

VISVESVARAYA TECHNOLOGICAL UNIVERSITY

“JnanaSangama”, Belgaum -590014, Karnataka.



LAB REPORT on

ARTIFICIAL INTELLIGENCE LAB

Submitted by

RACHANA A (1BM21CS154)

in partial fulfillment for the award of the degree of
BACHELOR OF ENGINEERING
in
COMPUTER SCIENCE AND ENGINEERING



B.M.S. COLLEGE OF ENGINEERING
(Autonomous Institution under VTU)
BENGALURU-560019
October-2023 to Feb-2024

**B. M. S. College of Engineering,
Bull Temple Road, Bangalore 560019**
(Affiliated To Visvesvaraya Technological University, Belgaum)
Department of Computer Science and Engineering



CERTIFICATE

This is to certify that the Lab work entitled “Artificial intelligence” carried out by **RACHANA A (1BM21CS154)**, who is bonafide student of **B. M. S. College of Engineering**. It is in partial fulfillment for the award of **Bachelor of Engineering in Computer Science and Engineering** of the Visvesvaraya Technological University, Belgaum during the year 2023-24. The Lab report has been approved as it satisfies the academic requirements in respect of a **Artificial intelligence course (22CS5PCAIN)**work prescribed for the said degree.

Dr. K. Panimozhi

Assistant Professor
Department of CSE
BMSCE, Bengaluru

Dr. Jyothi S Nayak

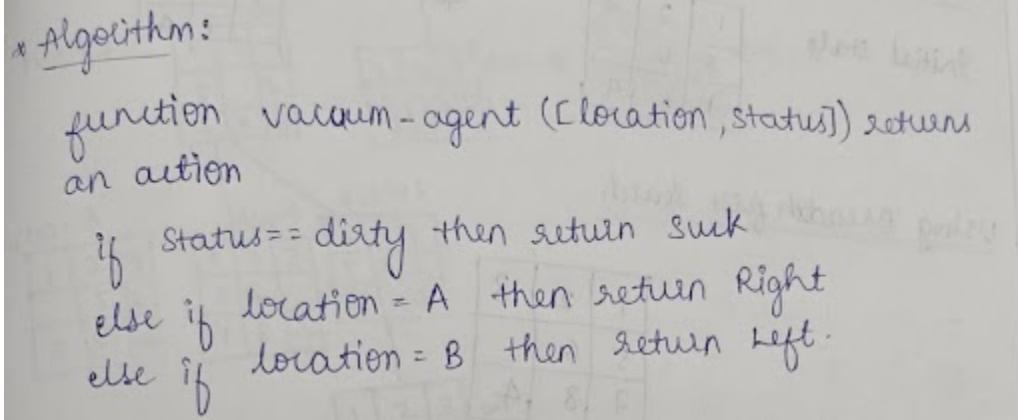
Professor and Head
Department of CSE
BMSCE, Bengaluru

PROGRAM-1

Title: Vacuum Cleaner Agent

Date:20-11-2023

Algorithm:



The image shows a handwritten algorithm for a vacuum cleaner agent. It starts with a header 'Algorithm:' followed by a function definition 'function vacuum-agent ([location, status]) returns an action'. Inside the function, there is a conditional block: 'if status == dirty then return suck', 'else if location = A then return Right', and 'else if location = B then return Left'.

```
* Algorithm:  
function vacuum-agent ([location, status]) returns  
an action  
    if status == dirty then return suck  
    else if location = A then return Right  
    else if location = B then return Left
```

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:

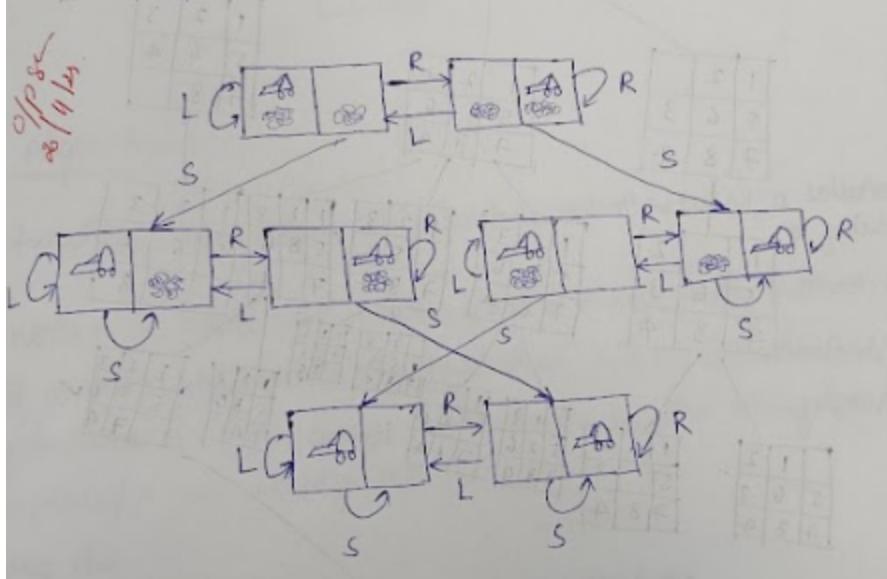
```

Enter Location of Vacuum: A
Enter status of A: 1
Enter status of other room: 0
Initial Location Condition{'A': '0', 'B': '0'}
GOAL STATE:
{'A': '0', 'B': '0'}
Actions Taken are:
Suck at Location A
Move to Location B
Performance Measurement: 1
> |

```

State space tree:

* State Space Tree:



PROGRAM-2

Title:TIC_TAC_TOE using MIN-MAX algorithm

Date:20-11-2023

Algorithm:

```
Algorithm:  
minmax (state, depth, player)  
if (player= max) then  
    best = { null, -infinity }  
else  
    best = { null, +infinity }  
if (depth=0 or gameover) then  
    score = evaluate this.state for player  
    return {null, score}  
for each valid move m for player in state s  
do  
    execute move m on s  
    [move, score] = minmax (s, depth-1, -player)  
    undo move m on s  
    if (player= max) then  
        if score > best.score then best = {move, score}  
    else  
        if score > best.score then best = {move, score}  
return best  
end
```

CODE:

```
board = [[" ", " ", " "], [" ", " ", " "], [" ", " ", " "]]  
print("0,0|0,1|0,2")  
print("1,0|1,1|1,2")  
print("2,0|2,1|2,2 \n\n")  
def print_board():  
    for row in board:  
        print("|".join(row))  
        print("-" * 5)  
  
def check_winner(player):  
    for i in range(3):  
        if all([board[i][j] == player for j in range(3)]) or all([board[j][i] == player for j in range(3)]):  
            return True  
  
    if all([board[i][i] == player for i in range(3)]) or all([board[i][2 - i] == player for i in range(3)]):  
        return True
```

```

return False

def is_full():
    return all([cell != " " for row in board for cell in row])

def minimax(depth, is_maximizing):
    if check_winner("X"):
        return -1
    if check_winner("O"):
        return 1
    if is_full():
        return 0
    if is_maximizing:
        max_eval = float("-inf")
        for i in range(3):
            for j in range(3):
                if board[i][j] == " ":
                    board[i][j] = "O"
                    eval = minimax(depth + 1, False)
                    board[i][j] = " "
                    max_eval = max(max_eval, eval)
        return max_eval
    else:
        min_eval = float("inf")
        for i in range(3):
            for j in range(3):
                if board[i][j] == " ":
                    board[i][j] = "X"
                    eval = minimax(depth + 1, True)
                    board[i][j] = " "
                    min_eval = min(min_eval, eval)
        return min_eval

def ai_move():
    best_move = None
    best_eval = float("-inf")
    for i in range(3):
        for j in range(3):
            if board[i][j] == " ":

```

```

board[i][j] = "O"
eval = minimax(0, False)
board[i][j] = " "
if eval > best_eval:
    best_eval = eval
    best_move = (i, j)

return best_move

while not is_full() and not check_winner("X") and not check_winner("O"):
    print_board()
    row = int(input("Enter row (0, 1, or 2): "))
    col = int(input("Enter column (0, 1, or 2): "))
    if board[row][col] == " ":
        board[row][col] = "X"
        if check_winner("X"):
            print_board()
            print("You win!")
            break
    if is_full():
        print_board()
        print("It's a draw!")
        break
    ai_row, ai_col = ai_move()
    board[ai_row][ai_col] = "O"
    if check_winner("O"):
        print_board()
        print("AI wins!")
        break

else:
    print("Cell is already occupied. Try again.")
```

OUTPUT:

```
0,0|0,1|0,2  
1,0|1,1|1,2  
2,0|,2,1|2,2
```

```
| |  
-----  
| |  
-----  
| |  
-----
```

```
Enter row (0, 1, or 2): 1  
Enter column (0, 1, or 2): 1  
o| |
```

```
-----  
|x|  
-----  
| |  
-----
```

```
Enter row (0, 1, or 2): 0  
Enter column (0, 1, or 2): 2  
o| |x
```

```
-----  
|x|  
-----  
o| |
```

```
Enter row (0, 1, or 2): 1  
Enter column (0, 1, or 2): 0  
o| |x
```

```
-----  
x|x|o  
-----  
o| |
```

Enter row (0, 1, or 2): 2
Enter column (0, 1, or 2): 1
0|0|X

X|X|0

0|X|

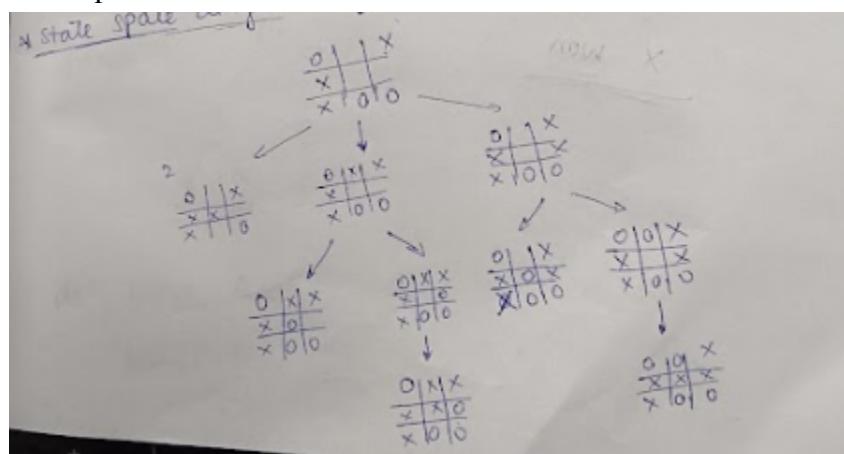
Enter row (0, 1, or 2): 2
Enter column (0, 1, or 2): 2
0|0|X

X|X|0

0|X|X

It's a draw!

