**B.M.S. COLLEGE OF ENGINEERING**

**BENGALURU** Autonomous Institute, Affiliated to VTU



Lab Record

**Artificial Intelligence (22CS5PCAIN)**

*Submitted in partial fulfillment for the 65h Semester Laboratory*

Bachelor of Technology in

Computer Science and Engineering

*Submitted by:*

**Tanisha Gotadke**

1BM21CS229

Department of Computer Science and Engineering B.M.S. College of Engineering Bull Temple Road, Basavanagudi, Bangalore 560019 Nov 23-Feb 2024

**B.M.S. COLLEGE OF ENGINEERING**

**DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING**



***CERTIFICATE***

This is to certify that the Artificial Intelligence (22CS5PCAIN) laboratory has beencarried out by Tanisha Gotadke (1BM21CS229) during the 5th Semester November - February 2024.

Signature of the Faculty Incharge: Sandhya A Kulkarni

Assistant Professor

Department of Computer Science and Engineering

B.M.S. College of Engineering, Bangalore

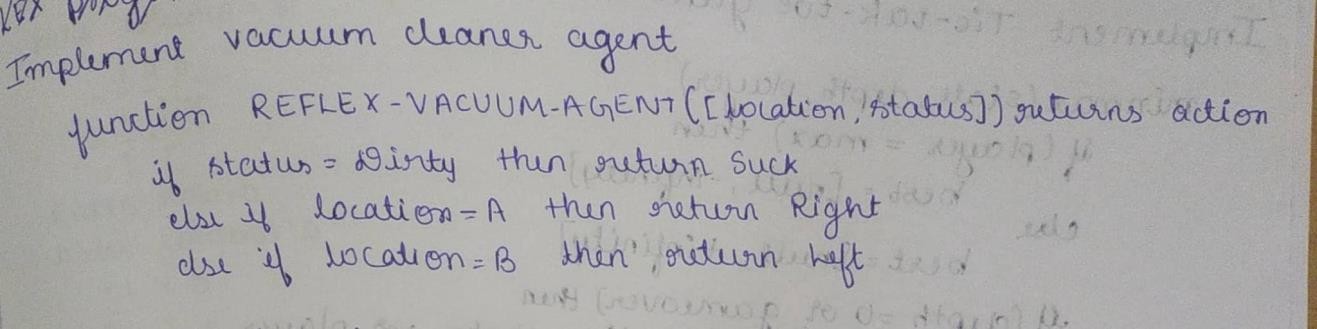
**Table of Contents**

|  |  |  |
| --- | --- | --- |
| **Sl. No.** | **Title** | **Page No.** |
| 1. | Tic Tac Toe |  |
| 2. | 8 Puzzle Breadth First Search Algorithm |  |
| 3. | 8 Puzzle Iterative Deepening Search Algorithm |  |
| 4. | 8 Puzzle A\* Search Algorithm |  |
| 5. | Vacuum Cleaner |  |
| 6. | Knowledge Base Entailment |  |
| 7. | Knowledge Base Resolution |  |
| 8. | Unification |  |
| 9. | FOL to CNF |  |
| 10. | Forward reasoning |  |

# Program- 1

Implement Vacuum cleaner problem for 2 rooms ,any type of agent can be considered simple reflex or model based etc.

Algorithm:



Code:

def vacuum\_world():

# initializing goal\_state

# 0 indicates Clean and 1 indicates Dirty goal\_state = {'A': '0', 'B': '0'}

cost = 0

location\_input = input("Enter Location of Vacuum") #user\_input of location vacuum is placed

status\_input = input("Enter status of " + location\_input) #user\_input if location is dirty or clean

status\_input\_complement = input("Enter status of other room") print("Initial Location Condition" + str(goal\_state))

if location\_input == 'A':

# Location A is Dirty.

print("Vacuum is placed in Location A") if status\_input == '1':

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

goal\_state['A'] = '0' cost += 1 #cost for suck

print("Cost for CLEANING A " + str(cost)) print("Location A has been Cleaned.")

if status\_input\_complement == '1': # if B is Dirty

print("Location B is Dirty.") print("Moving right to the Location B. ") cost += 1 #cost for moving right

print("COST for moving RIGHT" + str(cost)) # suck the dirt and mark it as clean goal\_state['B'] = '0'

cost += 1 #cost for suck print("COST for SUCK " + str(cost))

print("Location B has been Cleaned. ") else:

print("No action" + str(cost)) # suck and mark clean

print("Location B is already clean.")

if status\_input == '0':

print("Location A is already clean ")

if status\_input\_complement == '1':# if B is Dirty print("Location B is Dirty.")

print("Moving RIGHT to the Location B. ") cost += 1 #cost for moving right print("COST for moving RIGHT " + str(cost))

# suck the dirt and mark it as clean goal\_state['B'] = '0'

cost += 1 #cost for suck print("Cost for SUCK" + str(cost))

print("Location B has been Cleaned. ") else:

print("No action " + str(cost)) print(cost)

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

else:

print("Vacuum is placed in location B") # Location B is Dirty.

if status\_input == '1': print("Location B is Dirty.")

# suck the dirt and mark it as clean goal\_state['B'] = '0'

cost += 1 # cost for suck

print("COST for CLEANING " + str(cost)) print("Location B has been Cleaned.") if status\_input\_complement == '1':

# if A is Dirty

print("Location A is Dirty.") print("Moving LEFT to the Location A. ") cost += 1 # cost for moving right print("COST for moving LEFT" + str(cost)) # suck the dirt and mark it as clean goal\_state['A'] = '0'

cost += 1 # cost for suck print("COST for SUCK " + str(cost)) print("Location A has been Cleaned.")

else:

print(cost)

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

if status\_input\_complement == '1': # if A is Dirty print("Location A is Dirty.")

print("Moving LEFT to the Location A. ") cost += 1 # cost for moving right print("COST for moving LEFT " + str(cost)) # suck the dirt and mark it as clean goal\_state['A'] = '0'

cost += 1 # cost for suck print("Cost for SUCK " + str(cost))

print("Location A has been Cleaned. ") else:

print("No action " + str(cost)) # suck and mark clean

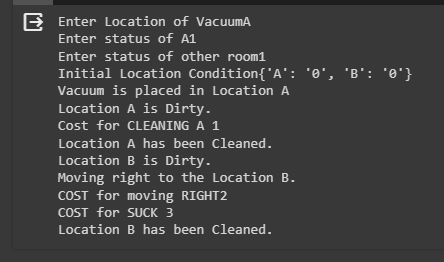
print("Location A is already clean.") # done cleaning

print("GOAL STATE: ")

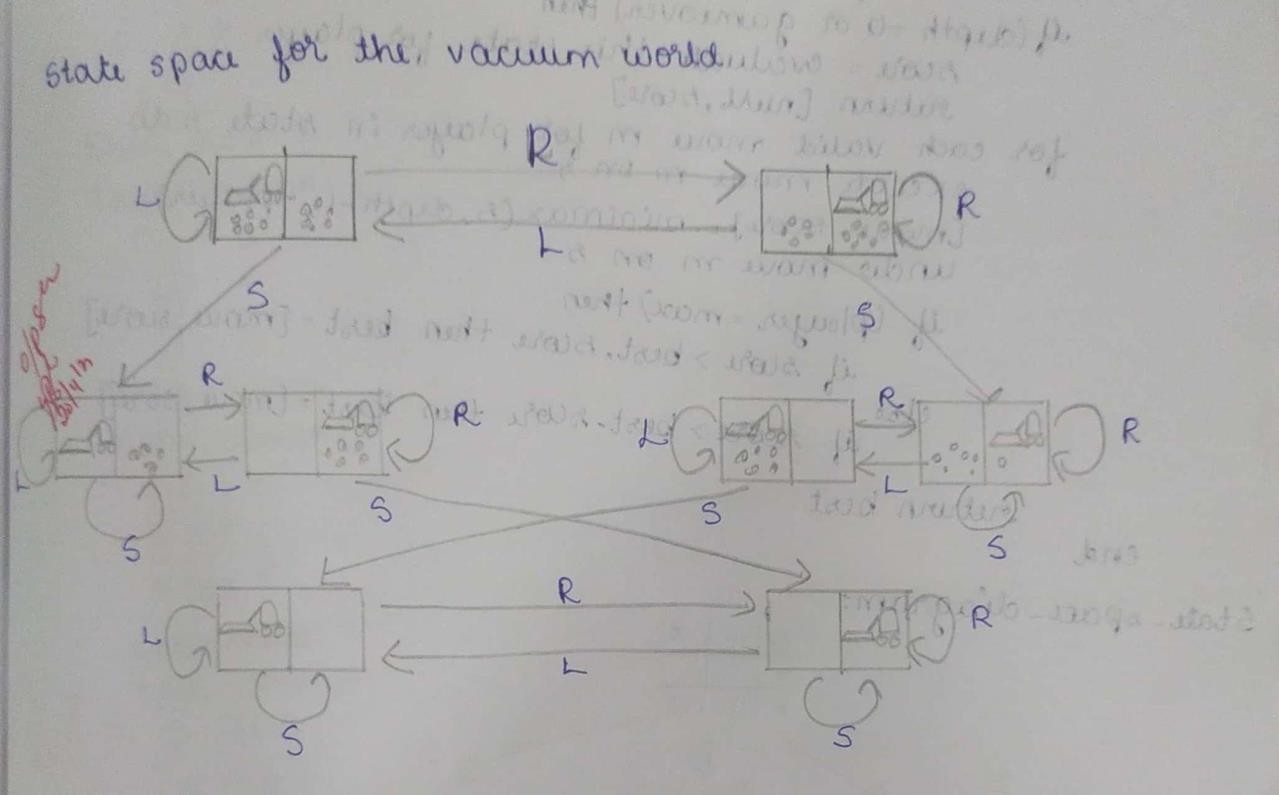
print(goal\_state)

print("Performance Measurement: " + str(cost)) vacuum\_world()

Output:



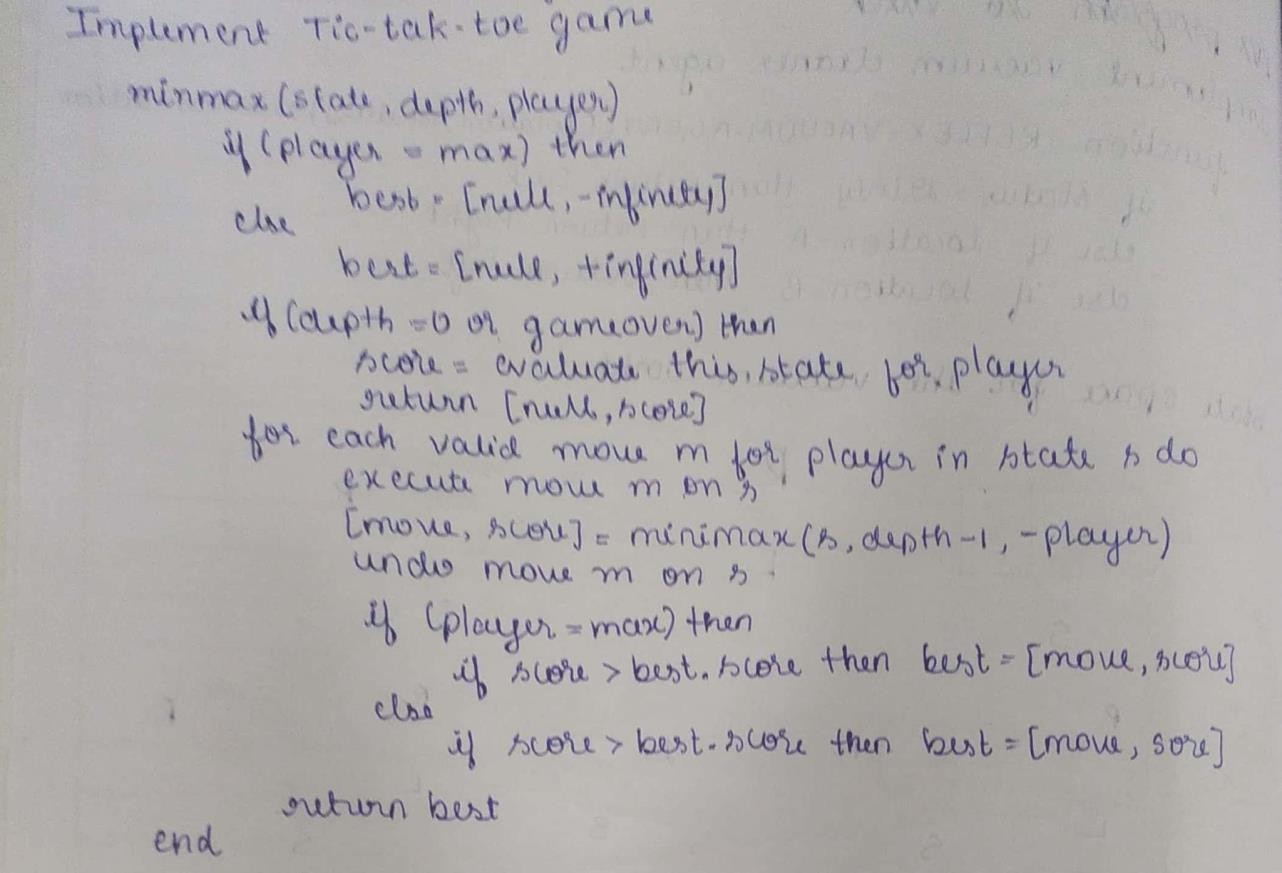
State-Space Diagram:



# Program- 2

Explore the working of Tic Tac Toe using Min max strategy

Algorithm:



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

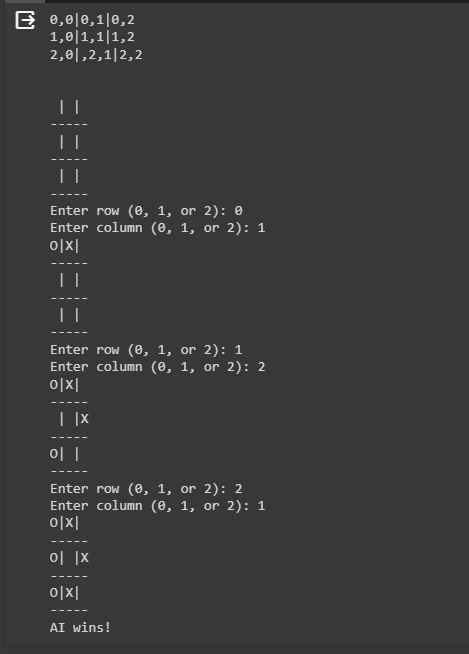
ai\_row, ai\_col = ai\_move() board[ai\_row][ai\_col] = "O" if check\_winner("O"):

print\_board() print("AI wins!") break

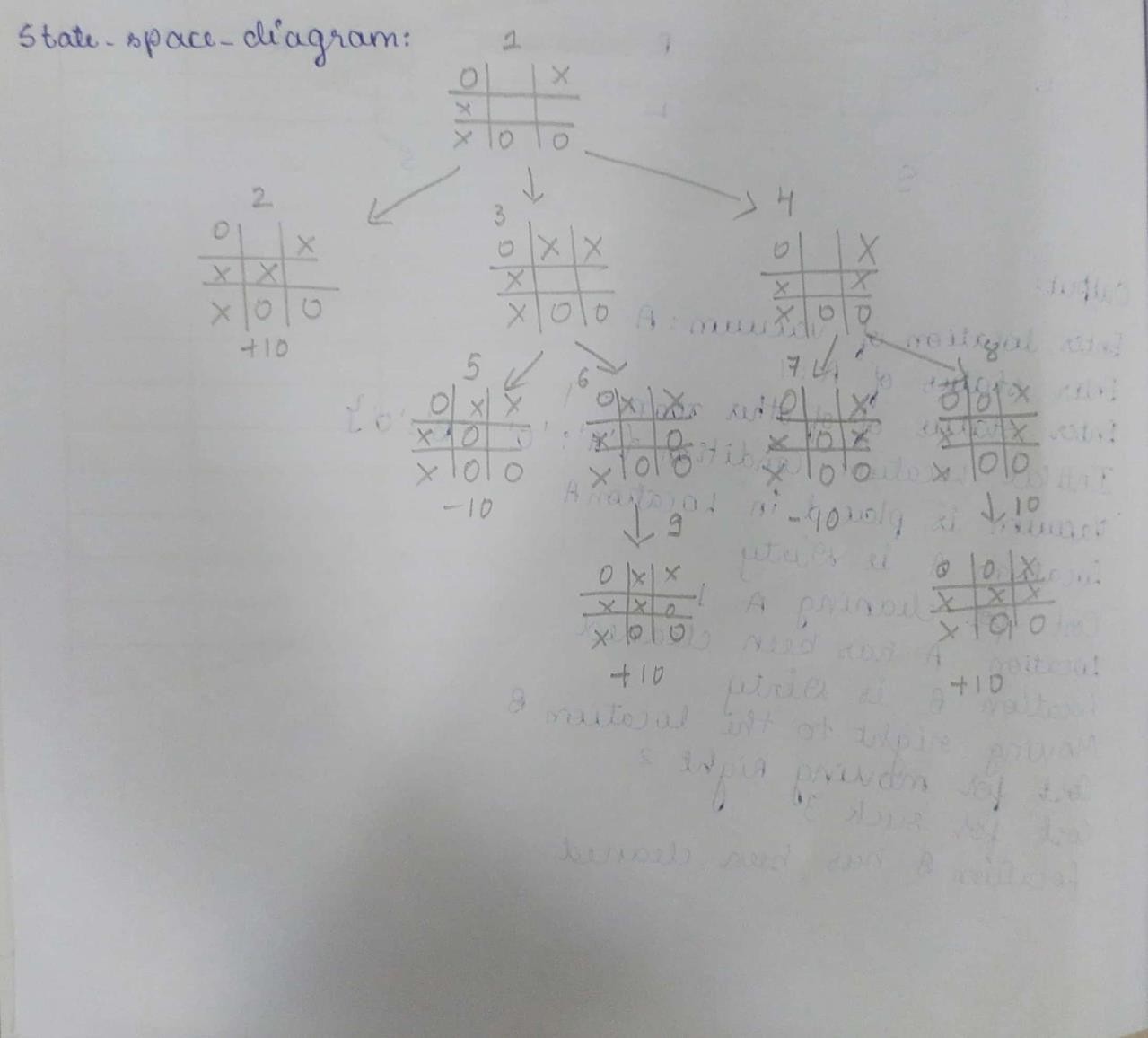
else:

print("Cell is already occupied. Try again.")

Output:



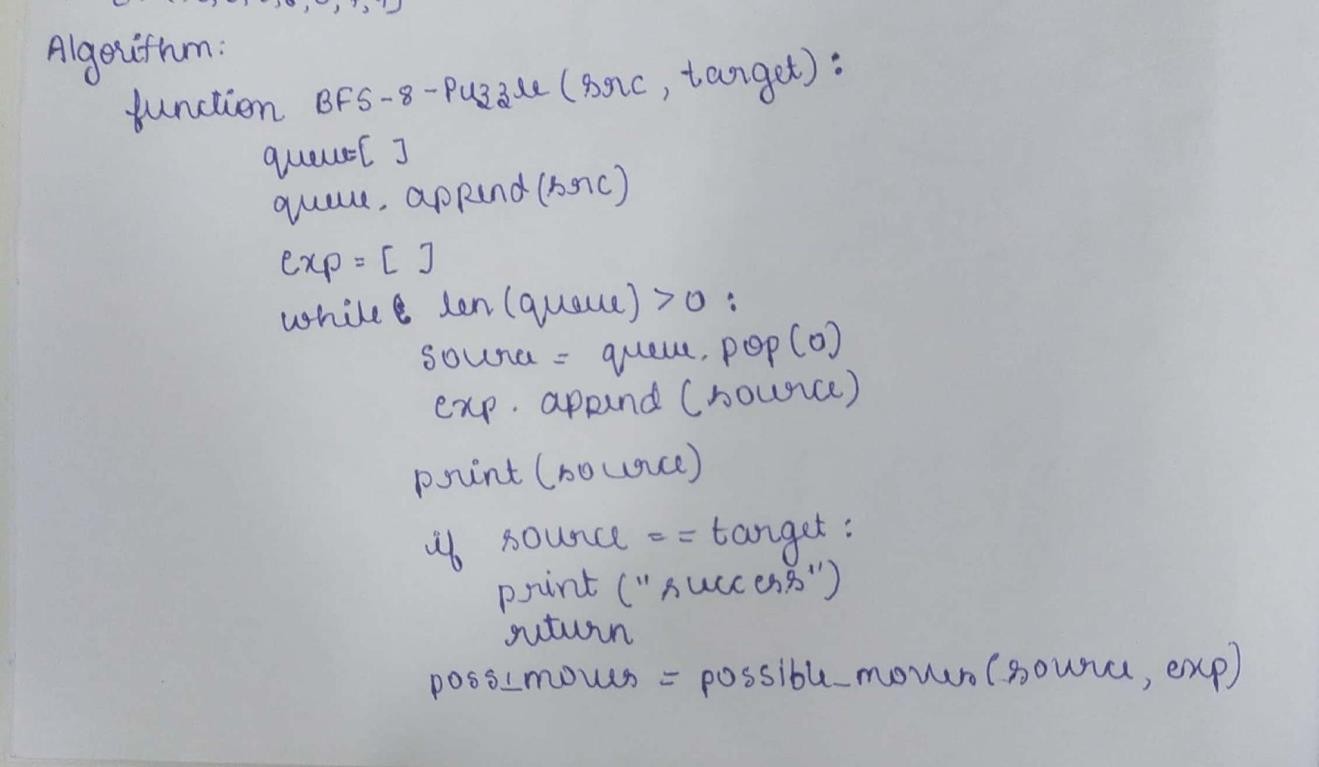
State-Space Diagram:



# Program- 3

Implement the 8 Puzzle Breadth First Search Algorithm.

