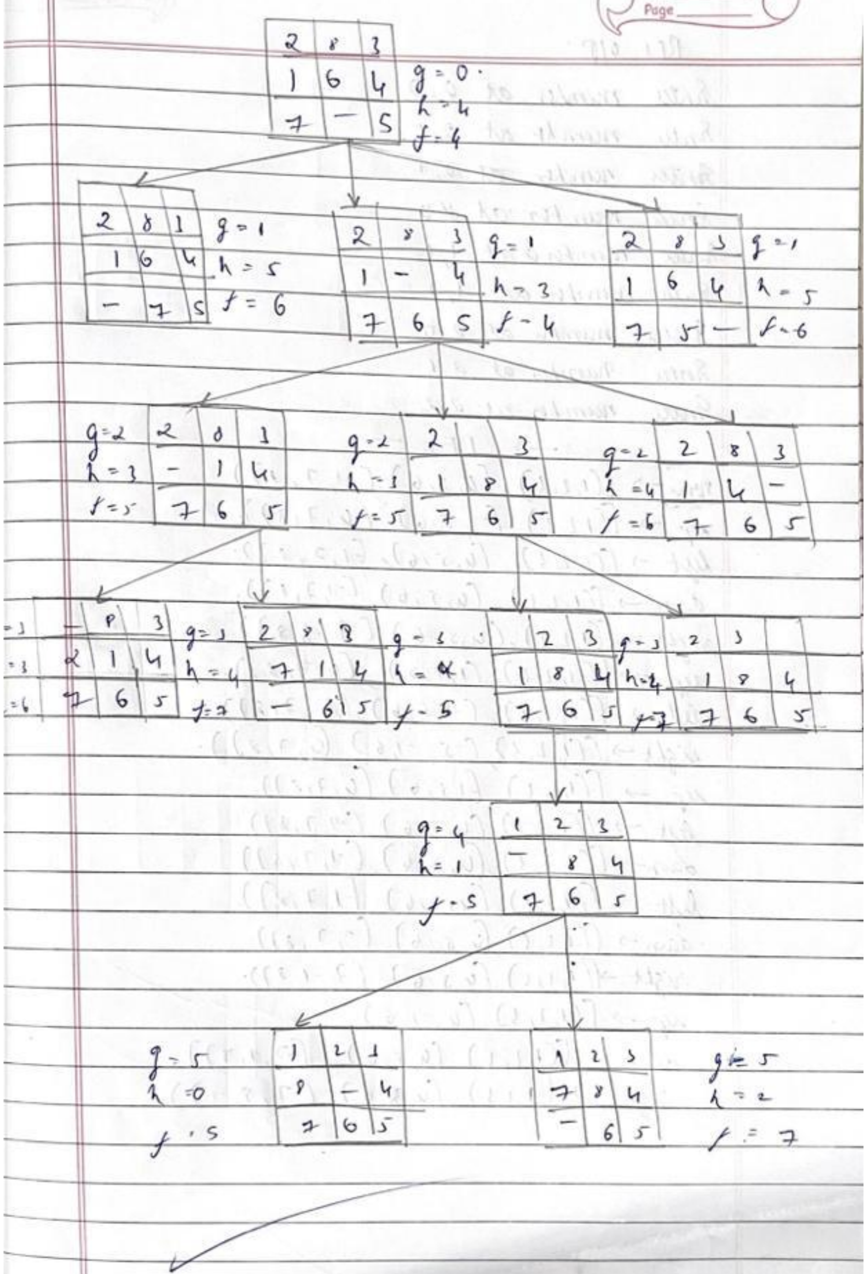
```
        print(" \\\'/ \n")
        for i in cur.data:
            for j in i:
                print(j,end=" ")
            print("")
        if(self.h(cur.data,goal) == 0):
            break
        for i in cur.generate_child():
            i.fval = self.f(i,goal)
            self.open.append(i)
        self.closed.append(cur)
        del self.open[0]

        """ sort the opne list based on f value """
        self.open.sort(key = lambda x:x.fval,reverse=False)


puz = Puzzle(3)
puz.process()
```

Output:

```
Initial state of the puzzle
1 2 5
3 4 8
6 7 0

Movement  1
1 2 5
3 4 0
6 7 8

Movement  2
1 2 0
3 4 5
6 7 8

Movement  3
1 0 2
3 4 5
6 7 8

Movement  4
0 1 2
3 4 5
6 7 8



...Program finished with exit code 0
Press ENTER to exit console.
```

State Space Tree:

State space tree:



7/12/22 **Program 5 - Vacuum Cleaner**

Algorithm:

7/12/22

Vacuum Cleaner agent

Aim: To implement Vaccum Cleaner agent program

Algorithm:

if location A = dirty
    cost + = 1
    clean A
    if location B is dirty:
        move to B
        cost + = 1
        clean B
        cost + = 1
else:
    no action

if location A is clean:
    if location B = "dirty"
        cost + = 1
        move to B
        clean B
        cost + = 1
    else
        No action.

if location B = dirty.
    cost + = 1
    clean B
    if location A is dirty
        ~~move~~ move to A.
        cost + = 1
        clean A
        cost + = 1.

if location B is clean
    if location A is dirty
        move to A
        cost + +
        clean A
        cost + = 1
    else
        No action

print (path cost).
print (goal state).