State Space Tree:



PROGRAM-3

Title: 8-Puzzle problem using breadth-first search

Date: 11-12-2023

Algorithm:

The image shows a handwritten algorithm for Breadth-first search (BFS) for an 8-Puzzle problem. The algorithm is structured as follows:

```
Algorithm:
function Breadth-first-search(problem) returns a solution or failure
    node ← a node with state = problem.initial_state, pathcost = 0
    if problem.goal-test(node.state) then return Solution(node)
    frontier ← a FIFO queue with node as the only element
    explored ← an empty set
    loop do
        if Empty? (frontier) then return failure
        node ← pop (frontier)
        add node.state to explored
        for each action in problem.Action (node.state) do
            child ← child-node (problem, node, action)
            if child.state is not in explored or frontier then
                if problem.goal-test (child.state) then return
                    Solution
                        (child)
```

Code:

```
import numpy as np
import pandas as pd
import os

def gen(state, m, b):
    temp = state.copy()

    if m == 'd':
        temp[b + 3], temp[b] = temp[b], temp[b + 3]
    elif m == 'u':
        temp[b - 3], temp[b] = temp[b], temp[b - 3]
    elif m == 'l':
        temp[b - 1], temp[b] = temp[b], temp[b - 1]
    elif m == 'r':
        temp[b + 1], temp[b] = temp[b], temp[b + 1]
```

```

return temp # Return the modified state

def possible_moves(state, visited_states):
    b = state.index(0)
    d = []

    if b not in [0, 1, 2]:
        d.append('u')
    if b not in [6, 7, 8]:
        d.append('d')
    if b not in [0, 3, 6]:
        d.append('l')
    if b not in [2, 5, 8]:
        d.append('r')

    pos_moves_it_can = []

    for i in d:
        pos_moves_it_can.append(gen(state, i, b))

    return [move_it_can for move_it_can in pos_moves_it_can if move_it_can not in
visited_states]

def bfs(src, target):
    queue = []
    queue.append(src)
    cost=0

    exp = []

    while len(queue) > 0:
        source = queue.pop(0)
        exp.append(source)
        print("queue")
        for q in queue:
            print(q)
            print("-----***-----")

        print(source[0],"|",source[1],"|",source[2])
        print(source[3],"|",source[4],"|", source[5])

```

```
print(source[6],"|", source[7],"|",source[8])
print()
cost=cost+1

if source == target:
    print("success")
    print("path cost",cost)
    return

poss_moves_to_do = possible_moves(source, exp)

for move in poss_moves_to_do:
    if move not in exp and move not in queue:
        queue.append(move)

src = [1, 2, 3, 4, 5, 6, 0, 7, 8]
target = [1, 2, 3, 4, 5, 6, 7, 8, 0]
bfs(src, target)
```

OUTPUT:

1	2	3
5	6	0
7	8	4

1	2	0
5	6	3
7	8	4

1	2	3
5	6	4
7	8	0

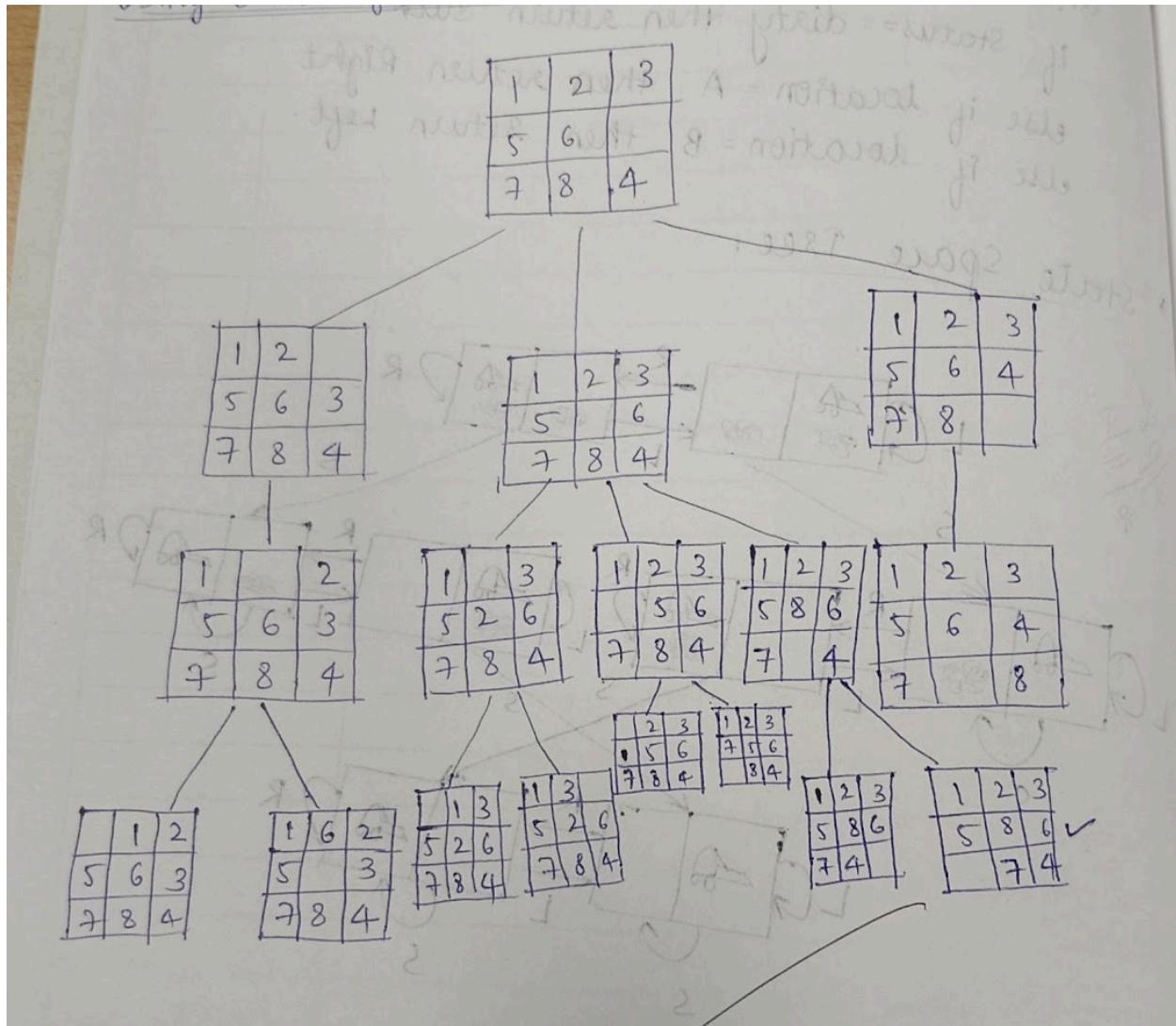
1	2	3
5	0	6
7	8	4

1	0	2
5	6	3
7	8	4

1	2	3
5	6	4
7	0	8

1	0	3
5	2	6
7	8	4

State Space Diagram:



PROGRAM-4

Title:Iterative Deepening Search

Date:18-12-2023

Algorithm:

Algorithm : search algorithm 18/12/23

// Returns true if target is reachable from src within max_depth

bool IDDFS (src, target, max_depth)

 for limit from 0 to max_depth

 if DLS (src, target, limit) = true

 return true

 return false

bool DLS (src, target, limit)

 if (src == target)

 return true;

 // if reached maximum depth stop searching

 if (limit <= 0)

 return false;

 foreach adjacent i of src

 if DLS (i, target, limit-1)

 return true

 return false

Code:

```
from collections import defaultdict  
class Graph:
```

```
def __init__(self, vertices):  
  
    # No. of vertices  
    self.V = vertices  
  
    # default dictionary to store graph  
    self.graph = defaultdict(list)  
    self.ans = list()  
  
    # function to add an edge to graph  
    def addEdge(self, u, v):  
        self.graph[u].append(v)
```

```

# A function to perform a Depth-Limited search
# from given source 'src'
def DLS(self,src,target,maxDepth,l):

    if src == target :
        # print(self.ans)
        return True

    # If reached the maximum depth, stop recursing.
    if maxDepth <= 0 : return False

    # Recur for all the vertices adjacent to this vertex
    for i in self.graph[src]:
        if(self.DLS(i,target,maxDepth-1,l)):
            l.append(i)
            return True
    return False

# IDDFS to search if target is reachable from v.
# It uses recursive DLS()
def IDDFS(self,src, target, maxDepth):

    # Repeatedly depth-limit search till the
    # maximum depth
    for i in range(maxDepth):
        l = []
        if (self.DLS(src, target, i,l)):
            l.append(src)
            l.reverse()
            return l
    return l

# Create a graph given in the above diagram
n,e = map(int ,input("Enter no.of vertices and edges").split())
g = Graph (n);
for i in range(e):
    a,b = map(int , input().split())
    g.addEdge(a,b)

```

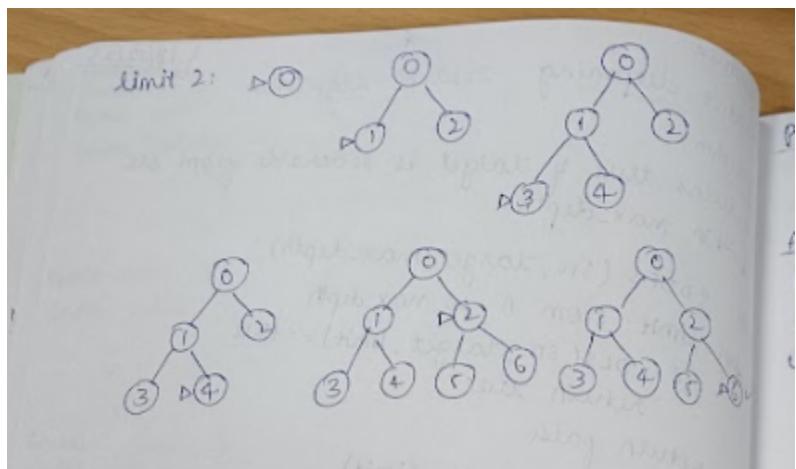
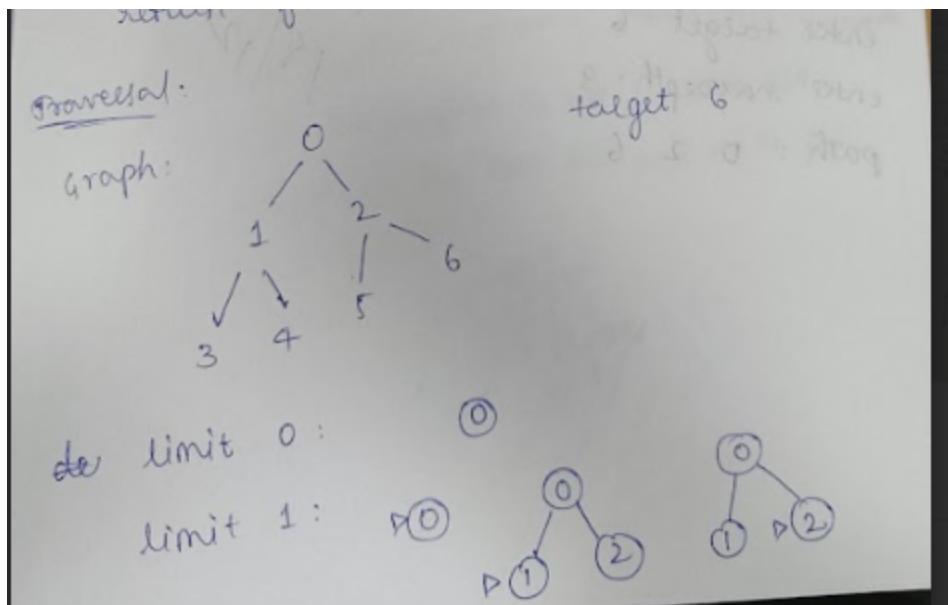
```
# g.addEdge(0, 1)
# g.addEdge(0, 2)
# g.addEdge(1, 3)
# g.addEdge(1, 4)
# g.addEdge(2, 5)
# g.addEdge(2, 6)
```

```
target = int(input("Enter the target vertex"))
maxDepth = int(input("Enter the max depth"))
src = 0
l = g.IDDFS(src, target, maxDepth)
if len(l)!=0:
    print(l)
    print ("Target is reachable from source " +
          "within max depth")
else :
    print ("Target is NOT reachable from source " +
          "within max depth")
```