Algorithm:



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) cost+=1 exp.append(source)

print(source[0],'|',source[1],'|',source[2])

print(source[3],'|',source[4],'|', source[5])

print(source[6],'|', source[7],'|',source[8]) print()

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

return

poss\_moves\_to\_do = possible\_moves(source, exp) for move in poss\_moves\_to\_do:

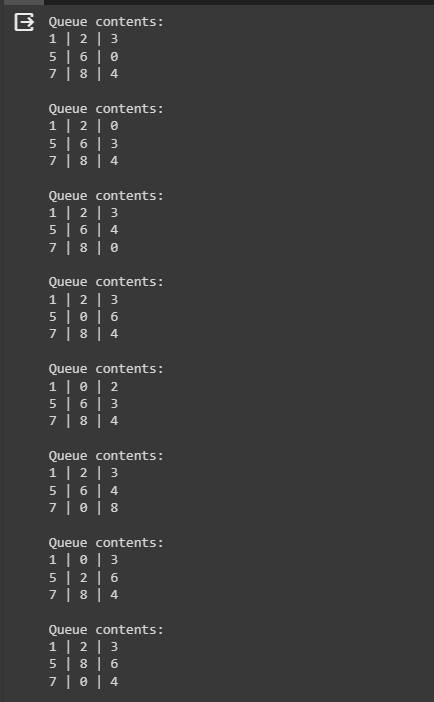
if move not in exp and move not in queue: queue.append(move)

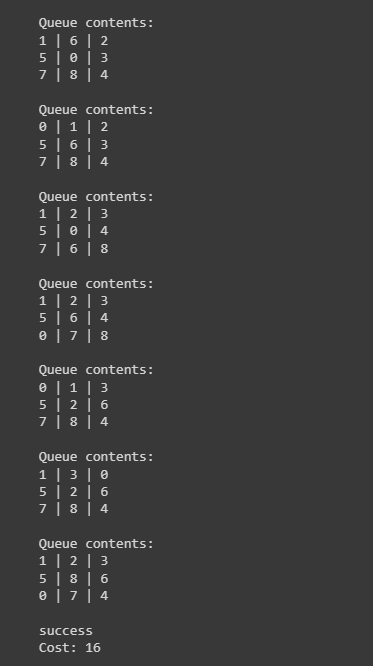
src = [1, 2, 3, 5, 6, 0, 7, 8, 4]

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

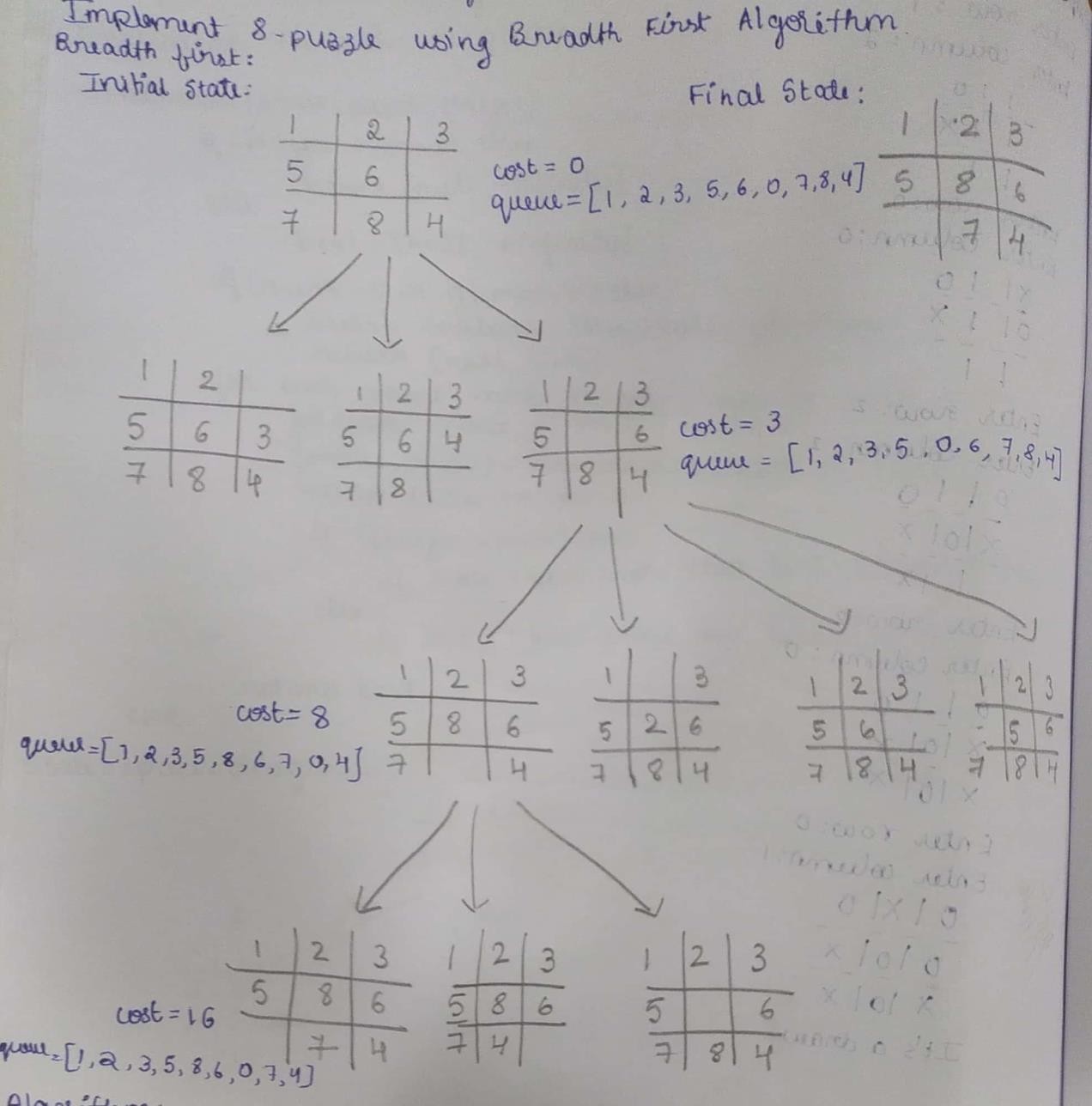
bfs(src, target)

Output:

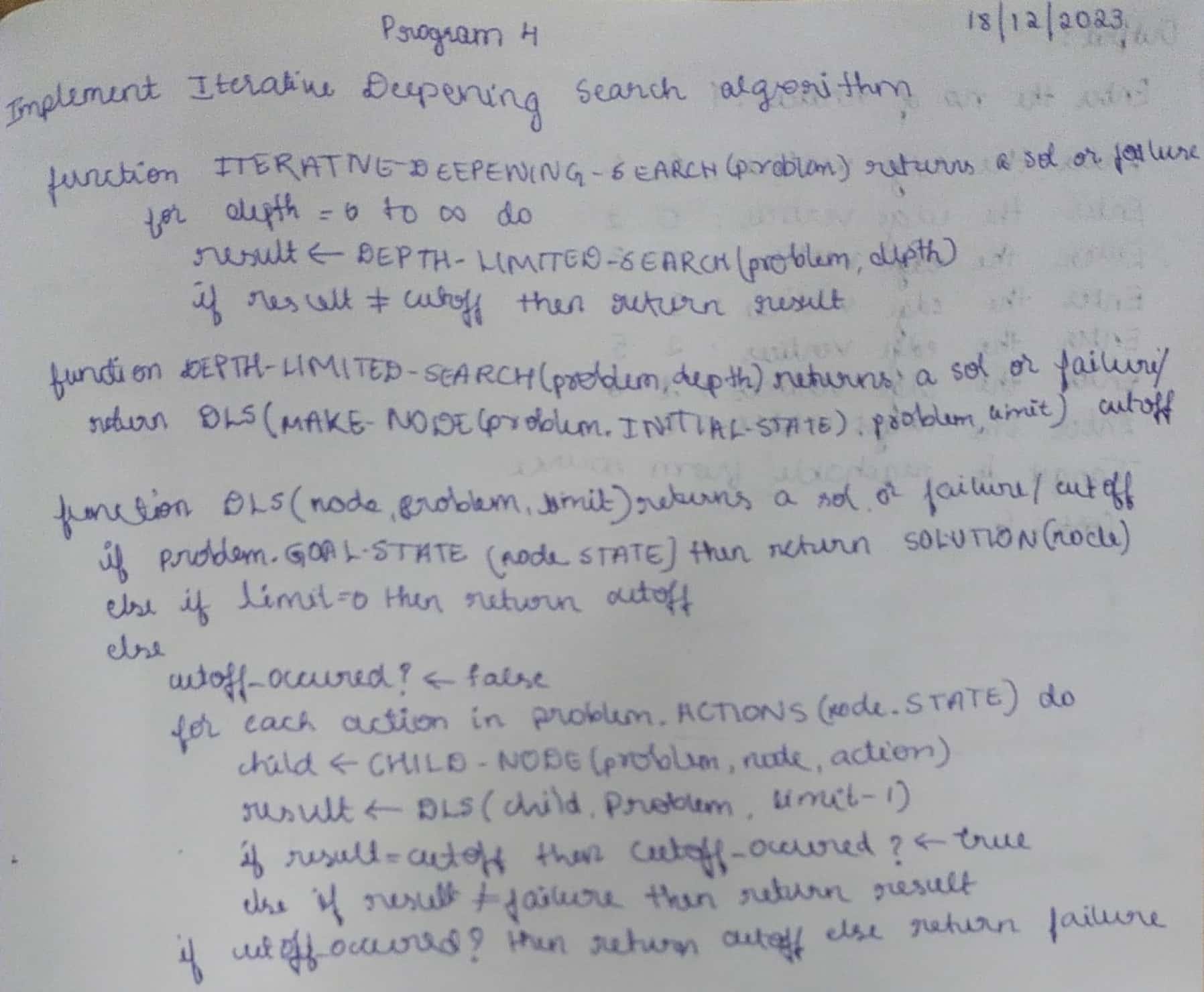




State-Space Diagram:



# Program- 4

Implement Iterative deepening search algorithm. Algorithm:

Code:

from collections import defaultdict cost=0

class Graph:

def init (self,vertices): self.V = vertices

self.graph = defaultdict(list) def addEdge(self,u,v):

self.graph[u].append(v)

def DLS(self,src,target,maxDepth):

if src == target : return True

if maxDepth <= 0 : return False for i in self.graph[src]:

if(self.DLS(i,target,maxDepth-1)): return True

return False

def IDDFS(self,src, target, maxDepth): for i in range(maxDepth):

if (self.DLS(src, target, i)): return True

return False src = 0

pin=int(input('Enter the number of verices:')) g=Graph(pin)

while(pin>1):

e1=int(input('Enter the first vertex:')) e2=int(input('Enter the second vertex:')) g.addEdge(e1,e2)

pin-=1

target=int(input('Enter the target vertex:')) maxDepth=int(input('Enter the max depth:')) pen=1

while(pen<=maxDepth):

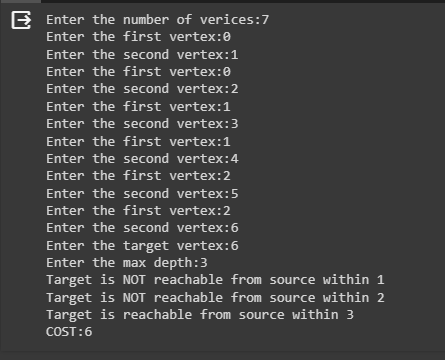
if g.IDDFS(src, target, pen) == True:

print ("Target is reachable from source within",pen) print("COST:6")

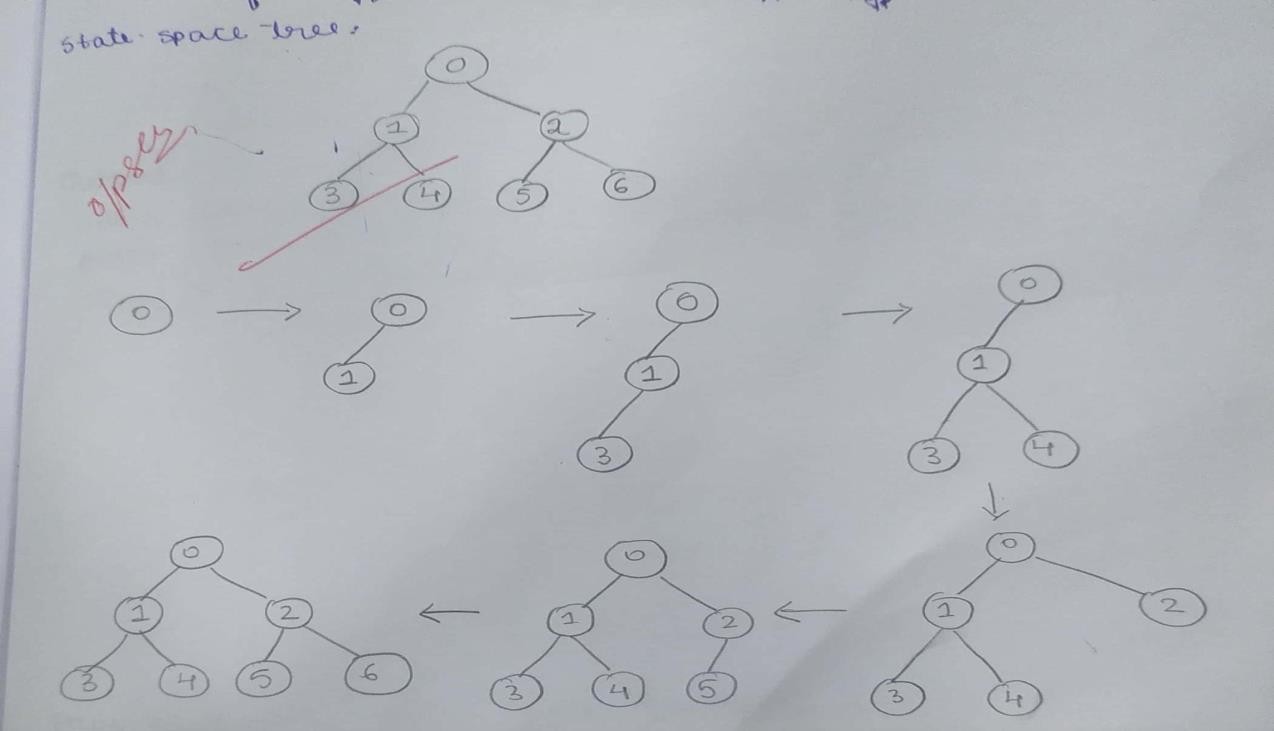
else :

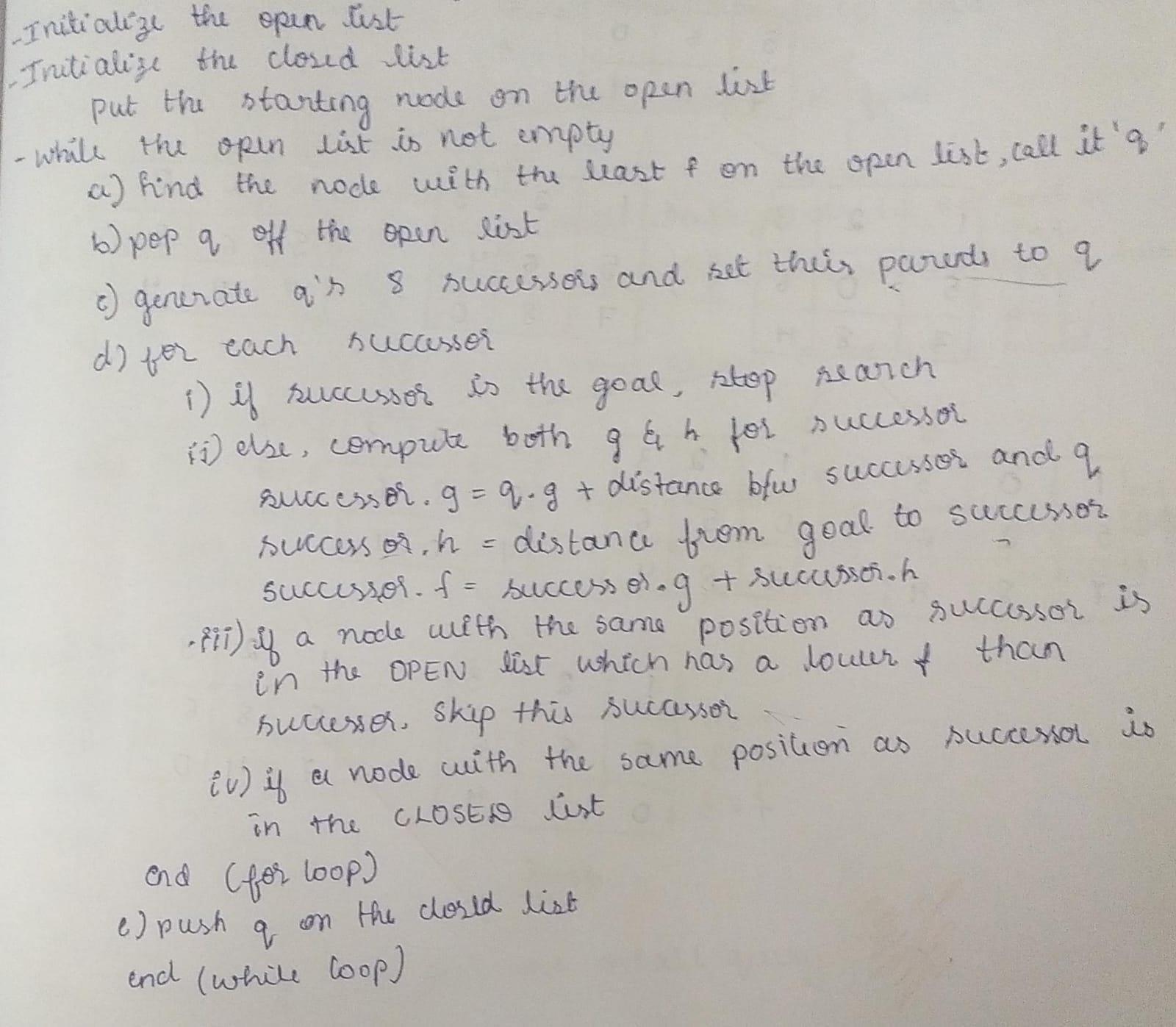
print ("Target is NOT reachable from source within",pen) pen+=1

Output:



State-Space Diagram:



Implement A\* for 8 puzzle problem Algorithm:

Code:

# Program- 5

from copy import deepcopy import numpy as np

import time

def bestsolution(state):

bestsol = np.array([], int).reshape(-1, 9) count = len(state) - 1

while count != -1:

bestsol = np.insert(bestsol, 0, state[count]['puzzle'], 0) count = (state[count]['parent'])

return bestsol.reshape(-1, 3, 3)

def all(checkarray): set=[]

for it in set:

for checkarray in it: return 1

else:

return 0

def manhattan(puzzle, goal):

a = abs(puzzle // 3 - goal // 3)

b = abs(puzzle % 3 - goal % 3) mhcost = a + b

return sum(mhcost[1:])

# will calcuates the number of misplaced tiles in the current state as compared to the goal state

def misplaced\_tiles(puzzle,goal): mscost = np.sum(puzzle != goal) - 1 return mscost if mscost > 0 else 0

#3[on\_true] if [expression] else [on\_false]

# will indentify the coordinates of each of goal or initial state values def coordinates(puzzle):

pos = np.array(range(9))

for p, q in enumerate(puzzle): pos[q] = p

return pos

# start of 8 puzzle evaluvation, using Manhattan heuristics def evaluvate(puzzle, goal):

steps = np.array([('up', [0, 1, 2], -3),('down', [6, 7, 8],

3),('left', [0, 3, 6], -1),('right', [2, 5, 8], 1)],

dtype = [('move', str, 1),('position', list),('head',

int)])

dtstate = [('puzzle', list),('parent', int),('gn', int),('hn', int)]

# initializing the parent, gn and hn, where hn is manhattan distance function call

costg = coordinates(goal) parent = -1

gn = 0

hn = manhattan(coordinates(puzzle), costg)

state = np.array([(puzzle, parent, gn, hn)], dtstate)

# We make use of priority queues with position as keys and fn as value. dtpriority = [('position', int),('fn', int)]

priority = np.array( [(0, hn)], dtpriority)

while 1:

priority = np.sort(priority, kind='mergesort', order=['fn', 'position'])

position, fn = priority[0]

priority = np.delete(priority, 0, 0)

# sort priority queue using merge sort,the first element is picked for exploring remove from queue what we are exploring

puzzle, parent, gn, hn = state[position] puzzle = np.array(puzzle)