Code:

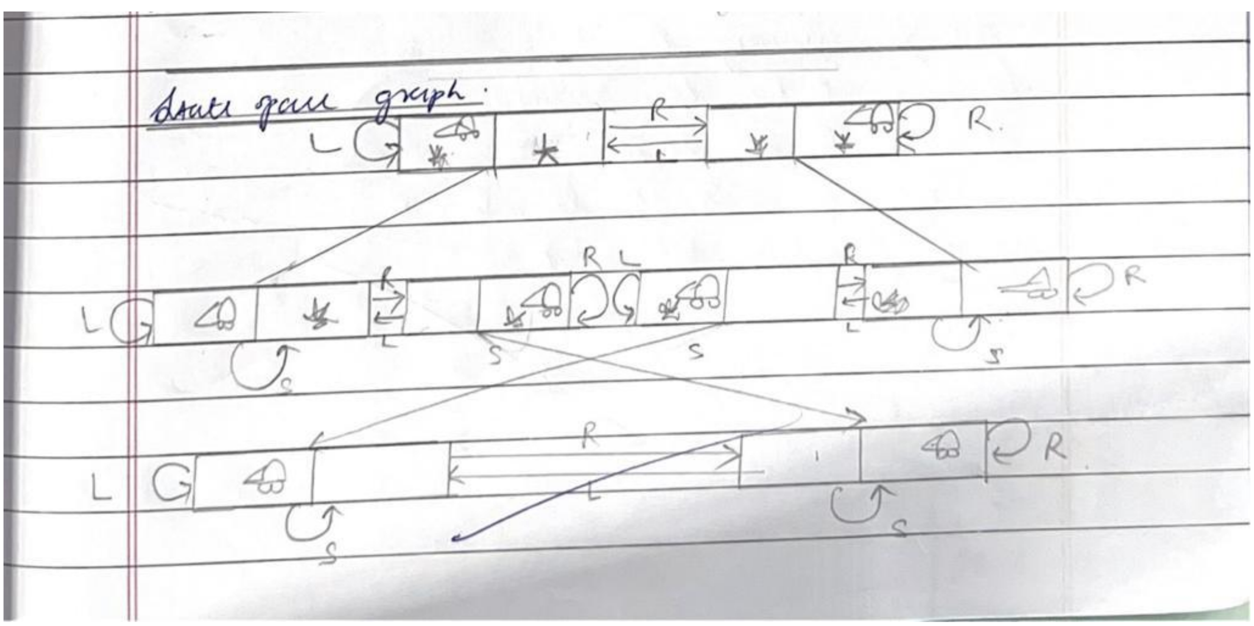
```
def vacuum_world():
    goal_state = {'A': '0', 'B': '0'}
    cost = 0
```

```
    actions = []
    location_input = input("Enter Location of Vacuum: ")
    status_input = input("Enter status of " + location_input + ": ")
    status_input_complement = input("Enter status of other room: ")
    print("Initial Location Condition" + str(goal_state))
    if location_input == 'A':
        location_complement = 'B'
    else:
        location_complement = 'A'
    if status_input == '1':
        actions.append("Suck at Location "+location_input)
        goal_state[location_input] = '0'
        cost += 1
        actions.append("Move to Location "+location_complement)
        if status_input_complement == '1':
            cost += 1
            actions.append("Suck at Location "+location_complement)
            goal_state[location_complement] = '0'
            cost += 1
    if status_input == '0':
        actions.append("Move to Location "+location_complement)
        if status_input_complement == '1':
            actions.append("Suck at Location "+location_complement)
            cost += 1
            goal_state[location_complement] = '0'
            cost += 1
    print("GOAL STATE: ")
    print(goal_state)
    print("Actions Taken are: ")
    for var in actions:
        print(var)
    print("Performance Measurement: " + str(cost))
vacuum_world()
```

Output:

```
Enter Location of Vacuum : A
Enter status of A 1
Enter status of other room : 1
Vacuum is placed in Location A
Location A is Dirty.
Cost for CLEANING A 1
Location A has been Cleaned.
Location B is Dirty.
Moving right to the Location B.
COST for moving RIGHT2
COST for SUCK 3
Location B has been Cleaned.
GOAL STATE:
{'A': '0', 'B': '0'}
Performance Measurement: 3


...Program finished with exit code 0
Press ENTER to exit console.
```

State Space Tree:



28/12/22 **Program 6 - Knowledge Base**

Algorithm:

28/12/22

Lab - 6 - Knowledge Base

Aim: Create a knowledge base using propositional logic and show that the given query entails the knowledge base or not.

Algorithm-

function TT-ENTAILS?(KB, a) returns true or false.
inputs: KB, the knowledge base, a sentence in propositional logic a, the query, a sentence in propositional logic

symbols a list of the proposition symbol in KB and a
return TT-CHECK-ALL(KB, a, symbols, { })

function TT-CHECK-ALL(KB, a, symbols, model) returns true or false

if Empty?(symbols) then
    if PL-TRUE?(KB, model) then return PL-TRUE?(a, model)
    else return true //if when KB is false.
    //always return true.
else do
    P ← FIRST(symbols) then.
    rest ← REST(symbols).
    return (TT-CHECK-ALL(KB, a, rest, model ∪ {P = TRUE})
    and
    TT-CHECK-ALL(KB, a, rest, model ∪ {P = false}))

Code:

```
combinations=[(True,True, True),(True,True,False),(True,False,True),(True,False,
False),(False,True, True),(False,True, False),(False, False,True),(False,False, False)]
variable={'p':0,'q':1, 'r':2}
kb=''
q=''
priority={'~':3,'v':1,'^':2}
def input_rules():
    global kb, q
    kb = (input("Enter rule: "))
    q = input("Enter the Query: ")
def entailment():
    global kb, q
    print('*'*10+"Truth Table Reference"+'*'*10)
    print('kb','alpha')
    print('*'*10)
    for comb in combinations:
        s = evaluatePostfix(toPostfix(kb), comb)
        f = evaluatePostfix(toPostfix(q), comb)
        print(s, f)
        print('-'*10)
        if s and not f:
            return False
    return True
def isOperand(c):
    return c.isalpha() and c!='v'

def isLeftParanthesis(c):
    return c == '('

def isRightParanthesis(c):
    return c == ')'

def isEmpty(stack):
    return len(stack) == 0

def peek(stack):
    return stack[-1]

def hasLessOrEqualPriority(c1, c2):
    try:
```

```
        return priority[c1]<=priority[c2]
    except KeyError:
        return False
def toPostfix(infix):
    stack = []
    postfix = ''
    for c in infix:
        if isOperand(c):
            postfix += c
        else:
            if isLeftParanthesis(c):
                stack.append(c)
            elif isRightParanthesis(c):
                operator = stack.pop()
                while not isLeftParanthesis(operator):
                    postfix += operator
                    operator = stack.pop()
            else:
                while (not isEmpty(stack)) and hasLessOrEqualPriority(c, peek(stack)):
                    postfix += stack.pop()
                stack.append(c)
    while (not isEmpty(stack)):
        postfix += stack.pop()

    return postfix
def evaluatePostfix(exp, comb):
    stack = []
    for i in exp:
        if isOperand(i):
            stack.append(comb[variable[i]])
        elif i == '~':
            val1 = stack.pop()
            stack.append(not val1)
        else:
            val1 = stack.pop()
            val2 = stack.pop()
            stack.append(_eval(i,val2,val1))
    return stack.pop()
def _eval(i, val1, val2):
    if i == '^':
```

```
        return val2 and val1
    return val2 or val1

input_rules()
ans = entailment()
if ans:
    print("The Knowledge Base entails query")
else:
    print("The Knowledge Base does not entail query")
```

OUTPUT:

```
Enter the kb:
~Pv~QvR P ~SvTvQ S
Enter the query:
R

Step    |Clause |Derivation
------------------------------
 1.     | ~Pv~QvR       | Given.
 2.     | P     | Given.
 3.     | ~SvTvQ        | Given.
 4.     | S     | Given.
 5.     | ~R    | Negated conclusion.
 6.     | ~QvR  | Resolved from ~Pv~QvR and P.
 7.     | ~Pv~Q | Resolved from ~Pv~QvR and ~R.
 8.     | ~Q    | Resolved from P and ~Pv~Q.
 9.     | Q     | Resolved from ~SvTvQ and S.
 10.    | ~SvR  | Resolved from ~SvTvQ and ~QvR.
 11.    | ~Sv~P | Resolved from ~SvTvQ and ~Pv~Q.
 12.    | ~S    | Resolved from ~SvTvQ and ~Q.
 13.    | R     | Resolved from S and ~SvR.
 14.    | ~P    | Resolved from S and ~Sv~P.
 15.    |       | Resolved ~R and R to ~RvR, which is in turn null.
A contradiction is found when ~R is assumed as true. Hence, R is true.


...Program finished with exit code 0
Press ENTER to exit console.
```

4/1/22 **Program 7 - Knowledge Base Resolution**

Algorithm:

4/1/23 Lab - 7 - Proof by Resolution.

