VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB REPORT
on

Artificial Intelligence LAB

Submitted by

RAHUL C SHIRUR (1BM21CS157)

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
Oct-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 lab” carried out by **RAHUL C SHIRUR (1BM21CS157),** who is a 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. The Lab report has been approved as it satisfies the academic requirements in respect of a **Artificial Intelligence lab (22CS5PCAIN)**work prescribed for the said degree.

Dr. Pallavi G B
Assistant Professor
Department of CSE
BMSCE, Bengaluru

Dr.Jyothi S Nayak
Professor and Head
Department of CSE
BMSCE, Bengaluru

`

Program-1

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

Algorithm:

function Reflex-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():
 # 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.")
```

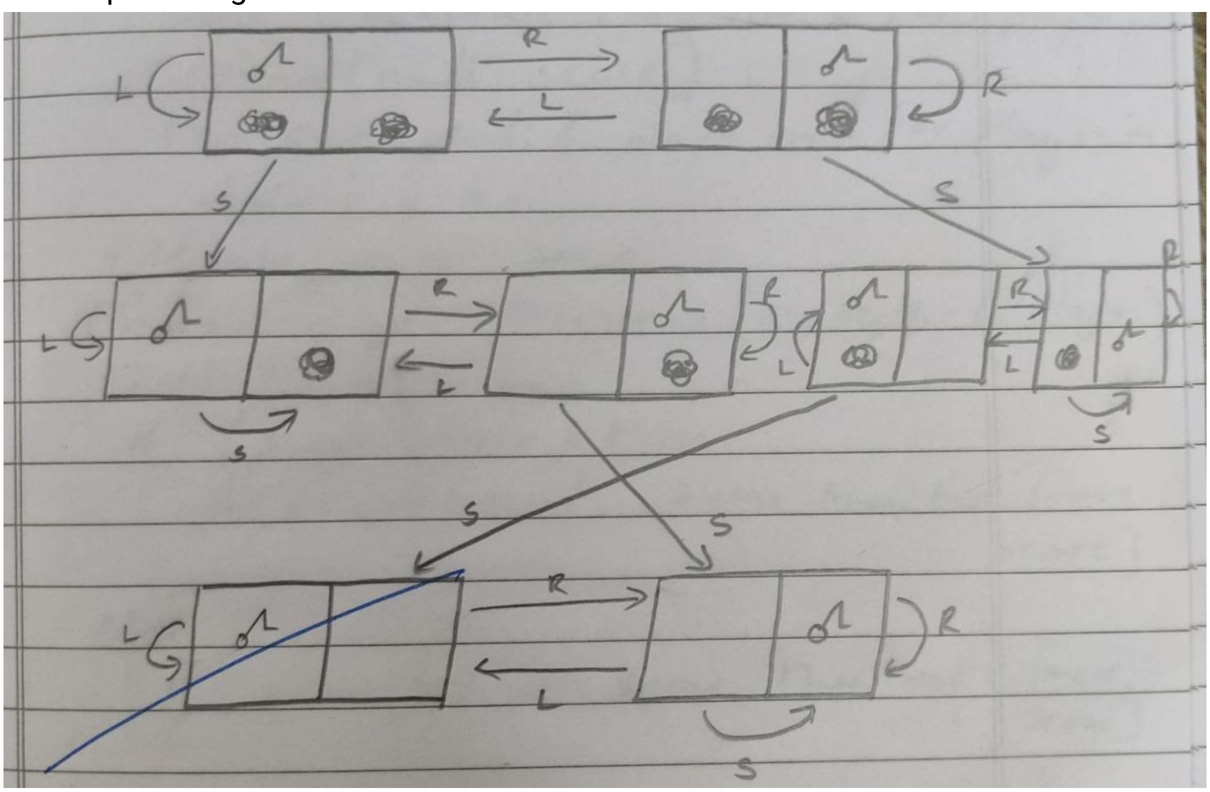
```
            print("Moving right to the Location B. ")
            cost += 1 #cost for moving right
            print("COST for moving RIGHT" + str(cost))
            # suck the dirt and mark it as clean
            goal_state['B'] = '0'
            cost += 1 #cost for suck
            print("COST 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:

```
Enter Location of VacuumA
Enter status of A1
Enter status of other room1
Initial Location Condition{'A': '0', 'B': '0'}
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.
```

State-Space Diagram:



Program-2

Explore the working of Tic Tac Toe using Min max strategy

Algorithm:

```
min max (state, depth, player)
        if(player = max) then
        best = [null, -infinity]
   else
        best = [null, + infinity]
     if (depth = 0  or game over) then
     score  = evaluate this state for player
     return [null, score]
     for each valid move m for player in
        state s do.
   execute move 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):
```

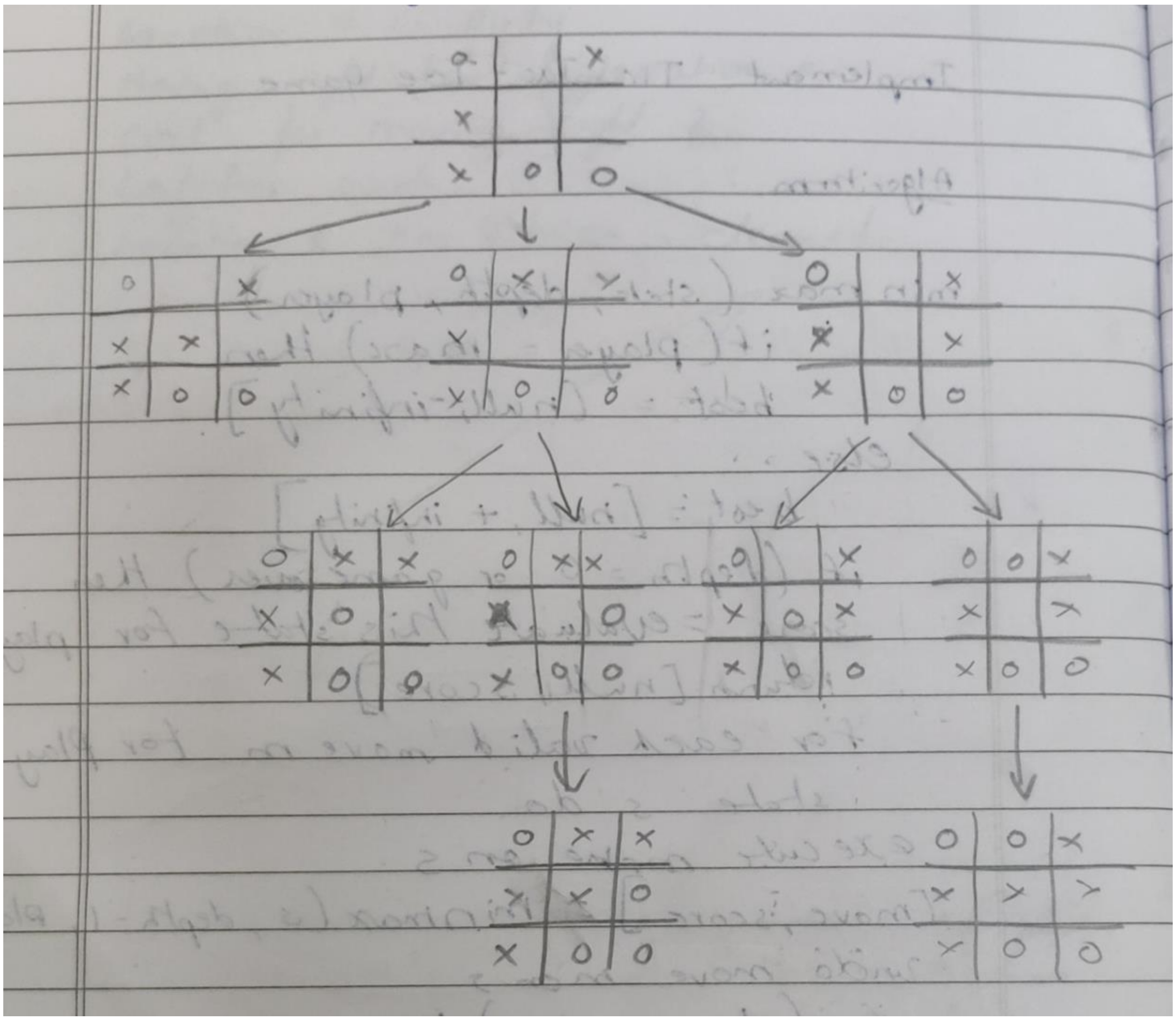
```
    if all([board[i][j] == player for j in range(3)]) or all([board[j][i]
== player for j in range(3)]):
      return True

  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
```

```python
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): 0
Enter column (0, 1, or 2): 1
O|X|
-----
 | |
-----
 | |
-----
Enter row (0, 1, or 2): 1
Enter column (0, 1, or 2): 2
O|X|
-----
 | |X
-----
O| |
-----
Enter row (0, 1, or 2): 2
Enter column (0, 1, or 2): 1
O|X|
-----
O| |X
-----
O|X|
-----
AI wins!
```

State-Space Diagram:



Program-3

Implement the 8 Puzzle Breadth First Search Algorithm.

Algorithm:

```
function BFS = 8 puzzle problem (src, target):
    queue [].
    queue.append (src)
    exp: [ ]
  while len(queue) > 0
  source = queue.pop(0)
  exp.append (source)
  print (source)
 if source == target:
    print ("success")
    return
  possiblemoves = possible_moves(source, exp).
```

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")
```

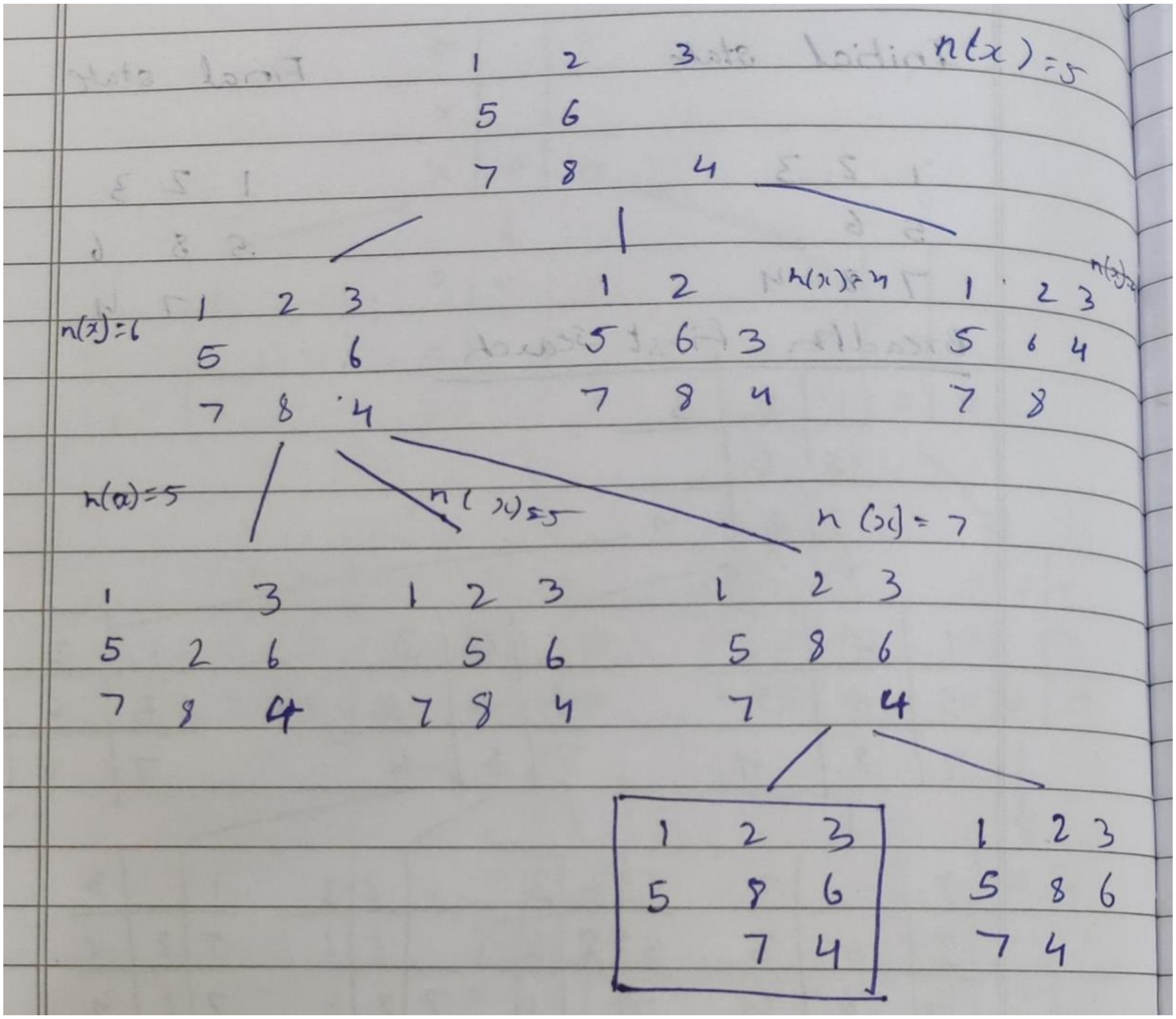
```python
            print("Cost:",cost)
            return

        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:

```
Queue contents:
1 | 2 | 3
5 | 6 | 0
7 | 8 | 4

Queue contents:
1 | 2 | 0
5 | 6 | 3
7 | 8 | 4

Queue contents:
1 | 2 | 3
5 | 6 | 4
7 | 8 | 0

Queue contents:
1 | 2 | 3
5 | 0 | 6
7 | 8 | 4

Queue contents:
1 | 0 | 2
5 | 6 | 3
7 | 8 | 4

Queue contents:
1 | 2 | 3
5 | 6 | 4
7 | 0 | 8

Queue contents:
1 | 0 | 3
5 | 2 | 6
7 | 8 | 4

Queue contents:
1 | 2 | 3
5 | 8 | 6
7 | 0 | 4
```

```
Queue contents:
1 | 6 | 2
5 | 0 | 3
7 | 8 | 4

Queue contents:
0 | 1 | 2
5 | 6 | 3
7 | 8 | 4

Queue contents:
1 | 2 | 3
5 | 0 | 4
7 | 6 | 8

Queue contents:
1 | 2 | 3
5 | 6 | 4
0 | 7 | 8

Queue contents:
0 | 1 | 3
5 | 2 | 6
7 | 8 | 4

Queue contents:
1 | 3 | 0
5 | 2 | 6
7 | 8 | 4

Queue contents:
1 | 2 | 3
5 | 8 | 6
0 | 7 | 4

success
Cost: 16
```

State-Space Diagram:



Program-4

Implement Iterative deepening search algorithm.

Algorithm:

- Algorithm-

function interative-deepening search (problem)
returns a solution or failure.
For depth = 0 to ∞
result ← depth_limited_search (problem,
depth)
if result ≠ 'cut off' then return result.

Code:

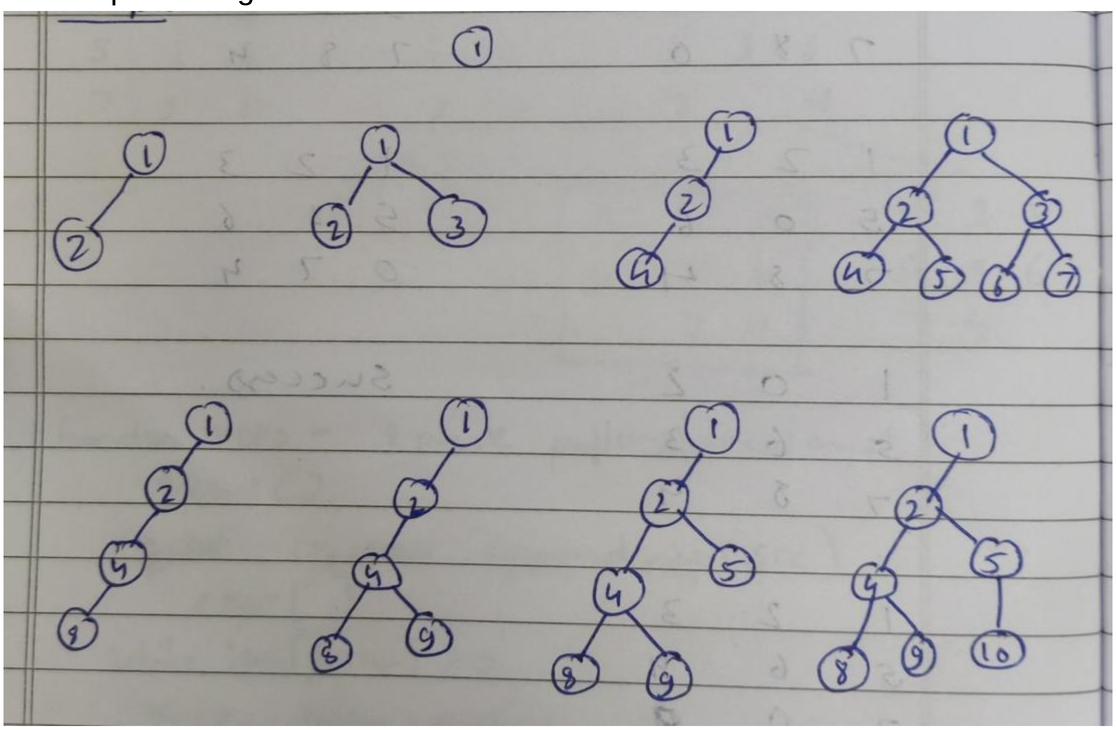
```
from collections import defaultdict
cost=0
class Graph:
  def __init__(self,vertices):
    self.V = vertices
    self.graph = defaultdict(list)
  def addEdge(self,u,v):
    self.graph[u].append(v)
  def DLS(self,src,target,maxDepth):
    if src == target :
      return True
    if maxDepth <= 0 : 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):
```

```python
      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:

```
Enter the number of verices:7
Enter the first vertex:0
Enter the second vertex:1
Enter the first vertex:0
Enter the second vertex:2
Enter the first vertex:1
Enter the second vertex:3
Enter the first vertex:1
Enter the second vertex:4
Enter the first vertex:2
Enter the second vertex:5
Enter the first vertex:2
Enter the second vertex:6
Enter the target vertex:6
Enter the max depth:3
Target is NOT reachable from source within 1
Target is NOT reachable from source within 2
Target is reachable from source within 3
COST:6
```

State-Space Diagram:



Program-5

Implement A* for 8 puzzle problem

Algorithm:

```
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 != '-':
                temp += 1
        return temp;

while true :
    Cur = self.open[0]
    for i in cur.data :
        for j in i:
            Print(j)
        Print(" ")

    if(self.h(cur-data, goal == 0);
        break.
```

Code:

```
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():
                    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)],
dtstate)
                    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

                    q = np.array([(len(state) - 1, fn)], dtpriority)
                    priority = np.append(priority, q, 0)
                      # Checking if the node in openstates are matching
the goal state.
                    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]
```

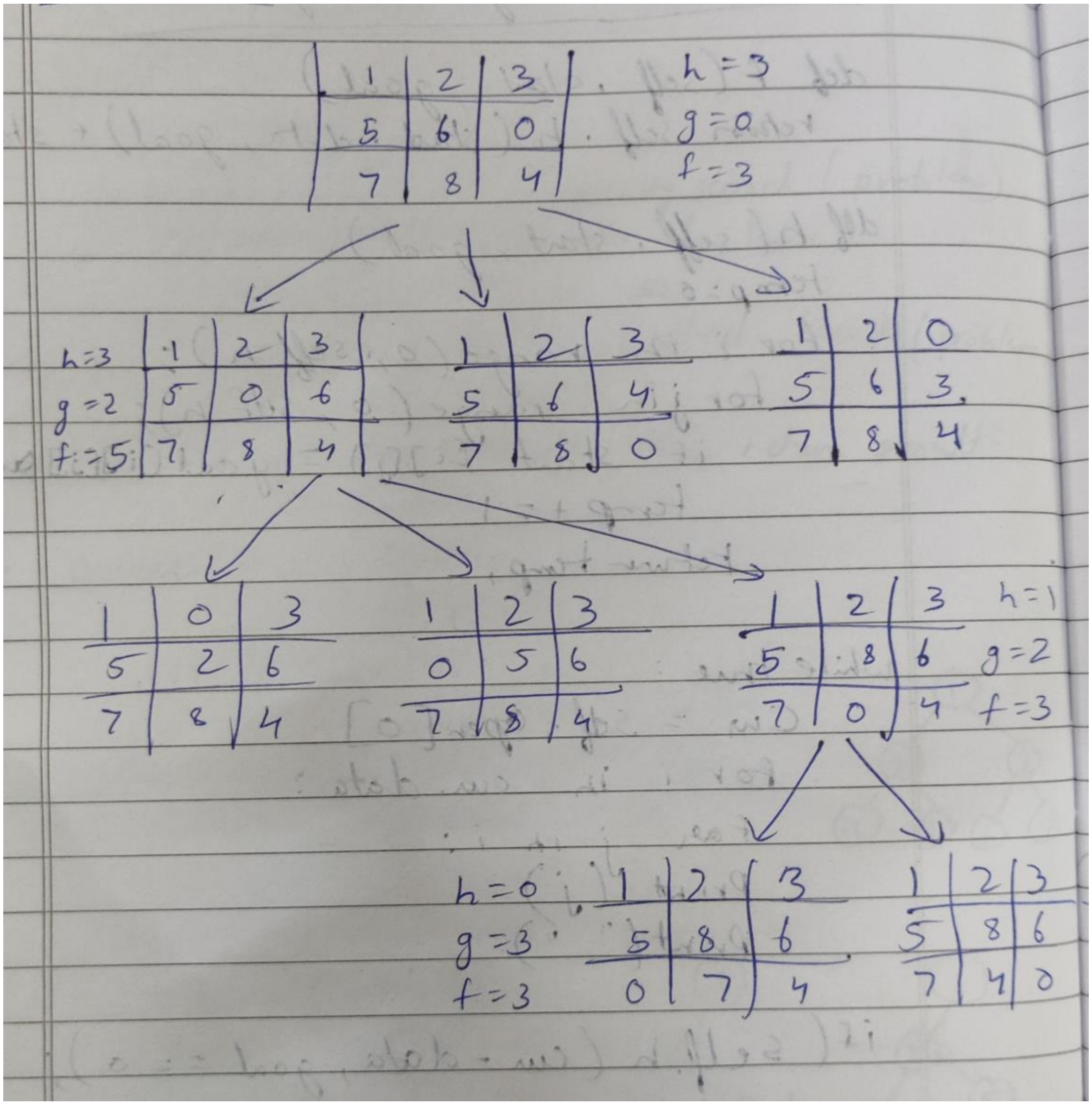
```
                # The check function is called, if the node has been
previously explored or not.
                if ~(np.all(list(state['puzzle']) == openstates,
1)).any():
                    end_time = time.time()
                    if (( end_time - start_time ) > 2):
                        print(" The 8 puzzle is unsolvable \n")
                        break
                    # calls the Misplaced_tiles function to calcuate the
cost
                    hn = misplaced_tiles(coordinates(openstates), costg)
                    # generate and add new state in the list
                    q = np.array([(openstates, position, gn, hn)],
dtstate)
                    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

                    q = np.array([(len(state) - 1, fn)], dtpriority)
                    priority = np.append(priority, q, 0)
                    # Checking if the node in openstates are matching the
goal state.
                    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)
```

```python
 # 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:

```
 Input vals from 0-8 for start state
enter vals :1
enter vals :2
enter vals :3
enter vals :5
enter vals :6
enter vals :0
enter vals :7
enter vals :8
enter vals :4
 Input vals from 0-8 for goal state
Enter vals :1
Enter vals :2
Enter vals :3
Enter vals :5
Enter vals :8
Enter vals :6
Enter vals :0
Enter vals :7
Enter vals :4
1. Manhattan distance
2. Misplaced tiles2
 The 8 puzzle is solvable

   1 2 3
   5 6 0
   7 8 4

   1 2 3
   5 0 6
   7 8 4

   1 2 3
   5 8 6
   7 0 4

   1 2 3
   5 8 6
   0 7 4
Steps to reach goal: 3
Total nodes visited:  3

Total generated: 8
```

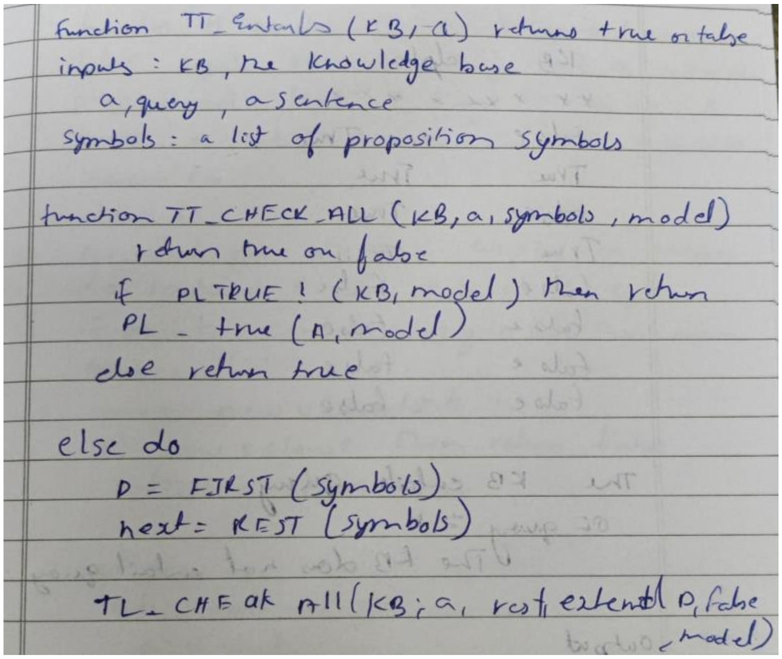
State-Space Diagram:



Program-6

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

Algorithm:



function TT_Entails (KB, α) returns true or false
inputs : KB, the knowledge base
α, query, a sentence
symbols : a list of proposition symbols

function TT_CHECK_ALL (KB, α, symbols, model)
return true or false
if PL TRUE ! (KB, model) then return
PL _ true (A, model)
else return true

else do
P = FIRST (symbols)
next= REST (symbols)

TL_ CHECK all (KB; α, 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,'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 rule: pvq
Enter the Query: q
***********Truth Table Reference***********
kb alpha
***********
True True
----------
True True
----------
True False
----------
The Knowledge Base does not entail query
```

Proof:

Truth table:

p	q	p v q
T	T	T
T	F	T
F	T	T
F	F	F

$p \vee q \nvDash q$

Program-7

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

Algorithm:

function PL-Resolution(KB, α): returns true or
false inputs: KB, knowledge base, a sentence
propositional logic α, query, a sentence, PL
clauses ← set of clauses in CNF representation
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 empty clause then return
true
new ← new ∪ resolvents
if new ⊆ clause then return false
clause ← clauses ∪ 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 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):
```

```python
        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))
```

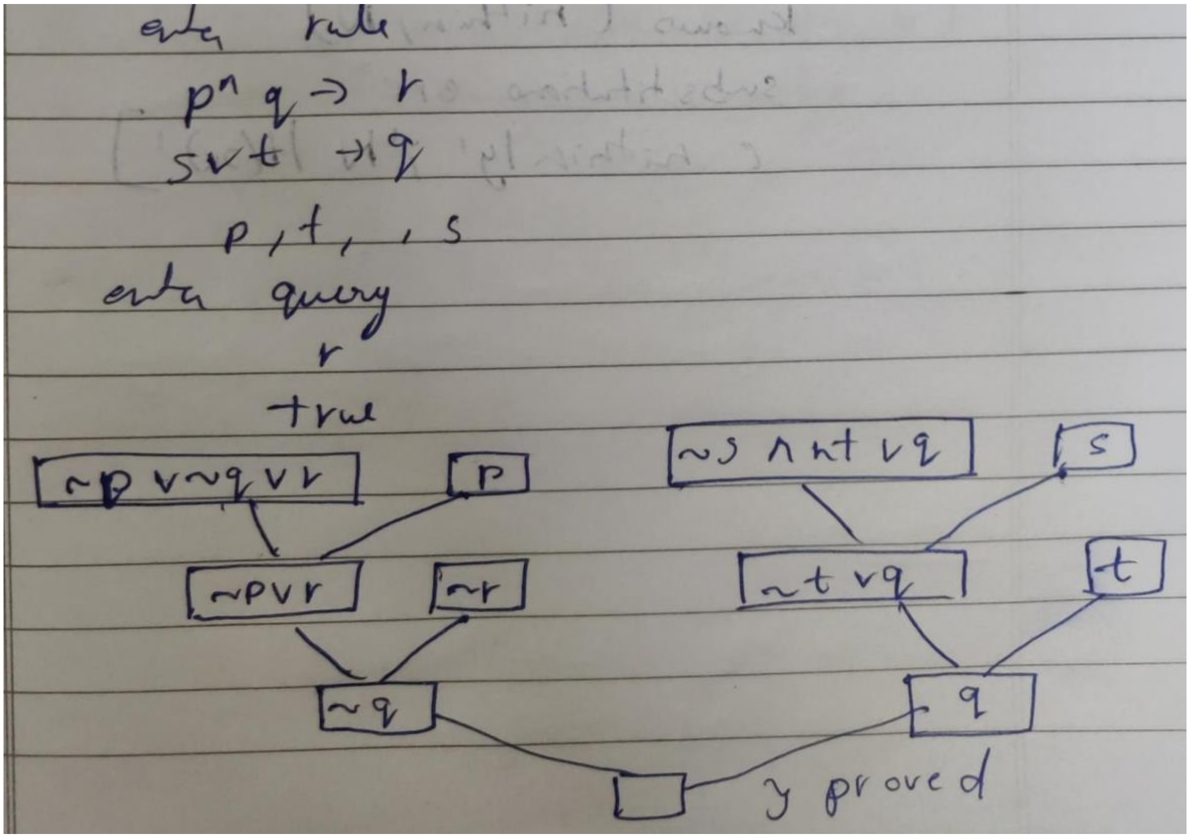
```
            return result

        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:

```
True
```

Proof:



Program-8

Implement unification in first order logic

Algorithm:

Step1 : begin by making substitue set empty
Step2: unify atomic sentences in a recursive manner :
a: check for expression that are identical
b. If one expression is a variable v_i & the in terms t_i which does not contain variable v_i, then :
a: substitute t_i/v_i in existing substitution
b: Add t_i/v_i to substitution setting
c: If both expressions are functions, the function name must be same arguments must be same in both the expressions

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 []
```

```python
    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:

```
Enter the first expression
knows(y,f(x))
Enter the second expression
knows(nithin,N)
The substitutions are:
['nithin / y', 'N / f(x)']
```

Proof:

Proof :

- Here, predicate is same
So, by replacing y with nithin, we can unify both statements
Replace f(x) with N, unification is possible

Program-9

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

Algorithm:

S1: Eliminate biconditionals
S2: Eliminate conditionals
S3: Move negation inwards
S4: Standarize variables
S5: Skolemization
S6: Distribute ∧ V
S7: move universals quantifiers outward
S8: Convert to CNF

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:

```
Enter FOL:
food(x)=>likes(pooja,x)
The CNF form of the given FOL is:
~food(x)Vlikes(pooja,x)
```

Proof:

Proof : food(x) => likes(pooja, x)
remove conditionals using
$p \to q$
then $\sim p \vee q$
$\therefore \sim$ food(x) $\vee$ likes(pooja, x)

Program-10

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

Algorithm:

S1 : initialize KB,
start with an empty KB
add known FOL to KB
S2 : initialize agenda
create agenda to store statements to be
proceeded
add known facts with satisfied
antecedents
S3 : Repeat unit query is answered.
- write the agenda is non empty.
- pop a statement from agenda
if statement is a fact or known truth
- skip to next iteration
- if statement is a rule with statisfied
antecedents
apply the rule to generate a new
consequent. add the new consequent
to agenda

Step 4 : Termination
if agenda is empty & quey is not
answered.

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
```

```python
        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]
```

```python
                    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)")
```

```python
    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. criminal(west)
        2. weapon(m1)
        3. owns(china,m1)
        4. enemy(china,america)
        5. sells(west,m1,china)
        6. american(west)
        7. hostile(china)
        8. missile(m1)
```

Proof:

pop : missile (m_1) : (fact, skip)
pop : enemy (china ; america) : (fact, skip)

Iteration 2 :
pop : missile(x) ⇒ weapon (x)

iteration 3 :
pop = weapon (m_1)

intration 4 :
owns (china, M_1)

itterations 5 :
pop (missile (x) ∧ own(China, x) ⇒
sells (west , x, China)
rule, add, sells (west , m, China)

iteration 6 :
pop : sells (west, m1, china) (fact, stop)

iteration 7 :
pop (american(x) ∧ weapon only) sells(x,y,z)
and hostile (z) ⇒ Criminal (x) rule, add
criminal (west) to agenda !

iteration 9 :
pop : Criminal (west) (query found, return 'query'