OUTPUT:

```
Enter no.of vertices and edges7 6
0 1
0 2
1 3
1 4
2 5
2 6
Enter the target vertex6
Enter the max depth3
[0, 2, 6]
Target is reachable from source within max depth
```

STATE SPACE DIAGRAM:



PROGRAM-5

Title: 8-Puzzle using A* algorithm

Date: 8-01-2024

Algorithm:

Algorithm:
make an open list containing only the starting node
make an empty closed list
while (the destination node has not been reached):
 consider the node with the lowest f score in the
 open list.
 if (this node is our destination node):
 finished
 if not:
 put the current node in the closed list and
 look at all of its neighbors
 for each neighbor of the current node:
 if (neighbor has lower g value than current and
 is in the closed list):
 replace the neighbor with the new, lower, g value
 current node is now the neighbor's parent
 else if (current g value is lower and this neighbor
 is in the open list):
 replace the neighbor with the new, lower, g value
 change the neighbor's parent to our current
 node
 else if this neighbor is not in both lists:
 add it to the open list and set its g
 value to current g + 1
 add it to the open list and set its g
 value to current g + 1

Ball
8/1/24

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("")  
print("Heuristic value:", self.h(cur.data, goal))  
print("Path cost:", cur.level)  
  
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 open list based on f value  
self.open.sort(key=lambda x: x.fval, reverse=False)  
puz = Puzzle(3)  
puz.process()
```

OUTPUT:

```
Enter the start state matrix
```

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

```
Enter the goal state matrix
```

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

```
|  
|  
\'/'
```

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

```
Heuristic value: 3
```

```
Path cost: 0
```

```
|  
|  
\'/'
```

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

```
Heuristic value: 2
```

```
Path cost: 1
```

```
|  
|  
\'/'
```

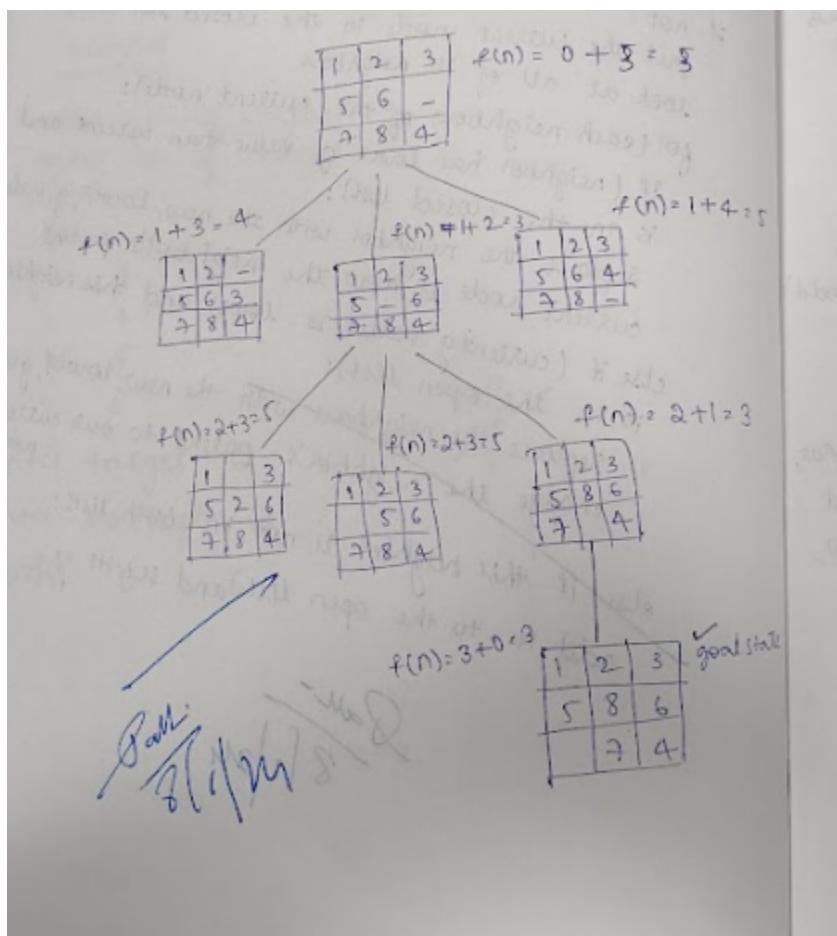
```
1 2 3  
5 8 6  
7 _ 4
```

```
Heuristic value: 1
```

```
Path cost: 2
```

Path cost: 2
 / \
 1 2 3
 5 8 6
 - 7 4
 Heuristic value: 0
 Path cost: 3

STATE SPACE DIAGRAM:



PROGRAM-6

Title: Entailment

Date:22-01-2024

Algorithm:

Algorithm :

function TT-Entail? (KB, q) returns true or false
inputs : KB, the knowledge base
q, the query, a sentence
symbols: a list of the propositional symbols in KB
and q

function TT-CHECK-ALL (KB, q, symbols, model)
returns true or false.
if Empty? (symbols) then
 if PL-true? (KB, model) then return PL-true?
 else return false.
else do
 p = First (symbols); rest = Rest (symbols)
 return TT-CHECK-ALL (KB, q, rest, EXTEND (p, model)) and
 TT-CHECK-ALL (KB, q, rest, EXTEND (p,
 false, model))

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,'~":"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
    else:
        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 rule: (p^q)v(~p^r)
Enter the Query: pvq
*****Truth Table Reference*****
kb alpha
*****
True True
-----
True True
-----
False True
-----
False True
-----
True True
-----
False True
-----
True False
-----
The Knowledge Base does not entail query
> |