# Identify the blank square in input blank = int(np.where(puzzle == 0)[0]) gn = gn + 1

c = 1

start\_time = time.time()

for s in steps: c = c + 1

if blank not in s['position']:

# generate new state as copy of current openstates = deepcopy(puzzle)

openstates[blank], openstates[blank + s['head']] = openstates[blank + s['head']], openstates[blank]

# The all function is called, if the node has been previously explored or not

if ~(np.all(list(state['puzzle']) == openstates,

1)).any():

dtstate)

end\_time = time.time()

if (( end\_time - start\_time ) > 2):

print(" The 8 puzzle is unsolvable ! \n") exit

# calls the manhattan function to calcuate the cost hn = manhattan(coordinates(openstates), costg)

# generate and add new state in the list

q = np.array([(openstates, position, gn, hn)],

state = np.append(state, q, 0)

# f(n) is the sum of cost to reach node and the cost

to rech fromt he node to the goal state

fn = gn + hn

the goal state.

q = np.array([(len(state) - 1, fn)], dtpriority) priority = np.append(priority, q, 0)

# Checking if the node in openstates are matching

if np.array\_equal(openstates, goal): print(' The 8 puzzle is solvable ! \n') return state, len(priority)

return state, len(priority)

# start of 8 puzzle evaluvation, using Misplaced tiles heuristics def evaluvate\_misplaced(puzzle, goal):

steps = np.array([('up', [0, 1, 2], -3),('down', [6, 7, 8],

3),('left', [0, 3, 6], -1),('right', [2, 5, 8], 1)],

dtype = [('move', str, 1),('position', list),('head',

int)])

dtstate = [('puzzle', list),('parent', int),('gn', int),('hn', int)]

costg = coordinates(goal)

# initializing the parent, gn and hn, where hn is misplaced\_tiles function call

parent = -1

gn = 0

hn = misplaced\_tiles(coordinates(puzzle), costg) state = np.array([(puzzle, parent, gn, hn)], dtstate)

# We make use of priority queues with position as keys and fn as value. dtpriority = [('position', int),('fn', int)]

priority = np.array([(0, hn)], dtpriority)

while 1:

priority = np.sort(priority, kind='mergesort', order=['fn', 'position'])

position, fn = priority[0]

# sort priority queue using merge sort,the first element is picked for exploring.

priority = np.delete(priority, 0, 0) puzzle, parent, gn, hn = state[position] puzzle = np.array(puzzle)

# Identify the blank square in input blank = int(np.where(puzzle == 0)[0]) # Increase cost g(n) by 1

gn = gn + 1 c = 1

start\_time = time.time() for s in steps:

c = c + 1

if blank not in s['position']:

# generate new state as copy of current

openstates = deepcopy(puzzle)

openstates[blank], openstates[blank + s['head']] = openstates[blank + s['head']], openstates[blank]

# The check function is called, if the node has been previously explored or not.

if ~(np.all(list(state['puzzle']) == openstates,

1)).any():

cost

dtstate)

end\_time = time.time()

if (( end\_time - start\_time ) > 2):

print(" The 8 puzzle is unsolvable \n") break

# calls the Misplaced\_tiles function to calcuate the

hn = misplaced\_tiles(coordinates(openstates), costg) # generate and add new state in the list

q = np.array([(openstates, position, gn, hn)],

state = np.append(state, q, 0)

# f(n) is the sum of cost to reach node and the cost

to rech fromt he node to the goal state

fn = gn + hn

goal state.

q = np.array([(len(state) - 1, fn)], dtpriority) priority = np.append(priority, q, 0)

# Checking if the node in openstates are matching the

if np.array\_equal(openstates, goal): print(' The 8 puzzle is solvable \n') return state, len(priority)

return state, len(priority)

# Program start

# User input for initial state puzzle = []

print(" Input vals from 0-8 for start state ")

for i in range(0,9):

x = int(input("enter vals :")) puzzle.append(x)

# User input of goal state goal = []

print(" Input vals from 0-8 for goal state ") for i in range(0,9):

x = int(input("Enter vals :")) goal.append(x)

n = int(input("1. Manhattan distance \n2. Misplaced tiles")) if(n ==1 ):

state, visited = evaluvate(puzzle, goal) bestpath = bestsolution(state)

print(str(bestpath).replace('[', ' ').replace(']', '')) totalmoves = len(bestpath) - 1

print('Steps to reach goal:',totalmoves) visit = len(state) - visited

print('Total nodes visited: ',visit, "\n") print('Total generated:', len(state))

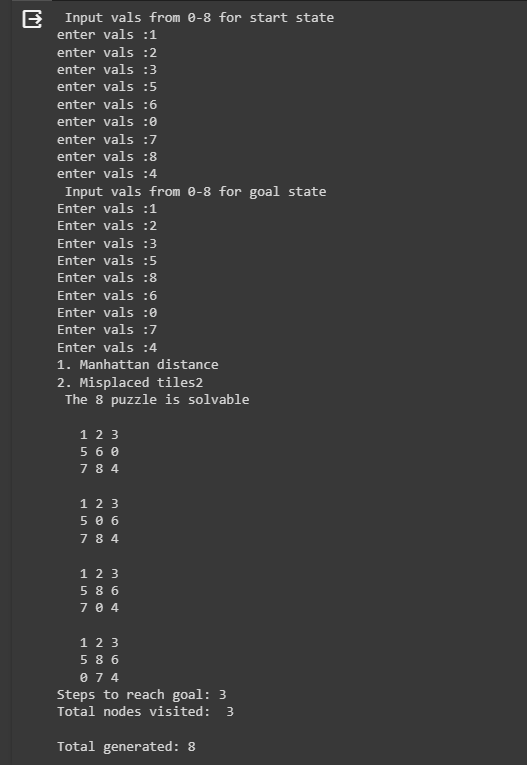
if(n == 2):

state, visited = evaluvate\_misplaced(puzzle, goal) bestpath = bestsolution(state) print(str(bestpath).replace('[', ' ').replace(']', '')) totalmoves = len(bestpath) - 1

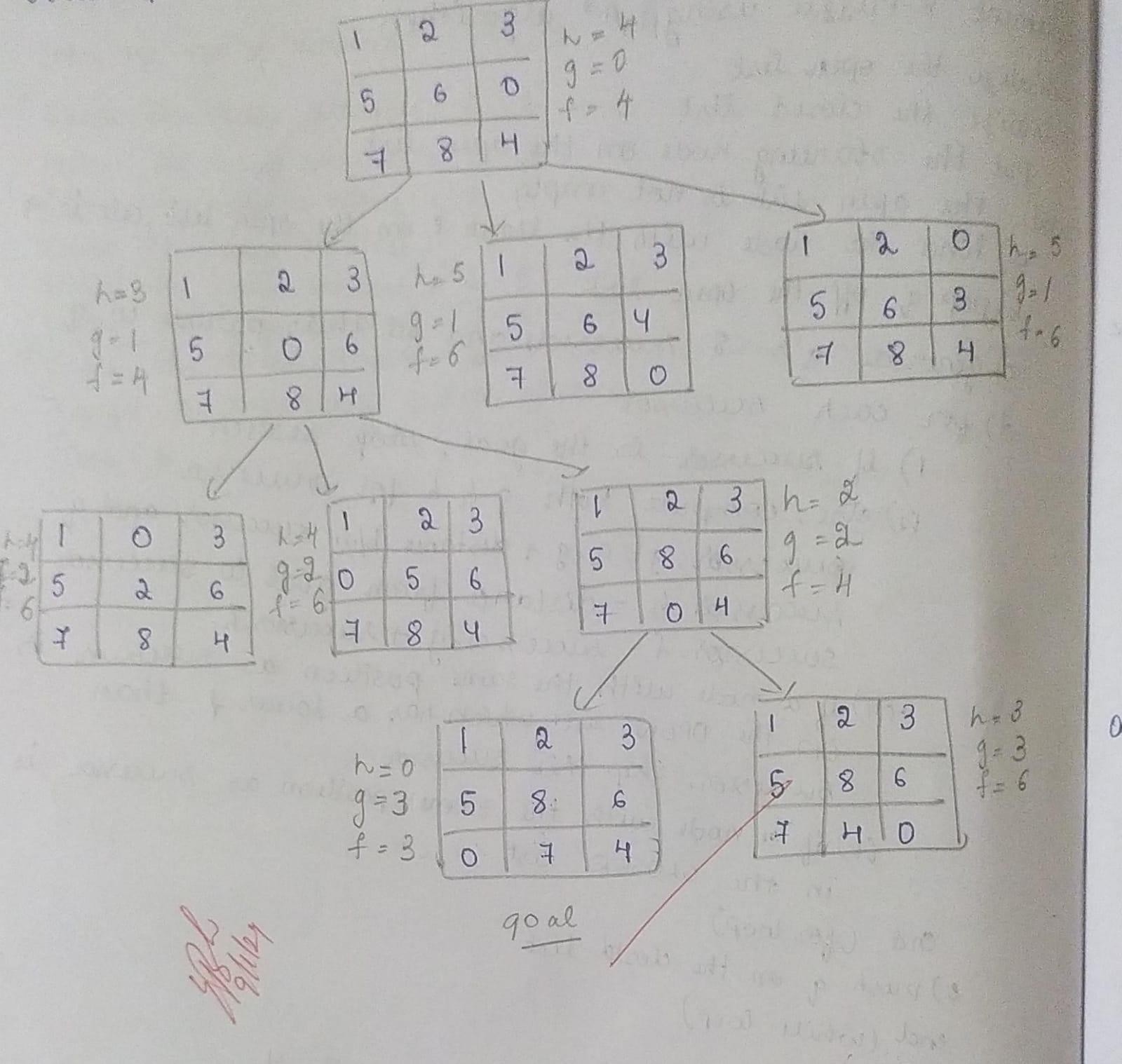
print('Steps to reach goal:',totalmoves) visit = len(state) - visited

print('Total nodes visited: ',visit, "\n") print('Total generated:', len(state))

Output:



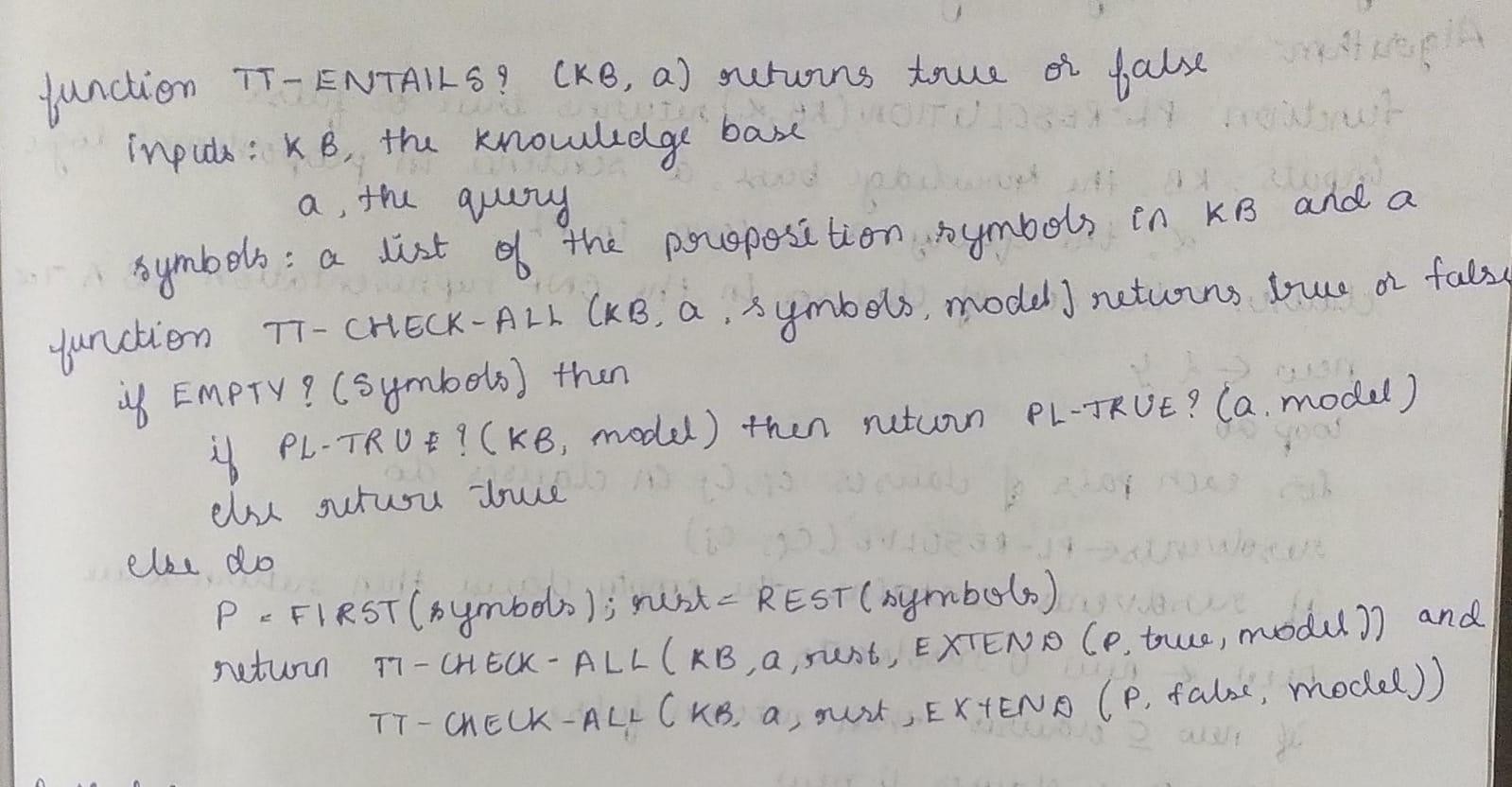
State-Space Diagram:



# Program- 6

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

Algorithm:



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

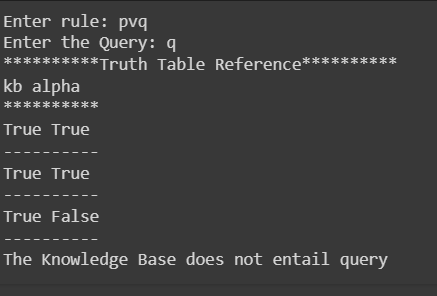
return val2 or val1

input\_rules() ans = entailment() if ans:

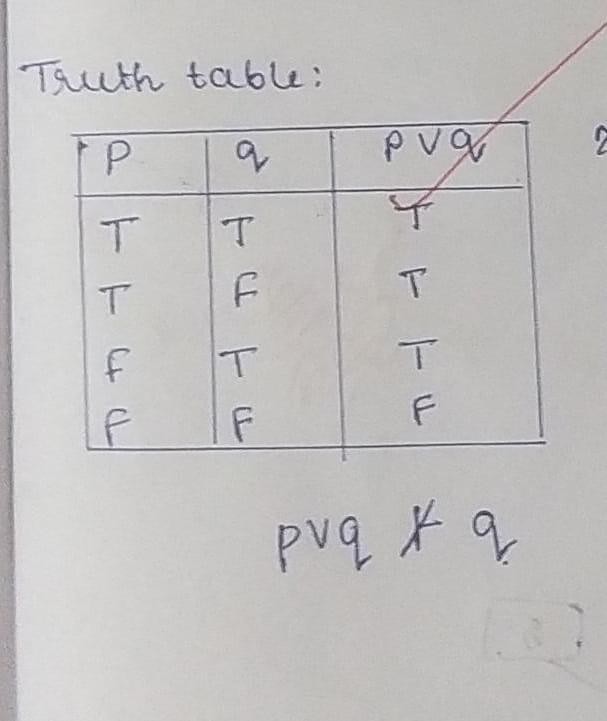
print("The Knowledge Base entails query") else:

print("The Knowledge Base does not entail query")

Output:



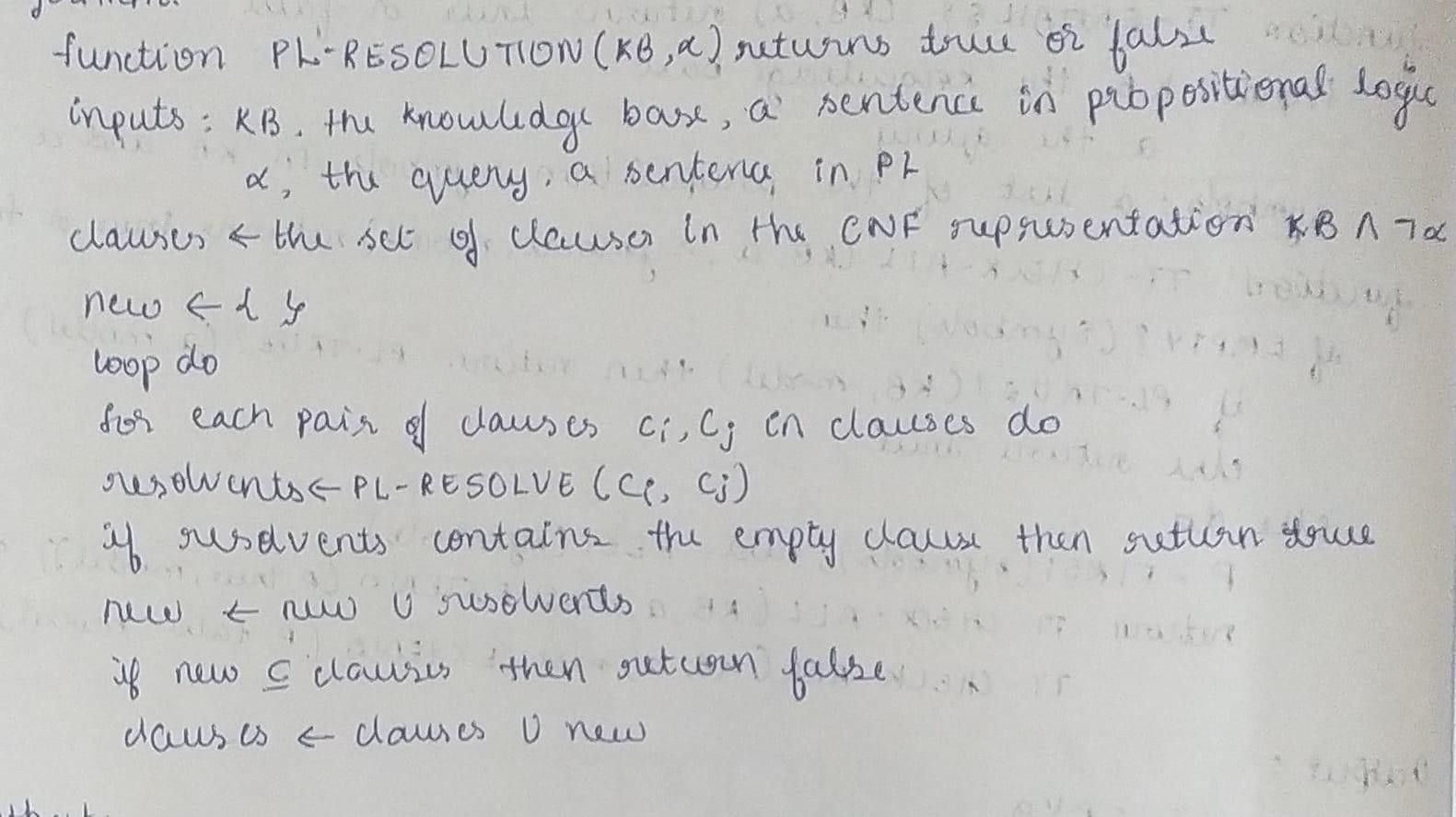
Proof:



# Program- 7

Create a knowledge base using prepositional logic and prove the given query using resolution

Algorithm:



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 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

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(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

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')