Aim: To create a knowledge base using propositional logic and prove the given query using resolution

Algorithm:

function PL-Resolution (KB, a) returns true or false
inputs: KB, the knowledge base, a sentence in propositional logic a,
the query, a sentence in propositional logic

clauses ← the set of clauses in the CNF representation of KB ∧ ¬a new {}
loop do
for each pair of clauses C_i, C_j in clauses do
resolvents ← PL_RESOLVE (C_i, C_j)
if resolvents contains the empty clause
then return true
new ← new ∪ resolvents.
if new ⊆ clauses then return false
clauses ← clauses ∪ new.

Code:

Code:

```
kb = []

# Reset kb to an empty list
def CLEAR():
    global kb
    kb = []

# Insert sentence to the kb
def TELL(sentence):
    global kb
    # If the sentence is a clause, insert directly.
    if isClause(sentence):
        kb.append(sentence)
    # If not, convert to CNF, and then insert clauses one by one.
    else:
        sentenceCNF = convertCNF(sentence)
        if not sentenceCNF:
            print("Illegal input")
            return
        # Insert clauses one by one when there are multiple clauses
        if isAndList(sentenceCNF):
            for s in sentenceCNF[1:]:
                kb.append(s)
        else:
            kb.append(sentenceCNF)

# 'ASK' the kb whether a sentence is True or not
def ASK(sentence):
    global kb

    # Negate the sentence, and convert it to CNF accordingly.
    if isClause(sentence):
        neg = negation(sentence)
    else:
        sentenceCNF = convertCNF(sentence)
        if not sentenceCNF:
            print("Illegal input")
            return
        neg = convertCNF(negation(sentenceCNF))
```

```
    # Insert individual clauses that we need to ask to ask_list.
    ask_list = []
    if isAndList(neg):
        for n in neg[1:]:
            nCNF = makeCNF(n)
            if type(nCNF).__name__ == 'list':
                ask_list.insert(0, nCNF)
            else:
                ask_list.insert(0, nCNF)
    else:
        ask_list = [neg]
# Create a new list combining the asked sentence and kb.
    # Resolution will happen between the items in the list.
    clauses = ask_list + kb[:]

    # Recursivly conduct resoltion between items in the clauses list
    # until it produces an empty list or there's no more pregress.
    while True:
        new_clauses = []
        for c1 in clauses:
            for c2 in clauses:
                if c1 is not c2:
                    resolved = resolve(c1, c2)
                    if resolved == False:
                        continue
                    if resolved == []:
                        return True
                    new_clauses.append(resolved)

        if len(new_clauses) == 0:
            return False

        new_in_clauses = True
        for n in new_clauses:
            if n not in clauses:
                new_in_clauses = False
                clauses.append(n)

        if new_in_clauses:
            return False
```

```
    return False

# Conduct resolution on two CNF clauses.
def resolve(arg_one, arg_two):
    resolved = False

    s1 = make_sentence(arg_one)
    s2 = make_sentence(arg_two)

    resolve_s1 = None
    resolve_s2 = None

    # Two for loops that iterate through the two clauses.
    for i in s1:
        if isNotList(i):
            a1 = i[1]
            a1_not = True
        else:
            a1 = i
            a1_not = False

        for j in s2:
            if isNotList(j):
                a2 = j[1]
                a2_not = True
            else:
                a2 = j
                a2_not = False

            # cancel out two literals such as 'a' $ ['not', 'a']
            if a1 == a2:
                if a1_not != a2_not:
                    # Return False if resolution already happend
                    # but contradiction still exists.
                    if resolved:
                        return False
                    else:
                        resolved = True
                        resolve_s1 = i
                        resolve_s2 = j
```

```
                    break
                # Return False if not resolution happened
    if not resolved:
        return False

    # Remove the literals that are canceled
    s1.remove(resolve_s1)
    s2.remove(resolve_s2)

    # # Remove duplicates
    result = clear_duplicate(s1 + s2)

    # Format the result.
    if len(result) == 1:
        return result[0]
    elif len(result) > 1:
        result.insert(0, 'or')

    return result

# Prepare sentences for resolution.
def make_sentence(arg):
    if isLiteral(arg) or isNotList(arg):
        return [arg]
    if isOrList(arg):
        return clear_duplicate(arg[1:])
    return

# Clear out duplicates in a sentence.
def clear_duplicate(arg):
    result = []
    for i in range(0, len(arg)):
        if arg[i] not in arg[i+1:]:
            result.append(arg[i])
    return result

# Check whether a sentence is a legal CNF clause.
def isClause(sentence):
    if isLiteral(sentence):
        return True
```

```
    if isNotList(sentence):
        if isLiteral(sentence[1]):
            return True
        else:
            return False
    if isOrList(sentence):
        for i in range(1, len(sentence)):
            if len(sentence[i]) > 2:
                return False
            elif not isClause(sentence[i]):
                return False
        return True
    return False

# Check if a sentence is a legal CNF.
def isCNF(sentence):
    if isClause(sentence):
        return True
    elif isAndList(sentence):
        for s in sentence[1:]:
            if not isClause(s):
                return False
        return True
    return False

# Negate a sentence.
def negation(sentence):
    if isLiteral(sentence):
        return ['not', sentence]
    if isNotList(sentence):
        return sentence[1]

    # DeMorgan:
    if isAndList(sentence):
        result = ['or']
        for i in sentence[1:]:
            if isNotList(sentence):
                result.append(i[1])
            else:
                result.append(['not', sentence])
```

```
        return result
    if isOrList(sentence):
        result = ['and']
        for i in sentence[:]:
            if isNotList(sentence):
                result.append(i[1])
            else:
                result.append(['not', i])
        return result
    return None


# Convert a sentence into CNF.
def convertCNF(sentence):
    while not isCNF(sentence):
        if sentence is None:
            return None
        sentence = makeCNF(sentence)
    return sentence

# Help make a sentence into CNF.
def makeCNF(sentence):
    if isLiteral(sentence):
        return sentence

    if (type(sentence).__name__ == 'list'):
        operand = sentence[0]
        if isNotList(sentence):
            if isLiteral(sentence[1]):
                return sentence
            cnf = makeCNF(sentence[1])
            if cnf[0] == 'not':
                return makeCNF(cnf[1])
            if cnf[0] == 'or':
                result = ['and']
                for i in range(1, len(cnf)):
                    result.append(makeCNF(['not', cnf[i]]))
                return result
            if cnf[0] == 'and':
                result = ['or']
```

```
        for i in range(1, len(cnf)):
            result.append(makeCNF(['not', cnf[i]]))
        return result
    return "False: not"

# Implication Elimination:
if operand == 'implies' and len(sentence) == 3:
    return makeCNF(['or', ['not', makeCNF(sentence[1])], makeCNF(sentence[2])])
    # Biconditional Elimination:
if operand == 'biconditional' and len(sentence) == 3:
    s1 = makeCNF(['implies', sentence[1], sentence[2]])
    s2 = makeCNF(['implies', sentence[2], sentence[1]])
    return makeCNF(['and', s1, s2])

if isAndList(sentence):
    result = ['and']
    for i in range(1, len(sentence)):
        cnf = makeCNF(sentence[i])
        # Distributivity:
        if isAndList(cnf):
            for i in range(1, len(cnf)):
                result.append(makeCNF(cnf[i]))
            continue
        result.append(makeCNF(cnf))
    return result

if isOrList(sentence):
    result1 = ['or']
    for i in range(1, len(sentence)):
        cnf = makeCNF(sentence[i])
        # Distributivity:
        if isOrList(cnf):
            for i in range(1, len(cnf)):
                result1.append(makeCNF(cnf[i]))
            continue
        result1.append(makeCNF(cnf))
        # Associativity:
    while True:
        result2 = ['and']
        and_clause = None
```

```
            for r in result1:
                if isAndList(r):
                    and_clause = r
                    break

            # Finish when there's no more 'and' lists
            # inside of 'or' lists
            if not and_clause:
                return result1

            result1.remove(and_clause)

            for i in range(1, len(and_clause)):
                temp = ['or', and_clause[i]]
                for o in result1[1:]:
                    temp.append(makeCNF(o))
                    result2.append(makeCNF(temp))
            result1 = makeCNF(result2)
        return None
    return None

# Below are 4 functions that check the type of a variable
def isLiteral(item):
    if type(item).__name__ == 'str':
        return True
    return False


def isNotList(item):
    if type(item).__name__ == 'list':
        if len(item) == 2:
            if item[0] == 'not':
                return True
    return False


def isAndList(item):
    if type(item).__name__ == 'list':
        if len(item) > 2:
            if item[0] == 'and':
```

```
            return True
    return False


def isOrList(item):
    if type(item).__name__ == 'list':
        if len(item) > 2:
            if item[0] == 'or':
                return True
    return False

if __name__ == "__main__":
    CLEAR()

    print("Test 1")
    TELL(['implies', 'p', 'q'])
    TELL(['implies', 'r', 's'])
    ASK(['implies',['or','p','r'], ['or', 'q', 's']])

    CLEAR()

    print("Test 2")
    TELL('p')
    TELL(['implies',['and','p','q'],'r'])
    TELL(['implies',['or','s','t'],'q'])
    TELL('t')
    ASK('r')

    CLEAR()

    print("Test 3")
    TELL('a')
    TELL('b')
    TELL('c')
    TELL('d')
    ASK(['or', 'a', 'b', 'c', 'd'])

    CLEAR()

    print("Test 4")
```

```
TELL('a')
TELL('b')
TELL(['or', ['not', 'a'], 'b'])
TELL(['or', 'c', 'd'])
TELL('d')
ASK('c')
```

Output:

```
Test 1
True
Test 2
True
Test 3
True
Test 4
False
```

11/1/23 Program 8 - Unification in First Order Logic

Algorithm:

11/1/23 Lab 8 - Proof by Unification:

Aim: To implement unification in First order logic

```
import re

def getAttribute (expression):
        expression = expression.split ("(") [1:]
        expression = "(".join (expression)
        expression = expression.split (")") [:-1]
        expression = ")".join (expression)
        attributes = expression.split (',')
        return attributes

def getInitialPredicate (expression):
        return expression.split("(")[0]

def isConstant(char):
        return char.isupper() and len(char) == 1

def isVariable(char):
        return char.islower() and len(char) == 1

def replaceAttributes(exp, old, new):
    attributes = getAttributes (exp)
    predicate = getInitialPredicate (exp)
    for index, val in enumerate (attributes):
        if val == old:
            attributes [index] = new
    return predicate + "(" + ",".join(attributes) + ")"

def apply(exp, substitutions):
    for substitution in substitutions:
```

Code:

```
import re
def getAttributes(expression):
```

```
    expression = expression.split("(")[1:]
    expression = "(".join(expression)
    expression = expression.split(")")[:-1]
    expression = ")".join(expression)
    attributes = expression.split(',')
    return attributes

def getInitialPredicate(expression):
    return expression.split("(")[0]
def isConstant(char):
    return char.isupper() and len(char) == 1

def isVariable(char):
    return char.islower() and len(char) == 1
def replaceAttributes(exp, old, new):
    attributes = getAttributes(exp)
    predicate = getInitialPredicate(exp)
    for index, val in enumerate(attributes):
        if val == old:
            attributes[index] = new
    return predicate + "(" + ",".join(attributes) + ")"

def apply(exp, substitutions):
    for substitution in substitutions:
        new, old = substitution
        exp = replaceAttributes(exp, old, new)
    return exp
def checkOccurs(var, exp):
    if exp.find(var) == -1:
        return False
    return True


def getFirstPart(expression):
    attributes = getAttributes(expression)
    return attributes[0]


def getRemainingPart(expression):
    predicate = getInitialPredicate(expression)
```

```
    attributes = getAttributes(expression)
    newExpression = predicate + "(" + ",".join(attributes[1:]) + ")"
    return newExpression
def unify(exp1, exp2):
    if exp1 == exp2:
        return []

    if isConstant(exp1) and isConstant(exp2):
        if exp1 != exp2:
            print(f"{exp1} and {exp2} are constants. Cannot be unified")
            return []

    if isConstant(exp1):
        return [(exp1, exp2)]

    if isConstant(exp2):
        return [(exp2, exp1)]

    if isVariable(exp1):
        return [(exp2, exp1)] if not checkOccurs(exp1, exp2) else []

    if isVariable(exp2):
        return [(exp1, exp2)] if not checkOccurs(exp2, exp1) else []

    if getInitialPredicate(exp1) != getInitialPredicate(exp2):
        print("Cannot be unified as the predicates do not match!")
        return []

    attributeCount1 = len(getAttributes(exp1))
    attributeCount2 = len(getAttributes(exp2))
    if attributeCount1 != attributeCount2:
        print(f"Length of attributes {attributeCount1} and {attributeCount2} do not match. Cannot
be unified")
        return []

    head1 = getFirstPart(exp1)
    head2 = getFirstPart(exp2)
    initialSubstitution = unify(head1, head2)
    if not initialSubstitution:
        return []
```

```
    if attributeCount1 == 1:
        return initialSubstitution

    tail1 = getRemainingPart(exp1)
    tail2 = getRemainingPart(exp2)

    if initialSubstitution != []:
        tail1 = apply(tail1, initialSubstitution)
        tail2 = apply(tail2, initialSubstitution)

    remainingSubstitution = unify(tail1, tail2)
    if not remainingSubstitution:
        return []

    return initialSubstitution + remainingSubstitution
def main():
    print("Enter the first expression")
    e1 = input()
    print("Enter the second expression")
    e2 = input()
    substitutions = unify(e1, e2)
    print("The substitutions are:")
    print([' / '.join(substitution) for substitution in substitutions])
main()
```

Output:

11/1/23 Lab 8 - Proof by Unification:

Aim: To implement unification in First order logic

```
import re

def getAttribute (expression):
    expression = expression.split ("(") [1:]
    expression = "(".join (expression)
    expression = expression.split (")") [:-1]
    expression = ")".join (expression)
    attributes = expression.split (',')
    return attributes

def getInitialPredicate (expression):
    return expression.split("(")[0]

def isConstant (char):
    return char.isupper() and len(char) == 1

def isVariable (char):
    return char.islower() and len(char) == 1

def replaceAttributes (exp, old, new):
    attributes = getAttributes (exp)
    predicate = getInitialPredicate (exp).
    for index, val in enumerate (attributes):
        if val == old:
            attributes [index] = new
    return predicate + "(" + ",".join(attributes) + ")"

def apply (exp, substitutions):
    for substitution in substitutions:
```

18/1/23 Program 9 - First Order Logic to Conjunctive Normal Form

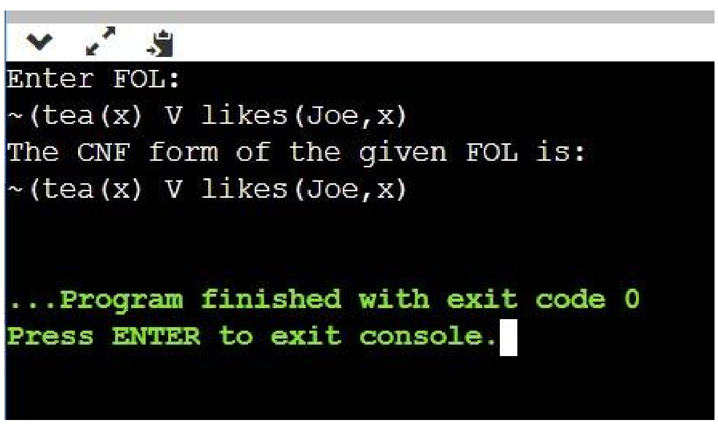
Code:

```
import re
def getAttributes(string):
    expr = '\([^)]+\)'
    matches = re.findall(expr, string)
    return [m for m in str(matches) if m.isalpha()]
def getPredicates(string):
    expr = '[a-z~]+\([A-Za-z,]+\)'
    return re.findall(expr, string)
def DeMorgan(sentence):
    string = ''.join(list(sentence).copy())
    string = string.replace('~~','')
    flag = '[' in string
    string = string.replace('~[','')
    string = string.strip(']')
    for predicate in getPredicates(string):
        string = string.replace(predicate, f'~{predicate}')
    s = list(string)
    for i, c in enumerate(string):
        if c == 'V':
          s[i] = '^'
        elif c == '^':
s[i] = 'V'
    string = ''.join(s)
    string = string.replace('~~','')
    return f'[{string}]' if flag else string
def Skolemization(sentence):
    SKOLEM_CONSTANTS = [f'{chr(c)}' for c in range(ord('A'), ord('Z')+1)]
    statement = ''.join(list(sentence).copy())
    matches = re.findall('[∀∃].', statement)
    for match in matches[::-1]:
        statement = statement.replace(match, '')
        statements = re.findall('\[\[[^]]+\]]', statement)
        for s in statements:
            statement = statement.replace(s, s[1:-1])
        for predicate in getPredicates(statement):
```

```
            attributes = getAttributes(predicate)
            if ''.join(attributes).islower():
                statement = statement.replace(match[1],SKOLEM_CONSTANTS.pop(0))
            else:
                aL = [a for a in attributes if a.islower()]
                aU = [a for a in attributes if not a.islower()][0]
                statement = statement.replace(aU, f'{SKOLEM_CONSTANTS.pop(0)}({aL[0] if
len(aL) else match[1]})')
    return statement
def fol_to_cnf(fol):

    statement = fol.replace("<=>", "_")
    while '_' in statement:
        i = statement.index('_')
        new_statement = '[' + statement[:i] + '=>' + statement[i+1:] + ']^['+ statement[i+1:] + '=>' +
statement[:i] + ']'
        statement = new_statement
    statement = statement.replace("=>", "-")
    expr = '\[([^]]+)\]'
    statements = re.findall(expr, statement)
    for i, s in enumerate(statements):
        if '[' in s and ']' not in s:
            statements[i] += ']'
    for s in statements:
        statement = statement.replace(s, fol_to_cnf(s))
    while '-' in statement:
        i = statement.index('-')
        br = statement.index('[') if '[' in statement else 0
        new_statement = '~' + statement[br:i] + 'V' + statement[i+1:]
        statement = statement[:br] + new_statement if br > 0 else new_statement
    while '~∀' in statement:
        i = statement.index('~∀')
        statement = list(statement)
        statement[i], statement[i+1], statement[i+2] = '∃', statement[i+2], '~'
        statement = ''.join(statement)
    while '~∃' in statement:
        i = statement.index('~∃')
        s = list(statement)
        s[i], s[i+1], s[i+2] = '∀', s[i+2], '~'
        statement = ''.join(s)
```

```
    statement = statement.replace('~[∀','[~∀')
    statement = statement.replace('~[∃','[~∃')
    expr = '(~[∀V∃].)'
    statements = re.findall(expr, statement)
    for s in statements:
        statement = statement.replace(s, fol_to_cnf(s))
    expr = '~\[[^]]+\]'
    statements = re.findall(expr, statement)
    for s in statements:
        statement = statement.replace(s, DeMorgan(s))
    return statement
def main():
    print("Enter FOL:")
    fol = input()
    print("The CNF form of the given FOL is: ")
    print(Skolemization(fol_to_cnf(fol)))
main()
```

Output :



18/1/23 **Program 10 - Forward Reasoning**

Algorithm:

classmate
Date
Page

18/1/23 Lab-10 Forward Chaining.

Aim – To create a knowledge base consisting of First Order logic statements and prove given query using forward chaining and forward reasoning.

Algorithm:

function: FOL-FC-ASK (KB, α) return a substitution or false.

inputs: KB, the knowledge base, a set of first order definite clauses α, the query, an atomic sentence.

local variables: new, the new sentences inferred on each iteration

Repeat until new is empty.
new ← { }
for each rule in KB do
$(p_1 \wedge \ldots \wedge p_n \Rightarrow q) \leftarrow$ STANDARDIZE_VARIABLE (rule).
for each θ such that SUBST(θ, $p_1 \ldots \wedge p_n$)
= SUBST(θ $p_1' \wedge \ldots \wedge p_n'$)
for some p_i' in KB.
$q' \leftarrow$ SUBST(θ, q)
if q' does not unify with some sentence already in KB or new then
add q' to new.
$\phi \leftarrow$ Unify (q', α).
if ϕ is not fail then return ϕ.

add new to KB.
return false.

Code:

```
import re

def isVariable(x):
    return len(x) == 1 and x.islower() and x.isalpha()

def getAttributes(string):
    expr = '\([^)]+\)'
    matches = re.findall(expr, string)
    return matches

def getPredicates(string):
    expr = '([a-z~]+)\([^&|]+\)'
    return re.findall(expr, string)
class Fact:
    def __init__(self, expression):
        self.expression = expression
        predicate, params = self.splitExpression(expression)
        self.predicate = predicate
        self.params = params
        self.result = any(self.getConstants())

    def splitExpression(self, expression):
        predicate = getPredicates(expression)[0]
        params = getAttributes(expression)[0].strip('()').split(',')
        return [predicate, params]

    def getResult(self):
        return self.result

    def getConstants(self):
        return [None if isVariable(c) else c for c in self.params]

    def getVariables(self):
        return [v if isVariable(v) else None for v in self.params]

    def substitute(self, constants):
        c = constants.copy()
        f = f"{self.predicate}({','.join([constants.pop(0) if isVariable(p) else p for p in
self.params])})"
        return Fact(f)
```

```
class Implication:
    def __init__(self, expression):
        self.expression = expression
        l = expression.split('=>')
        self.lhs = [Fact(f) for f in l[0].split('&')]
        self.rhs = Fact(l[1])

    def evaluate(self, facts):
        constants = {}
        new_lhs = []
        for fact in facts:
            for val in self.lhs:
                if val.predicate == fact.predicate:
                    for i, v in enumerate(val.getVariables()):
                        if v:
                            constants[v] = fact.getConstants()[i]
                    new_lhs.append(fact)
        predicate, attributes = getPredicates(self.rhs.expression)[0],
str(getAttributes(self.rhs.expression)[0])
        for key in constants:
            if constants[key]:
                attributes = attributes.replace(key, constants[key])
        expr = f'{predicate}{attributes}'
        return Fact(expr) if len(new_lhs) and all([f.getResult() for f in new_lhs]) else None
class KB:
    def __init__(self):
        self.facts = set()
        self.implications = set()

    def tell(self, e):
        if '=>' in e:
            self.implications.add(Implication(e))
        else:
            self.facts.add(Fact(e))
        for i in self.implications:
            res = i.evaluate(self.facts)
            if res:
                self.facts.add(res)

    def query(self, e):
```

```
        facts = set([f.expression for f in self.facts])
        i = 1
        print(f'Querying {e}:')
        for f in facts:
            if Fact(f).predicate == Fact(e).predicate:
                print(f'\t{i}. {f}')
                i += 1

    def display(self):
        print("All facts: ")
        for i, f in enumerate(set([f.expression for f in self.facts])):
            print(f'\t{i+1}. {f}')
def main():
    kb = KB()
    print("Enter KB: (enter e to exit)")
    while True:
        t = input()
        if(t == 'e'):
            break
        kb.tell(t)
    print("Enter Query:")
    q = input()
    kb.query(q)
    kb.display()
main()
```

Output:

```
Enter KB: (enter e to exit)
missile(x)=>weapon(x)
missile(m1)
enemy(x,america)=>hostile(x)
american(west)
enemy(china,america)
owns(china,m1)
missile(x)&owns(china,x)=>sells(west,x,china)
american(x)&weapon(y)&sells(x,y,z)&hostile(z)=>criminal(x)
e
Enter Query:
criminal(x)
Querying criminal(x):
        1. criminal(west)
All facts:
        1. weapon(m1)
        2. american(west)
        3. sells(west,m1,china)
        4. criminal(west)
        5. owns(china,m1)
        6. enemy(china,america)
        7. hostile(china)
        8. missile(m1)


...Program finished with exit code 0
Press ENTER to exit console.
```