```

Truth Table:

p	q	r	$\sim p$	$(p \wedge q)$	$(\sim p \wedge r)$	$((p \wedge q) \vee (\sim p \wedge r))$	$(p \vee q)$
T	T	T	F	T	F	T	T
T	T	F	F	F	F	T	T
T	F	T	F	F	F	F	T
T	F	F	F	F	F	F	T
T	F	F	F	F	F	F	T
F	T	T	T	F	T	T	T
F	T	F	T	F	F	F	T
F	F	T	T	(x, a, F) nothing	T	T	F
F	F	F	T	F	F	F	T

PROGRAM-7

Title: Resolution

Date:22-01-2024

Algorithm:

```
Algorithm:  
function PL-RESOLUTION (KB,  $\alpha$ ) returns true or false  
input: KB, the knowledge base, a sentence in propositional logic  
-positional logic  $\alpha$ , the query, a sentence in propositional logic  
clauses  $\leftarrow$  a set of clauses in the CNF representation of  $KB \wedge \neg \alpha$ .  
new  $\leftarrow \{\}$   
loop do  
for each pair of clauses  $c_i, c_j$  in clauses do  
resolvents  $\leftarrow$  PL-RESOLVE ( $c_i, c_j$ )  
if resolvents contains the empty clause then  
- we  
new  $\leftarrow$  new  $\cup$  resolvents  
if new  $\subseteq$  clauses then return false  
clauses  $\leftarrow$  clauses  $\cup$  new
```

Code

```
kb = []
```

```
def CLEAR():
```

```
    global kb
```

```
    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)

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]
    clauses = ask_list + kb[:]

    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

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 happened
            # 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

```

```
def make_sentence(arg):
    if isLiteral(arg) or isNotList(arg):
        return [arg]
    if isOrList(arg):
        return clear_duplicate(arg[1:])
    return
```

```
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
```

```
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
```

```
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

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

def convertCNF(sentence):
    while not isCNF(sentence):
        if sentence is None:
            return None
        sentence = makeCNF(sentence)
    return sentence

```

```

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"

    if operand == 'implies' and len(sentence) == 3:
        return makeCNF(['or', ['not', makeCNF(sentence[1])], makeCNF(sentence[2])])

    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

```

```
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
```

```
CLEAR()
```

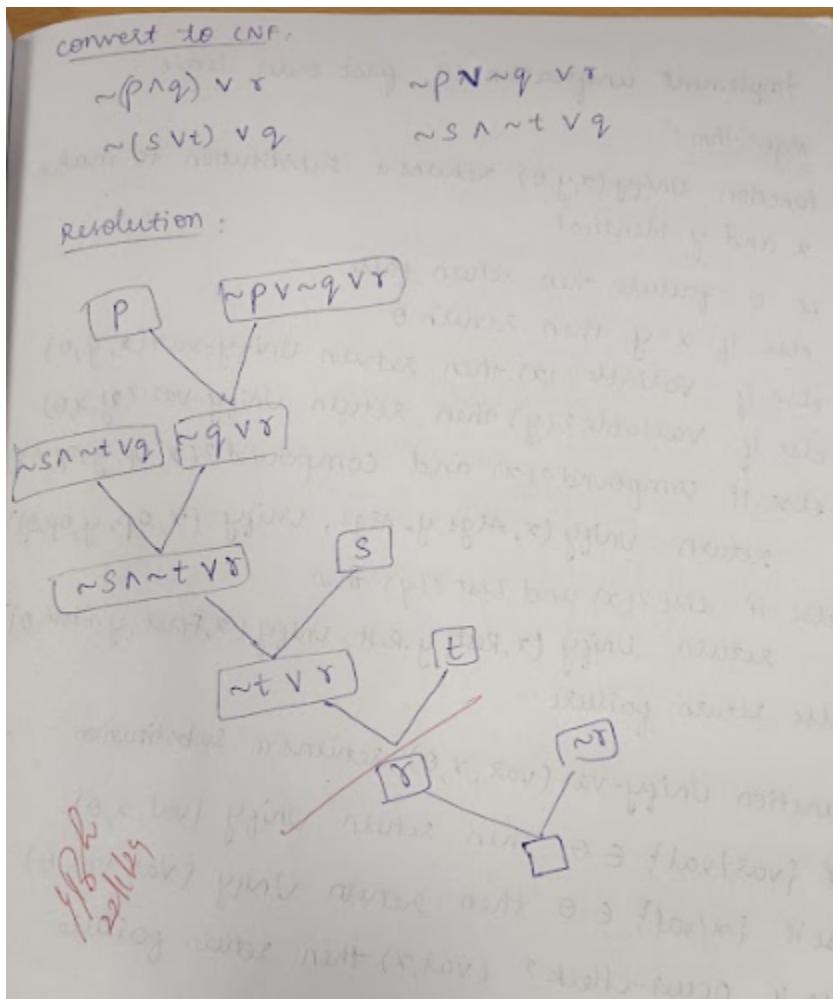
```
TELL('p')
TELL(['implies', ['and', 'p', 'q'], 'r'])
TELL(['implies', ['or', 's', 't'], 'q'])
TELL('t')
TELL('s')
print(ASK('r'))
```

OUTPUT:

True

>

Proof:



PROGRAM-8

Title: Unification in FOL

Date:29-01-2024

Algorithm:

Implement unification in first order logic

Algorithm:

function $\text{Unify}(x, y, \theta)$ returns a substitution to make x and y identical

if $\theta = \text{failure}$ then return false

else if $x = y$ then return θ

else if $\text{variable?}(x)$ then return $\text{Unify-var}(x, y, \theta)$

else if $\text{variable?}(y)$ then return $\text{Unify-var}(y, x, \theta)$

else if $\text{compound?}(x)$ and $\text{compound?}(y)$ and $\text{op}(x) = \text{op}(y)$

return $\text{Unify}(x.\text{Args}, y.\text{Args}, \text{Unify}(x.\text{Op}, y.\text{Op}, \theta))$

else if $\text{list?}(x)$ and $\text{list?}(y)$ then

return $\text{Unify}(x.\text{Rest}, y.\text{Rest}, \text{Unify}(x.\text{First}, y.\text{First}, \theta))$

else return failure

function $\text{Unify-var}(\text{var}, x, \theta)$ returns a substitution

if $\{\text{var}/\text{val}\} \in \theta$ then return $\text{Unify}(\text{val}, x, \theta)$

if $\{\text{val}/\text{var}\} \in \theta$ then return $\text{Unify}(\text{var}, \text{val}, \theta)$

else if $\{\text{x}/\text{val}\} \in \theta$ then return failure

else if $\text{occur-Check?}(\text{val}, x)$ then return failure

else return add $\{\text{var}/x\}$ to θ

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 []
else:
    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 []
else:
    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()

```

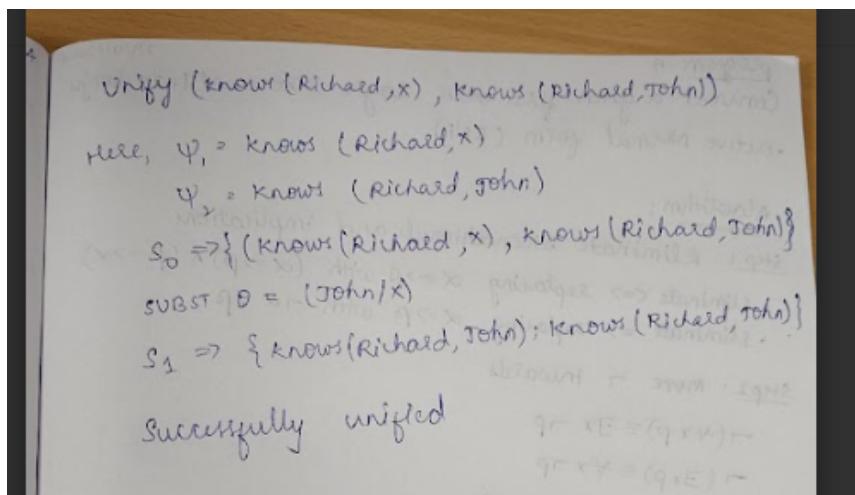
OUTPUT:

```

Enter the first expression
grants(x,y,z)
Enter the second expression
grants(b,c)
Length of attributes 3 and 2 do not match. Cannot be unified
The substitutions are:
[]
> |

```

Proof:

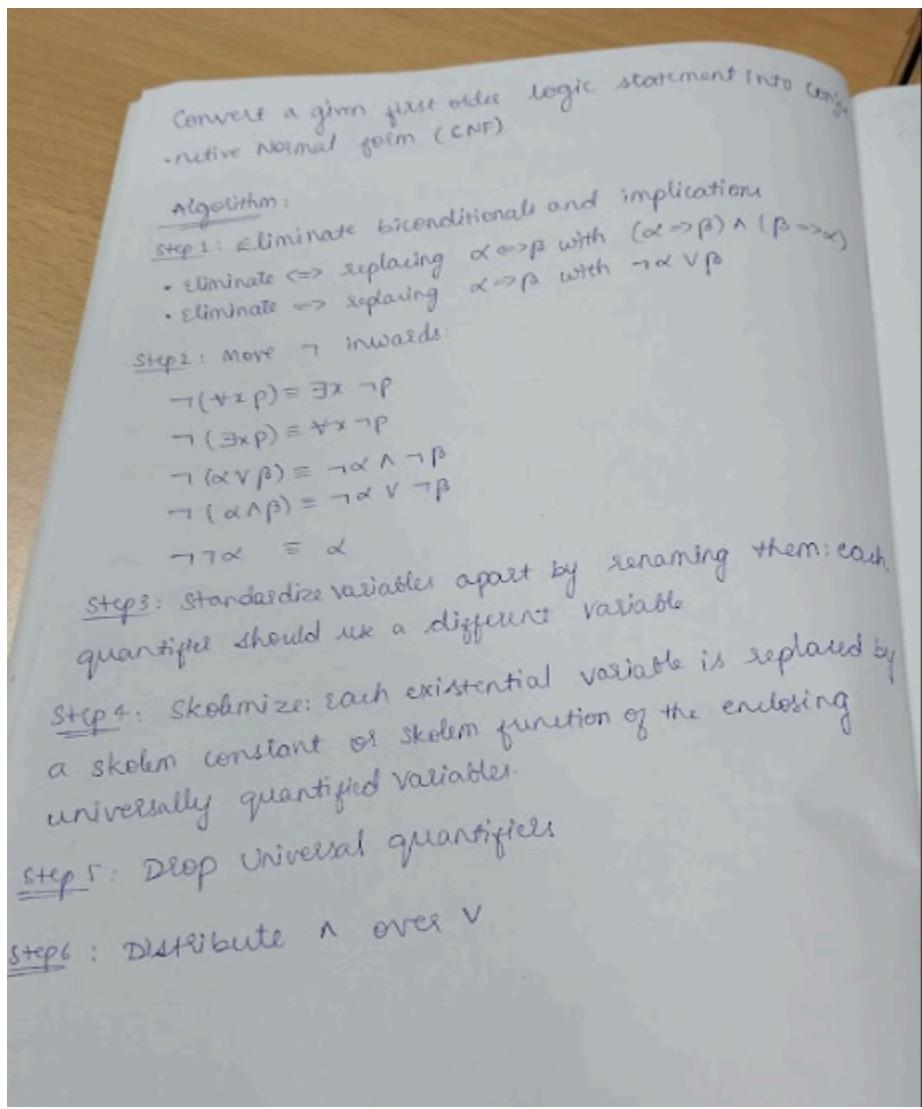


PROGRAM-9

Title: Convert FOL to CNF

Date:29-01-2024

Algorithm:



Code:

```
# Online Python compiler (interpreter) to run Python online.
```

```
# Write Python 3 code in this online editor and run it.
```

```
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-zA-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('[ \\forall \\exists ].', 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:

```

Enter FOL:
vx man(x)=>respects(woman,x)
The CNF form of the given FOL is:
~ man(A)Vrespects(woman,A)
>

```

Proof:

(proof)

FOL: $A \Rightarrow (B \Leftrightarrow C)$

Step 1: eliminate implication
 $\neg A \vee (B \Leftrightarrow C)$

Step 2: eliminate bi-directional implication
 $\neg A \vee (B \Rightarrow C) \wedge (C \Rightarrow B)$

$\neg A \vee (\neg B \vee C) \wedge (\neg C \vee B)$

Step 3: Apply distributive law
 $(\neg A \vee \neg B \vee C) \wedge (\neg A \vee \neg C \vee B)$

$(\neg A \vee \neg B \vee C) \wedge (\neg A \vee \neg C \vee B)$

$(\neg A \vee \neg B \vee C) \wedge (\neg A \vee \neg C \vee B)$

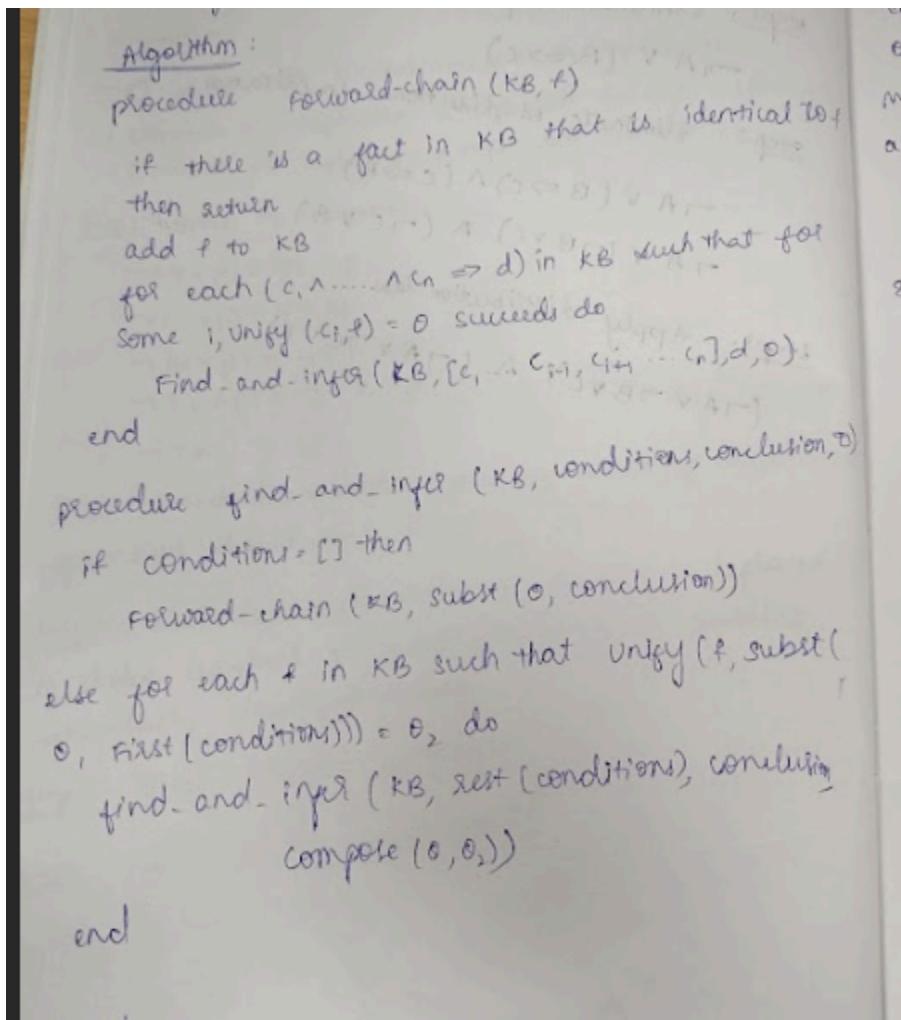
$(\neg A \vee \neg B \vee C) \wedge (\neg A \vee \neg C \vee B)$

PROGRAM-10

Title: Forward Chaining

Date: 29-01-2024

Algorithm:



The image shows a handwritten algorithm for Forward Chaining. It starts with a header "Algorithm:" followed by two procedures: "forward-chain(KB, f)" and "find-and-infer(KB, conditions, conclusion)". The "forward-chain" procedure checks if there's a fact in KB identical to f; if so, it returns. Otherwise, it adds f to KB and iterates over all rules (c₁ ∧ ... ∧ c_n ⇒ d) in KB. For each rule, it checks if unify(c_i, f) = θ succeeds for some i. If so, it calls "find-and-infer(KB, [c₁, ..., c_n, c_{i+1}, ..., c_n], d, θ)". The "find-and-infer" procedure handles two cases: if conditions is empty, it calls "forward-chain(KB, subst(θ, conclusion))". Otherwise, it iterates over KB for each fact f such that unify(f, subst(θ, first(conditions))) = θ₂. It then calls "find-and-infer(KB, rest(conditions), conclusion, compose(θ, θ₂))". Both procedures end with an "end" keyword.

Code:

```
import re

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

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

```
def getPredicates(string):
    expr = '([a-zA-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. criminal(west)
3. missile(m1)
4. owns(china,m1)
5. american(west)
6. sells(west,m1,china)
7. hostile(china)
8. enemy(china,america)
> |
```