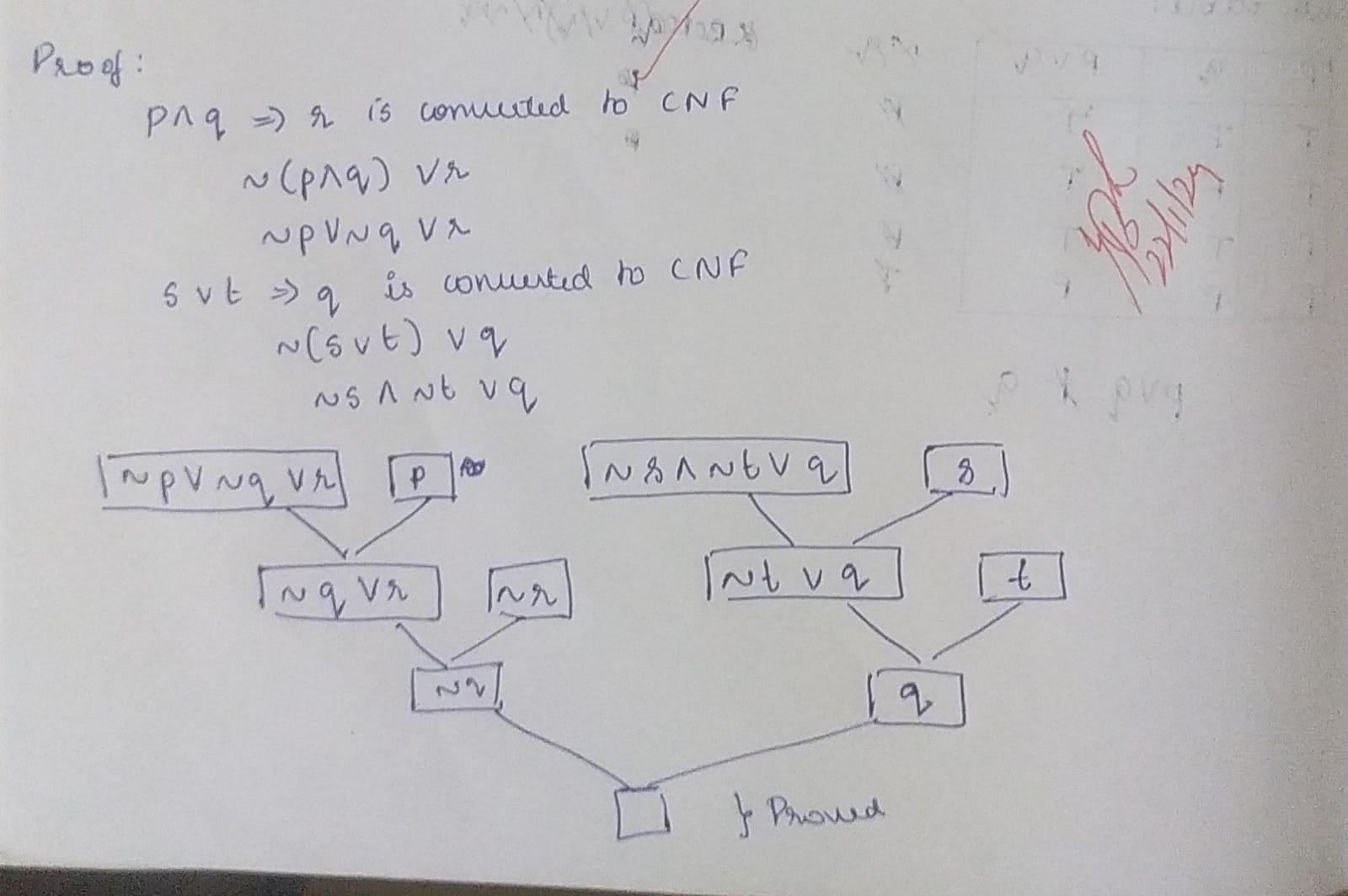
TELL('s')

print(ASK('r'))

Output:



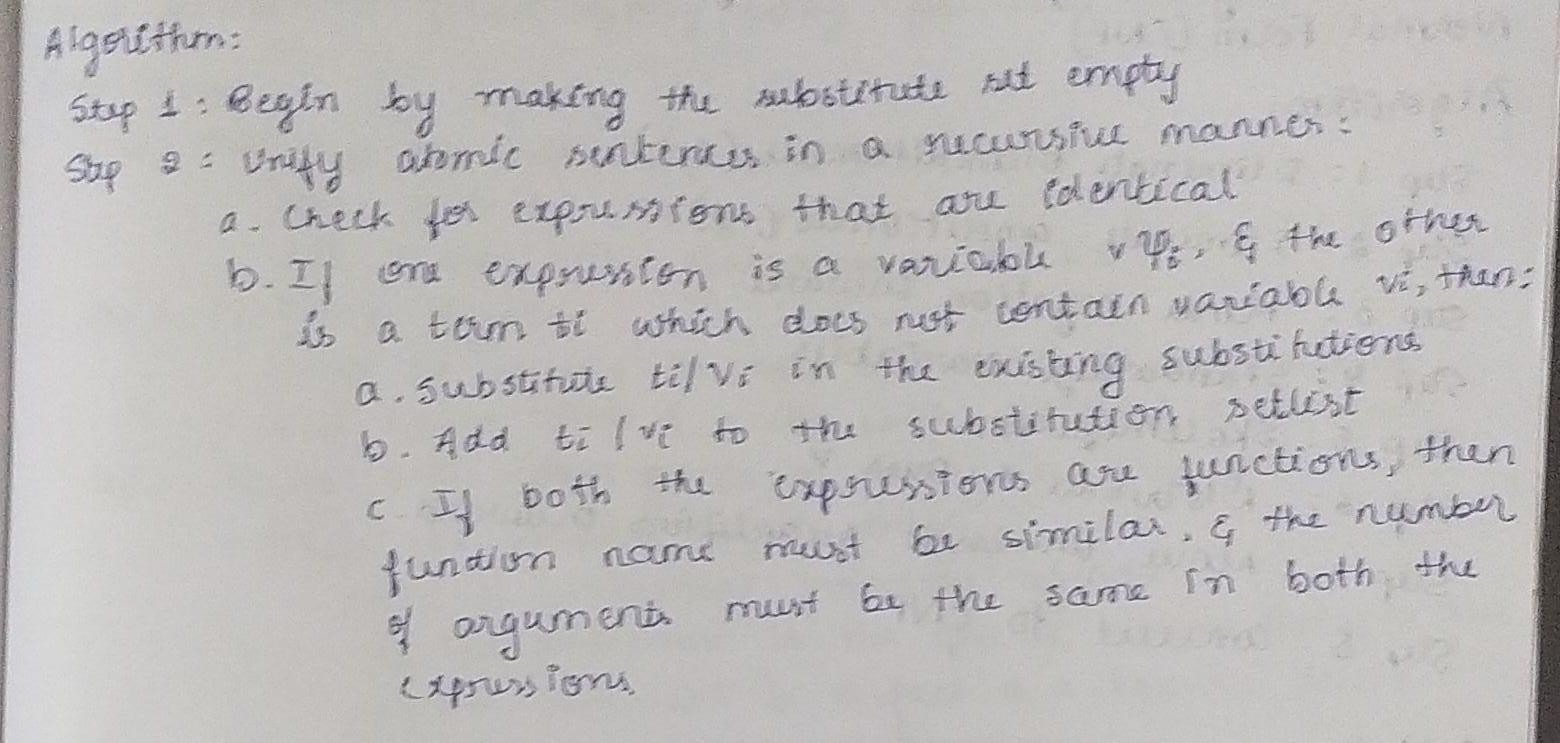
Proof:



# Program- 8

Implement unification in first order logic

Algorithm:



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():

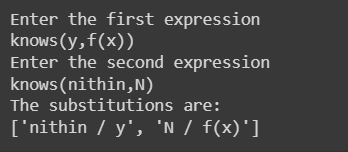
print("Enter the first expression") e1 = input()

print("Enter the second expression") e2 = input()

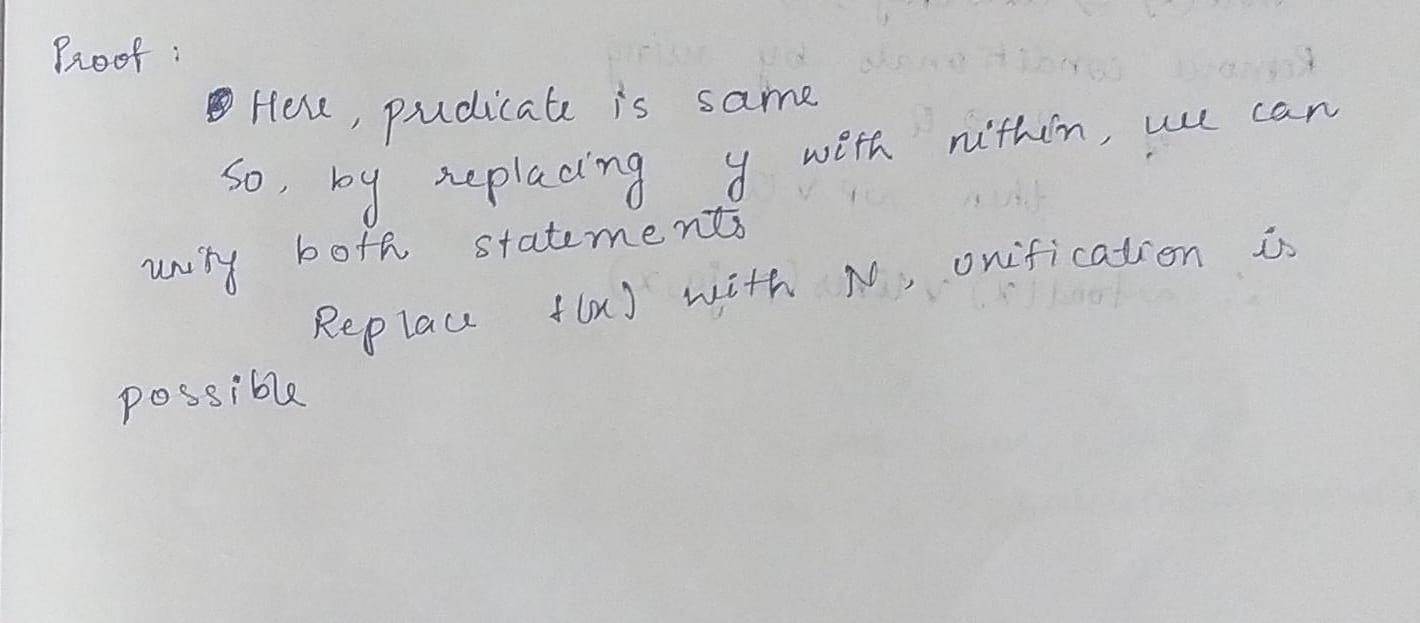
substitutions = unify(e1, e2) print("The substitutions are:")

print([' / '.join(substitution) for substitution in substitutions]) main()

Output:



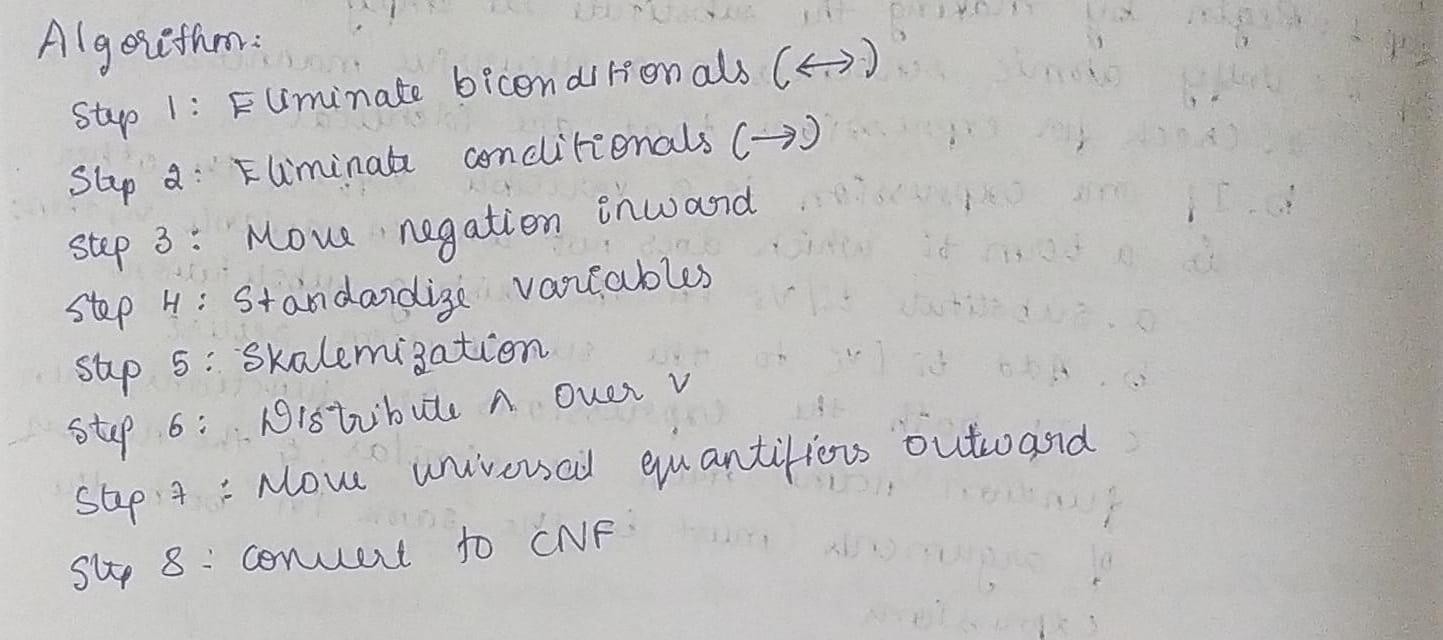
Proof:



# Program- 9

Convert a given first order logic statement into Conjunctive Normal Form (CNF).

Algorithm:



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

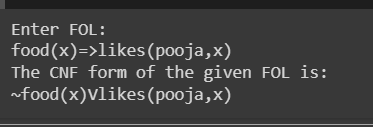
def main():

print("Enter FOL:") fol = input()

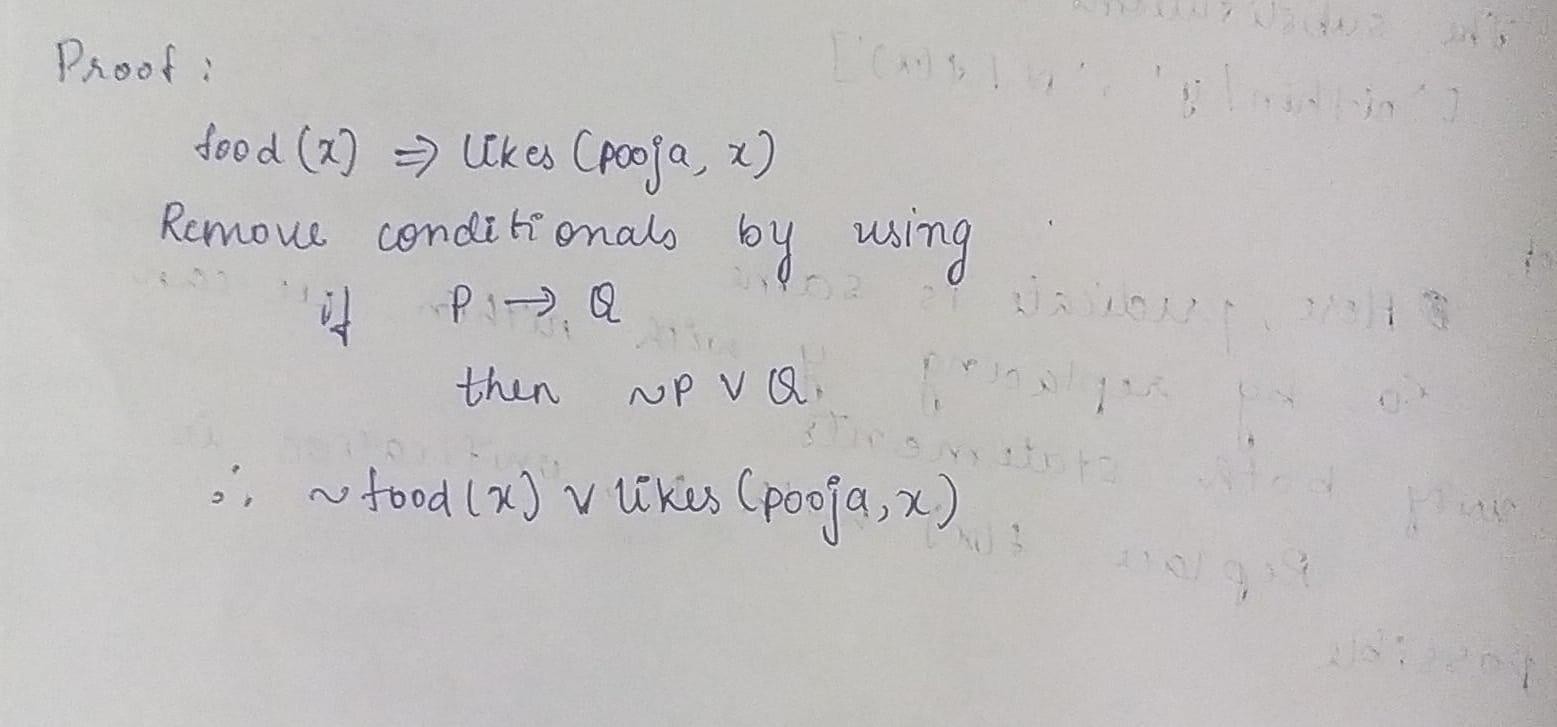
print("The CNF form of the given FOL is: ") print(Skolemization(fol\_to\_cnf(fol)))

main()

Output:



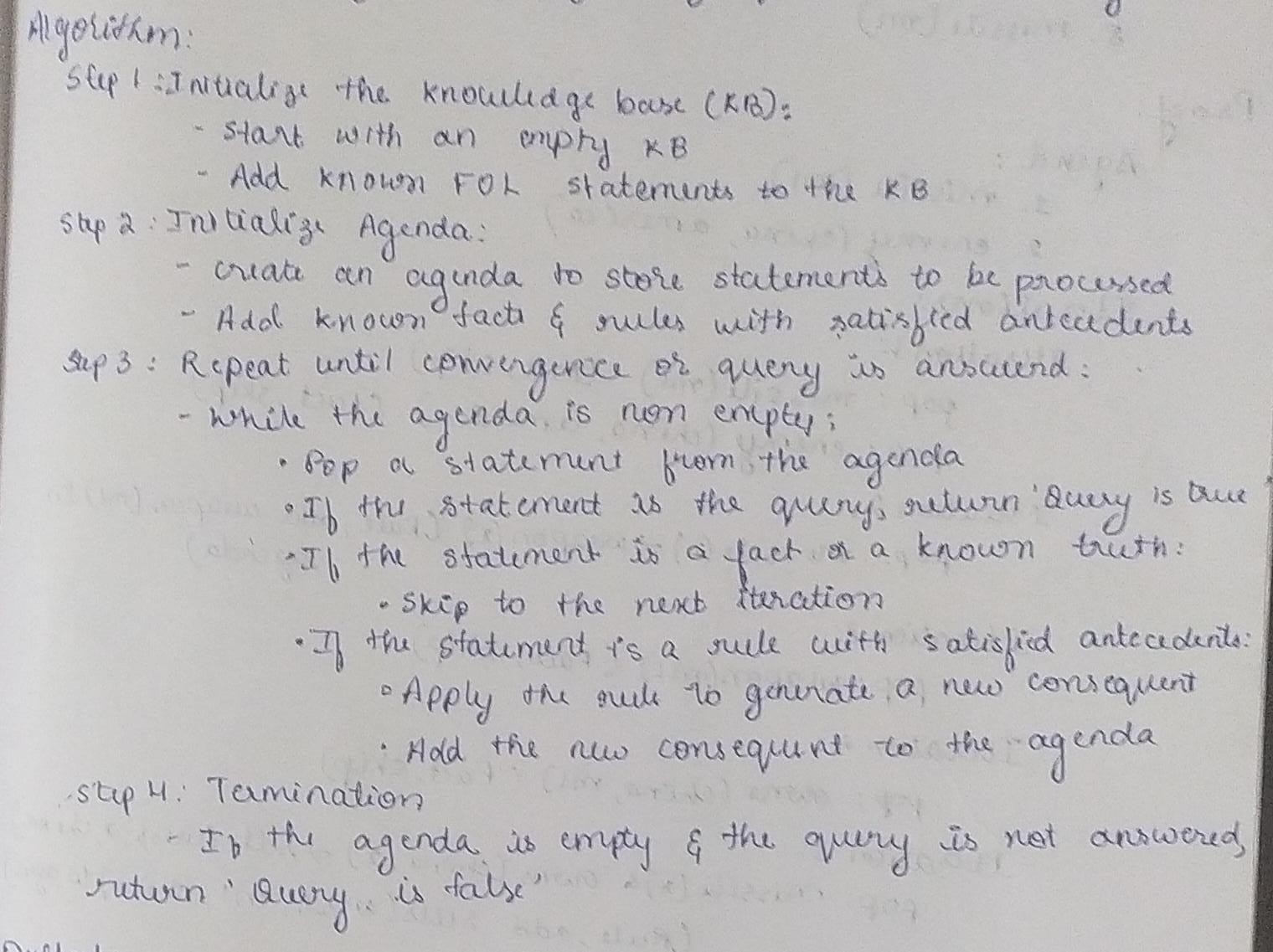
Proof:



# Program-10

Create a knowledge base consisting of first order logic statements and prove the given query using forward reasoning.

Algorithm:



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:

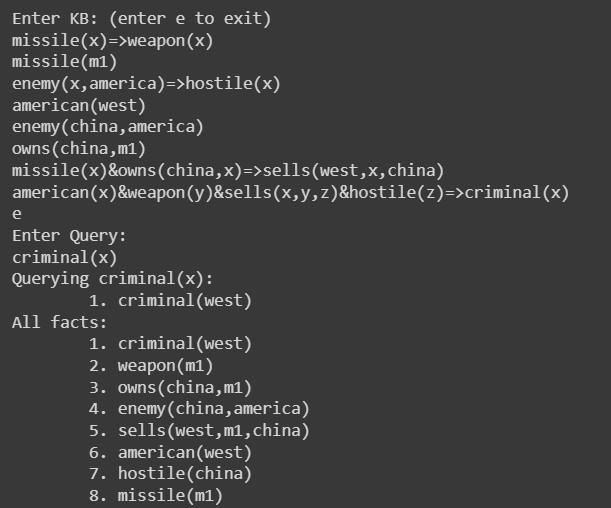
t = input() if(t == 'e'):

break kb.tell(t)

print("Enter Query:") q = input() kb.query(q) kb.display()

main()

Output:



Proof:

