AI Lab Report

Submitted by

DHANUSH R (1BM20CS041)
Batch: A3

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

BACHELOR OF ENGINEERING
in
COMPUTER SCIENCE AND ENGINEERING

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

2022-2023

Lab-Program-1

AIM: Implement Tic –Tac –Toe Game.

Objective: Tic –Tac –Toe Game is a very simple two-player game where both players get to choose any of the symbols between X and O. This game is played on a 3X3 grid board and one by one each player gets a chance to mark its respective symbol on the empty spaces of the grid.

Code:

```
from math import inf as infinity
from random import choice
import platform
import time
from os import system
HUMAN = -1
COMP = +1
board = [
    [0, 0, 0],
    [0, 0, 0],
    [0, 0, 0],
]

def evaluate(state):
       if wins(state, COMP):
         score = +1
    elif wins(state, HUMAN):
         score = -1
    else:
         score = 0
return score

def wins(state, player):
   win_state = [
         [state[0][0], state[0][1], state[0][2]],
         [state[1][0], state[1][1], state[1][2]],
         [state[2][0], state[2][1], state[2][2]],
         [state[0][0], state[1][0], state[2][0]],
         [state[0][1], state[1][1], state[2][1]],
         [state[0][2], state[1][2], state[2][2]],
         [state[0][0], state[1][1], state[2][2]],
         [state[2][0], state[1][1], state[0][2]],
    ]
    if [player, player, player] in win_state:
         return True
    else:
         return False
```

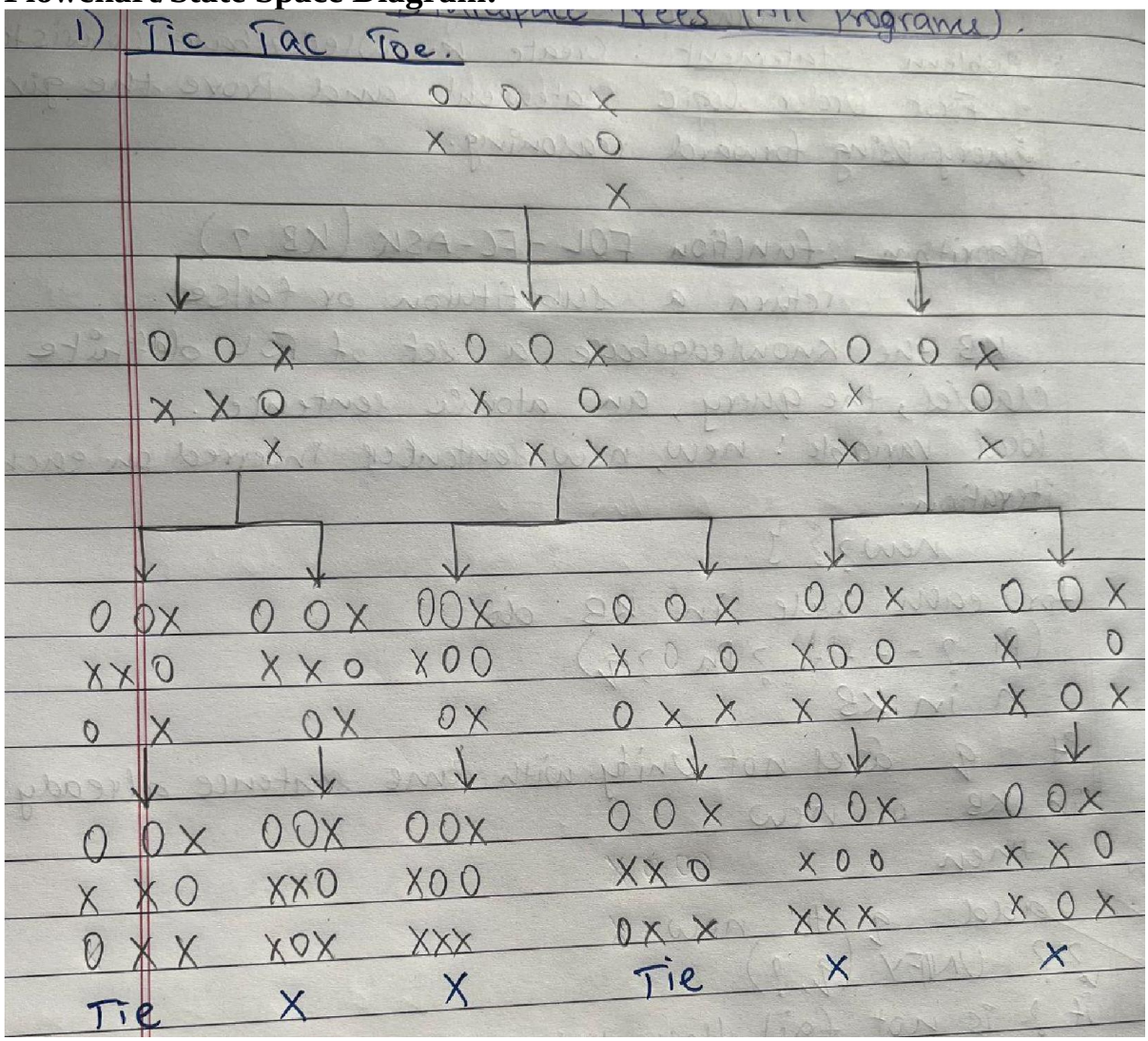
```
def game_over(state):
    return wins(state, HUMAN) or wins(state, COMP)

def empty_cells(state):
       cells = []
 for x, row in enumerate(state):
       for y, cell in enumerate(row):
          if cell == 0:
             cells.append([x, y])
 return cells

def valid_move(x, y):
   if [x, y] in empty_cells(board):
      return True
   else:
      return False

def set_move(x, y, player):
     if valid_move(x, y):
      board[x][y] = player
      return True
   else:
      return False

def minimax(state, depth, player):
   if player == COMP:
      best = [-1, -1, -infinity]
   else:
      best = [-1, -1, +infinity]
   if depth == 0 or game_over(state):
      score = evaluate(state)
      return [-1, -1, score]
   for cell in empty_cells(state):
      x, y = cell[0], cell[1]
      state[x][y] = player
      score = minimax(state, depth - 1, -player)
      state[x][y] = 0
      score[0], score[1] = x, y
      if player == COMP:
         if score[2] > best[2]:
            best = score  # max value
      else:
         if score[2] < best[2]:
            best = score  # min value
   return best
def clean():
   os_name = platform.system().lower()
   if 'windows' in os_name:
      system('cls')
```

```
    else:
        system('clear')

def render(state, c_choice, h_choice):
  chars = {
        -1: h_choice,
        +1: c_choice,
        0: ' '
    }
    str_line = '---------------'
    print('\n' + str_line)
    for row in state:
        for cell in row:
            symbol = chars[cell]
            print(f'| {symbol} |', end='')
        print('\n' + str_line)
def ai_turn(c_choice, h_choice):
       depth = len(empty_cells(board))
    if depth == 0 or game_over(board):
        return
    clean()
    print(f'Computer turn [{c_choice}]')
    render(board, c_choice, h_choice)
    if depth == 9:
        x = choice([0, 1, 2])
        y = choice([0, 1, 2])
    else:
        move = minimax(board, depth, COMP)
        x, y = move[0], move[1]
    set_move(x, y, COMP)
    time.sleep(1)

def human_turn(c_choice, h_choice):
        depth = len(empty_cells(board))
    if depth == 0 or game_over(board):
        return
    move = -1
    moves = {
        1: [0, 0], 2: [0, 1], 3: [0, 2],
        4: [1, 0], 5: [1, 1], 6: [1, 2],
        7: [2, 0], 8: [2, 1], 9: [2, 2],
    }
    clean()
    print(f'Human turn [{h_choice}]')
    render(board, c_choice, h_choice)
    while move < 1 or move > 9:
        try:
            move = int(input('Use numpad (1..9): '))
            coord = moves[move]
```

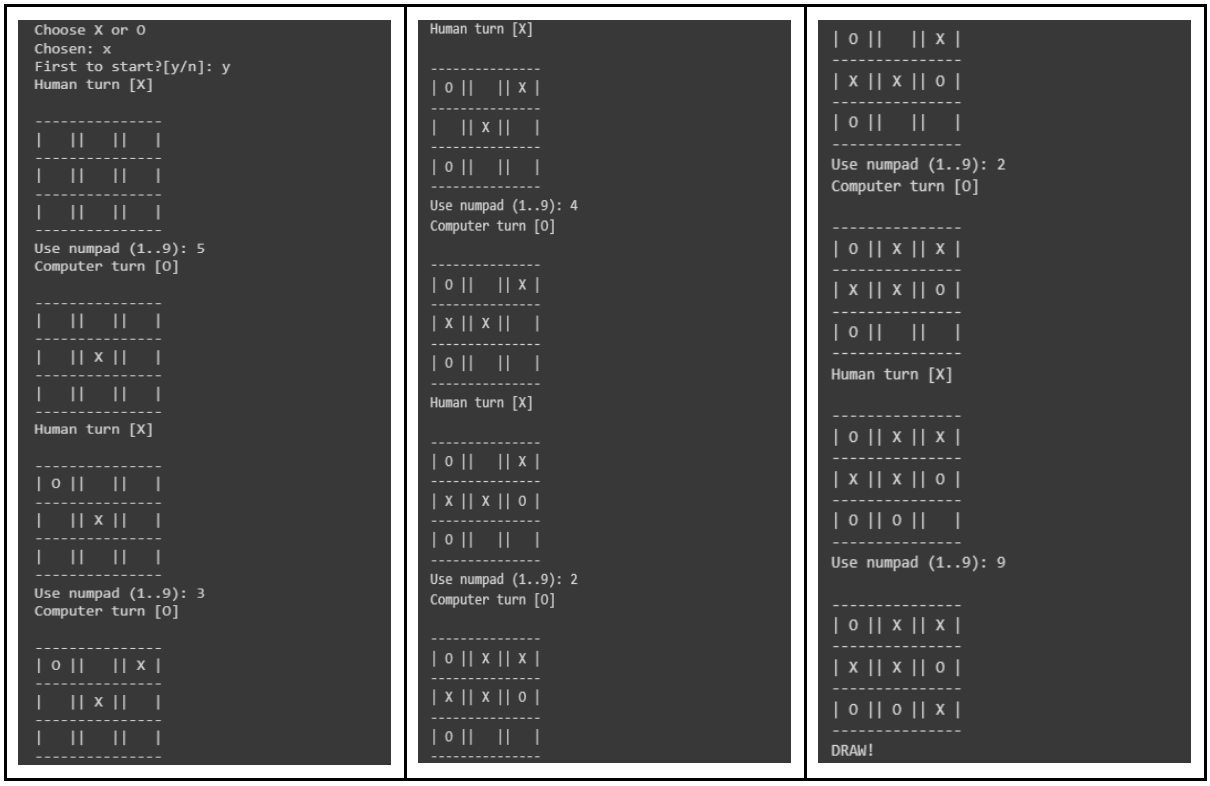
```
            can_move = set_move(coord[0], coord[1], HUMAN)
            if not can_move:
                print('Bad move')
                move = -1
        except (EOFError, KeyboardInterrupt):
            print('Bye')
            exit()
        except (KeyError, ValueError):
            print('Bad choice')

def main():
    clean()
    h_choice = ''  # X or O
    c_choice = ''  # X or O
    first = ''  # if human is the first
    while h_choice != 'O' and h_choice != 'X':
        try:
            print('')
            h_choice = input('Choose X or O\nChosen: ').upper()
        except (EOFError, KeyboardInterrupt):
            print('Bye')
            exit()
        except (KeyError, ValueError):
            print('Bad choice')
    # Setting computer's choice
    if h_choice == 'X':
        c_choice = 'O'
    else:
        c_choice = 'X'
    # Human may starts first
    clean()
    while first != 'Y' and first != 'N':
        try:
            first = input('First to start?[y/n]: ').upper()
        except (EOFError, KeyboardInterrupt):
            print('Bye')
            exit()
        except (KeyError, ValueError):
            print('Bad choice')
    # Main loop of this game
    while len(empty_cells(board)) > 0 and not game_over(board):
        if first == 'N':
            ai_turn(c_choice, h_choice)
            first = ''
        human_turn(c_choice, h_choice)
        ai_turn(c_choice, h_choice)
    # Game over message
    if wins(board, HUMAN):
        clean()
```

```
        print(f'Human turn [{h_choice}]')
        render(board, c_choice, h_choice)
        print('YOU WIN!')
    elif wins(board, COMP):
        clean()
        print(f'Computer turn [{c_choice}]')
        render(board, c_choice, h_choice)
        print('YOU LOSE!')
    else:
        clean()
        render(board, c_choice, h_choice)
        print('DRAW!')
    exit()
if __name__ == '__main__':
    main()
```

Flowchart/State Space Diagram:



Output:


Choose X or O
Chosen: x
First to start?[y/n]: y
Human turn [X]
Use numpad (1..9): 5
Computer turn [O]
Human turn [X]
Use numpad (1..9): 3
Computer turn [O]
Human turn [X]
Use numpad (1..9): 4
Computer turn [O]
Human turn [X]
Use numpad (1..9): 2
Computer turn [O]
Use numpad (1..9): 2
Computer turn [O]
Human turn [X]
Use numpad (1..9): 9
DRAW!

Lab-Program-2

AIM: Solve 8 puzzle problem.

Objective: The objective is to place the numbers on tiles to match the final configuration using the empty space. We can slide four adjacent (left, right, above, and below) tiles into the empty space.

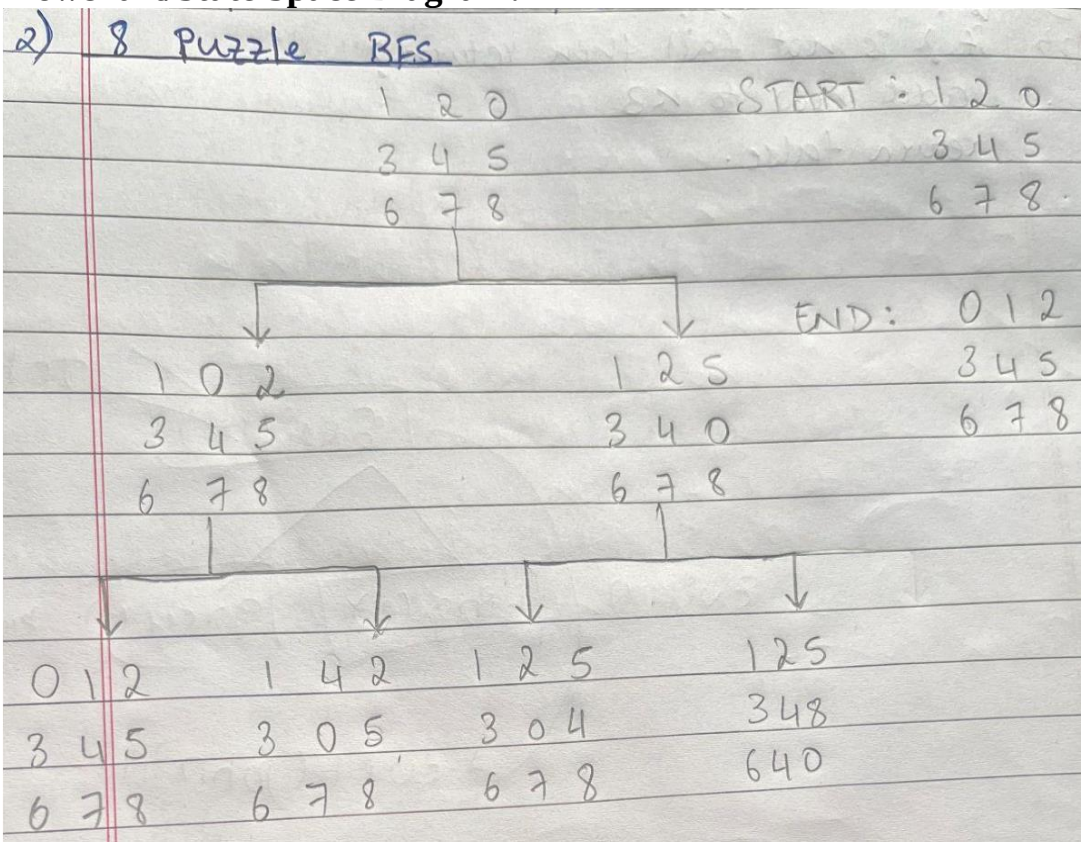
Code:

```
import numpy as np
import math
import time
# start = np.array([1,2,3,5,6,0,7,8,4]).reshape(3,3) #--->>> Hard array to solve
start = np.array([1,2,3,0,4,6,7,5,8]).reshape(3,3) #--->>> Easy array than above one to solve

goal = np.array([1,2,3,4,5,6,7,8,0]).reshape(3,3) #--->>> Goal state to achieve

def actions_array(array):
        goal = np.array([1,2,3,4,5,6,7,8,0]).reshape(3,3)
        possible_actions = []
        new_arrays = {}
        for i in range(len(array)):
                for j in range(len(array)):
                        if array[i][j] == 0:

                                #for moving up
                                if i > 0:
                                        up_array = array.copy()
                                        up_array[i][j], up_array[i-1][j] = up_array[i-1][j],
up_array[i][j]
                                        if not np.array_equal(up_array, start):
                                                new_arrays["up"] = up_array


                                #for moving down
                                if i < len(array) - 1:
                                        down_array = array.copy()
                                        down_array[i][j], down_array[i+1][j] =
down_array[i+1][j], down_array[i][j]
                                        if not np.array_equal(down_array, start):
                                                new_arrays["down"] = down_array


                                #for moving right
                                if j < len(array) - 1:
                                        right_array = array.copy()
                                        right_array[i][j], right_array[i][j+1] =
right_array[i][j+1], right_array[i][j]
```

```
                                        if not np.array_equal(right_array, start):
                                                new_arrays["right"] = right_array


                                #for moving left
                                if j > 0 :
                                        left_array = array.copy()
                                        left_array[i][j], left_array[i][j-1] = left_array[i][j-1],
left_array[i][j]
                                        if not np.array_equal(left_array, start):
                                                new_arrays["left"] = left_array


        return new_arrays



#H value by calculating number of misplaced tiles
def h_value(array):

        s = sum(abs((val-1)%3 - i%3) + abs((val-1)//3 - i//3)
      for i, val in enumerate(array.reshape(1,9)[0]) if val)

        return s


def main():
        run = True
        prev_step = []
        array = start.copy()
        ola = None
        count = 0

        tic = time.time()
        while run:

                h={}

                if ola is not None:
                        array = ola


                act = actions_array(array)

                for keys, values in act.items():
                        h[keys]=h_value(values)


                #find the smallest h value and its key in the dict
```

```
                new_dic =  dict(sorted(h.items(), key=lambda item: item[1]))
                res = list(new_dic.items())[0]
                r, v = res[0], res[1]

                if not prev_step:
                        prev_step.append(['start_array', array])

                else:
                        for i in range(len(prev_step)):
                                if np.array_equal(act[r], prev_step[i][1]):
                                #check if the 2nd value in dic is = to the lowest or not
                                #we are taking only the top two smallest
                                        new_h = list(new_dic.items())[1]
                                        r, v = new_h[0], new_h[1]


                if np.array_equal(act[r], goal):
                        print("\n")
                        print("'Problem Solved !. Steps included are : \n'")

                        prev_step.append([res[0], act[r]])
                        for i in prev_step:
                                print(i[0])
                                print(i[1])
                                print("\n")
                        run = False
                        toc = time.time()
                        print("Total number of steps: " + str(count))
                        print("Total amount of time in search: " + str(toc - tic) + " second(s)")

                else:
                        prev_step.append([r, act[r]])
                        ola = act[r]
                        # prev_step[res[0]] = act[res[0]]
                        count+=1

main()
```

Flowchart/State Space Diagram:



Output:

```
Problem Solved !. Steps included are :

start_array
[[1 2 3]
 [0 4 6]
 [7 5 8]]


right
[[1 2 3]
 [4 0 6]
 [7 5 8]]


down
[[1 2 3]
 [4 5 6]
 [7 0 8]]


right
[[1 2 3]
 [4 5 6]
 [7 8 0]]


Total number of steps: 2
Total amount of time in search: 0.014129400253295898 second(s)
```

Lab-Program-3

Aim: Implement Iterative deepening search algorithm.

Objective: The Iterative Deepening Depth-First Search (also ID-DFS) algorithm is an algorithm used to find a node in a tree. This means that given a tree data structure, the algorithm will return the first node in this tree that matches the specified condition.

Code:

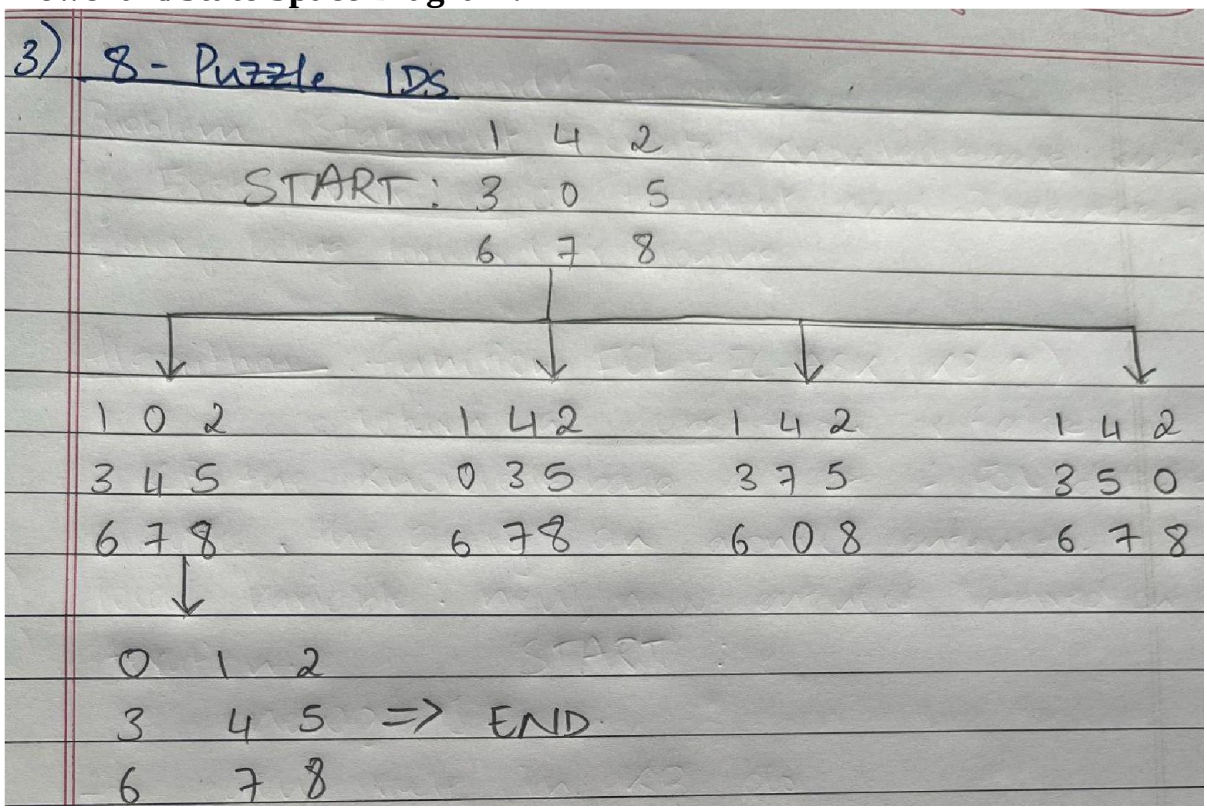
```
import time
import itertools
class Node:
def __init__(self, puzzle, last=None):
self.puzzle = puzzle
self.last = last
@property
def seq(self): # to keep track of the sequence used to get to the goal
node, seq = self, []
while node:
seq.append(node)
node = node.last
yield from reversed(seq)

@property
def state(self):
return str(self.puzzle.board) # hashable so it can be compared in sets

@property
def isSolved(self):
return self.puzzle.isSolved

@property
def getMoves(self):
return self.puzzle.getMoves

class Puzzle:
def __init__(self, startBoard):
self.board = startBoard

@property
def getMoves(self):
possibleNewBoards = []
zeroPos = self.board.index(0) # find the zero tile to determine possible moves
if zeroPos == 0:
possibleNewBoards.append(self.move(0,1))
possibleNewBoards.append(self.move(0,3))
```

```
elif zeroPos == 1:
possibleNewBoards.append(self.move(1,0))
possibleNewBoards.append(self.move(1,2))
possibleNewBoards.append(self.move(1,4))
elif zeroPos == 2:
possibleNewBoards.append(self.move(2,1))
possibleNewBoards.append(self.move(2,5))

elif zeroPos == 3:
possibleNewBoards.append(self.move(3,0))
possibleNewBoards.append(self.move(3,4))
possibleNewBoards.append(self.move(3,6))
elif zeroPos == 4:
possibleNewBoards.append(self.move(4,1))
possibleNewBoards.append(self.move(4,3))
possibleNewBoards.append(self.move(4,5))
possibleNewBoards.append(self.move(4,7))
elif zeroPos == 5:
possibleNewBoards.append(self.move(5,2))
possibleNewBoards.append(self.move(5,4))

possibleNewBoards.append(self.move(5,8))
elif zeroPos == 6:
possibleNewBoards.append(self.move(6,3))
possibleNewBoards.append(self.move(6,7))
elif zeroPos == 7:
possibleNewBoards.append(self.move(7,4))
possibleNewBoards.append(self.move(7,6))
possibleNewBoards.append(self.move(7,8))
else:
possibleNewBoards.append(self.move(8,5))
possibleNewBoards.append(self.move(8,7))
return possibleNewBoards # returns Puzzle objects (maximum of 4 at a time)

def move(self, current, to):
changeBoard = self.board[:] # create a copy
changeBoard[to], changeBoard[current] = changeBoard[current], changeBoard[to] # switch
the tiles at the passed positions
return Puzzle(changeBoard) # return a new Puzzle object


def printPuzzle(self): # prints board in 8 puzzle style
copyBoard = self.board[:]
for i in range(9):

if i == 2 or i == 5:
```

```
print((str)(copyBoard[i]))
else:
print((str)(copyBoard[i]), end=" ")
print('\n')


@property
def isSolved(self):
return self.board == [0,1,2,3,4,5,6,7,8] # goal boardclass Solver:


def __init__(self, Puzzle):
self.puzzle = Puzzle
def IDDFS(self):
def DLS(currentNode, depth):
if depth == 0:
return None
if currentNode.isSolved:
return currentNode
elif depth > 0:
for board in currentNode.getMoves:
nextNode = Node(board, currentNode)
if nextNode.state not in visited:
visited.add(nextNode.state)
goalNode = DLS(nextNode, depth - 1)
if goalNode != None: # I thought this should be redundant but itnever finds a soln if I take it
out
if goalNode.isSolved: # same as above ^
return goalNode
for depth in itertools.count():
visited = set()
startNode = Node(self.puzzle)
#print(startNode.isSolved)
goalNode = DLS(startNode, depth)
if goalNode != None:
if goalNode.isSolved:
return goalNode.seqstartingBoard = [7,2,4,5,0,6,8,3,1]
myPuzzle = Puzzle(startingBoard)
mySolver = Solver(myPuzzle)
start = time.time()
goalSeq = mySolver.IDDFS()
end = time.time()
counter = -1 # starting state doesn't count as a movefor node in goalSeq:
counter = counter + 1
node.puzzle.printPuzzle()
print("Total number of moves: " + str(counter))
```

```
totalTime = end - start
print("Total searching time: %.2f seconds" % (totalTime))
```

Flowchart/State Space Diagram:



Output:

```
[(0, 1), (0, 2), (1, 3), (1, 4), (2, 5), (2, 6), (3, 7), (3, 8), (4, 9), (4, 10)]


Enter target node to search:
8
(0, 1)
(0, 2)
(1, 3)
(1, 4)
(2, 5)
(2, 6)
(3, 7)
(3, 8)
(4, 9)
(4, 10)


target => 8 is reachable in GRAPH
```

AIM: Implement A* search algorithm.

Objective: A* Search Algorithm is a simple and efficient search algorithm that can be used to find the optimal path between two nodes in a graph. It will be used for the shortest path finding.

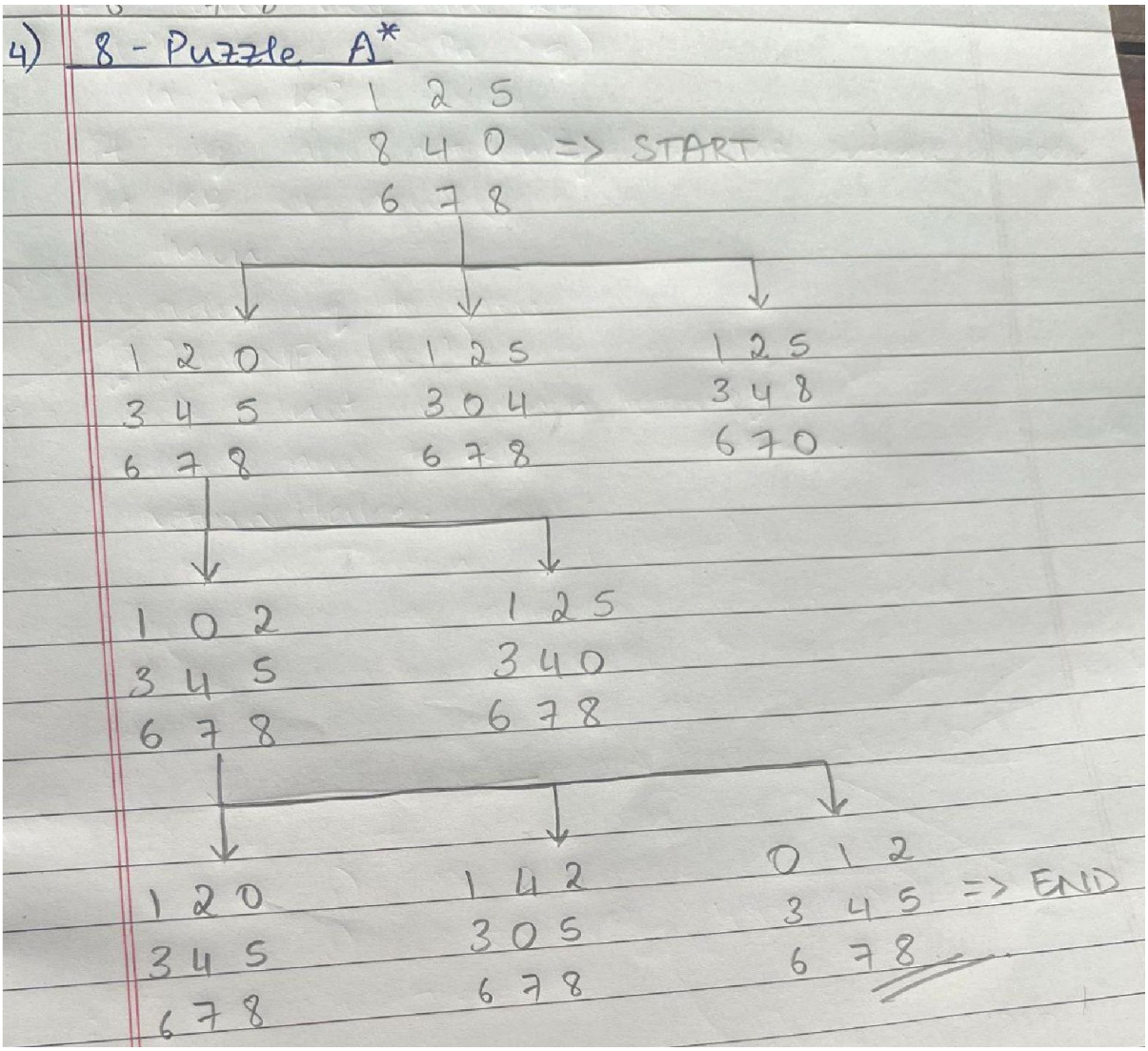
Code:

```
class Node:
    def __init__(self,data,level,fval):
        """ Initialize the node with the data, level of the node and the calculated fvalue """
        self.data = data
        self.level = level
        self.fval = fval

    def generate_child(self):
        """ Generate child nodes from the given node by moving the blank space
            either in the four directions {up,down,left,right} """
        x,y = self.find(self.data,'_')
        """ val_list contains position values for moving the blank space in either of
            the 4 directions [up,down,left,right] respectively. """
        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):
        """ Move the blank space in the given direction and if the position value are out
            of limits the return None """
        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):
```

```
        """ Copy function to create a similar matrix of the given node"""
        temp = []
        for i in root:
            t = []
            for j in i:
                t.append(j)
            temp.append(t)
        return temp

    def find(self,puz,x):
        """ Specifically used to find the position of the blank space """
        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):
        """ Initialize the puzzle size by the specified size,open and closed lists to empty """
        self.n = size
        self.open = []
        self.closed = []

    def accept(self):
        """ Accepts the puzzle from the user """
        puz = []
        for i in range(0,self.n):
            temp = input().split(" ")
            puz.append(temp)
        return puz

    def f(self,start,goal):
        """ Heuristic Function to calculate hueristic value f(x) = h(x) + g(x) """
        return self.h(start.data,goal)+start.level

    def h(self,start,goal):
        """ Calculates the different between the given puzzles """
        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):
        """ Accept Start and Goal Puzzle state"""
        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)
        """ Put the start node in the open list"""
        self.open.append(start)
        print("\n\n")
        while True:
            cur = self.open[0]
            print("")
            print("  | ")
            print("  | ")
            print(" \\\'/ \n")
            for i in cur.data:
                for j in i:
                    print(j,end=" ")
                print("")
            """ If the difference between current and goal node is 0 we have reached the goal
node"""
            if(self.h(cur.data,goal) == 0):
                break
            for i in cur.generate_child():
                i.fval = self.f(i,goal)
                self.open.append(i)
            self.closed.append(cur)
            del self.open[0]

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


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

Flowchart/State Space Diagram:



Output:

```
[(0, 1), (0, 2), (1, 3), (1, 4), (2, 5), (2, 6), (3, 7), (3, 8), (4, 9), (4, 10)]


Enter target node to search:
8
(0, 1)
(0, 2)
(1, 3)
(1, 4)
(2, 5)
(2, 6)
(3, 7)
(3, 8)
(4, 9)
(4, 10)


target => 8 is reachable in GRAPH
```

Lab-Program-5

AIM: Implement vacuum cleaner agent.

Objective: It is a goal based agent, and the goal of this agent, which is the vacuum cleaner, is to clean up the whole area.

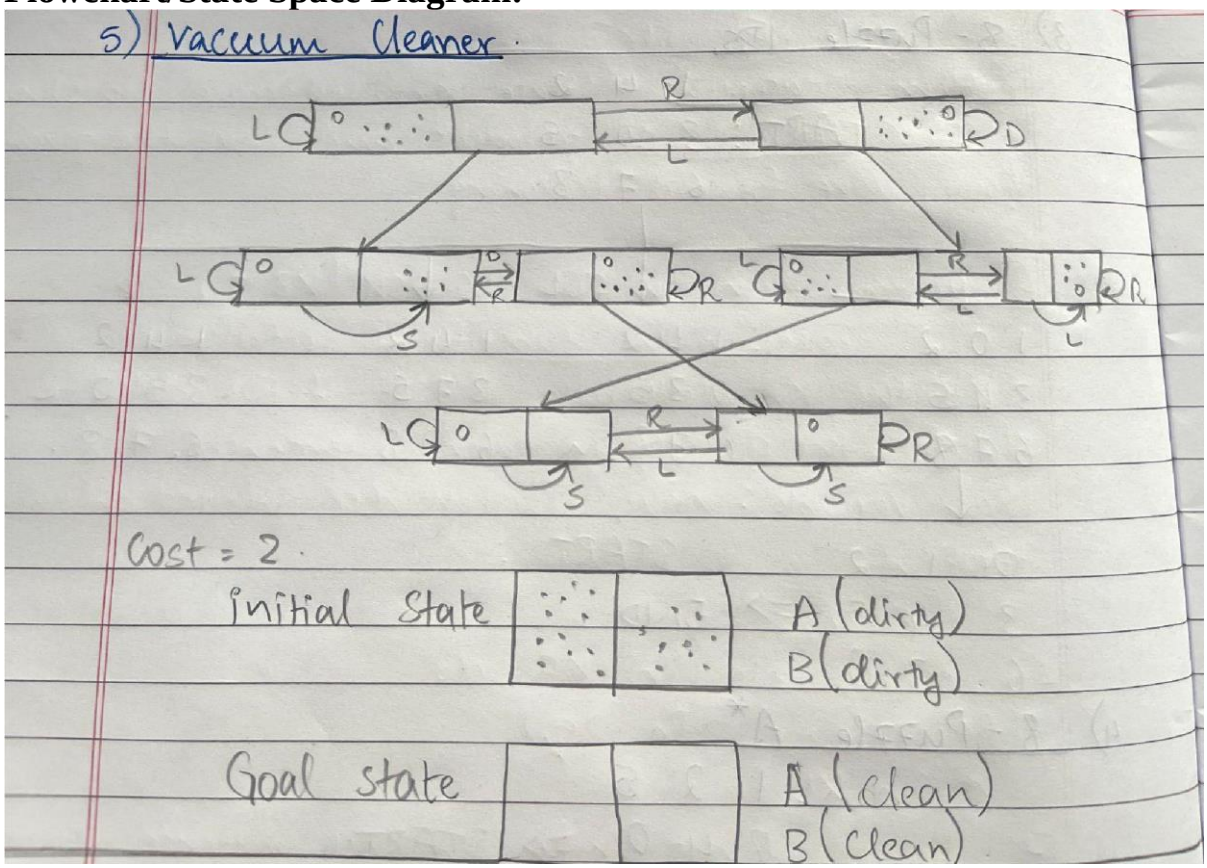
Code:

```
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()
```

Flowchart/State Space Diagram:



Output:

```
Enter Location of VacuumA
Enter status of A0
Enter status of other room1
Vacuum is placed in Location A
Location A is already clean
Location B is Dirty.
Moving RIGHT to the Location B.
COST for moving RIGHT 1
Cost for SUCK2
Location B has been Cleaned.
GOAL STATE:
{'A': '0', 'B': '0'}
Performance Measurement: 2
```

Lab-Program-6

AIM: Create a knowledgebase algorithm using prepositional logic.

Objective: A knowledge base is a collection of sentences α1,α2,...,αk that we know are true, i.e. α1∧α2∧.. ∧αk. The sentences in the knowledge base allows us to compactly specify the allowable states of the world.

Code:

```
import time
variable={'p':0,'q':1, 'r':2}
priority={'~':3,'v':1,'^':2}

def _eval(i, val1, val2):
    if i == '^':
        return val2 and val1
    return val2 or val1

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 delay(val):
    for i in range (val):
        time.sleep(1)
        print(".",end=" ")
    return

def CheckEntailment():
    kb=(input("Enter the knowledge base: "))
    query=(input("Enter the query: "))
    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]]
    postfix_kb=toPostfix(kb)
    postfix_q=toPostfix(query)
    print("\n-----------\n")
    for combination in combinations:
        eval_kb=evaluatePostfix(postfix_kb,combination)
        eval_q=evaluatePostfix(postfix_q,combination)
        delay(2)

        print(combination,":knowledgeBase=",eval_kb,":query=",eval_q)
        if(eval_kb==True):
            if(eval_q==False):
                print(" doesnt entail!!!")
                return False
    time.sleep(2)
    print(query+ " Entails "+ kb)

if __name__ == "__main__":
    CheckEntailment()
```

Output:

```
Enter the knowledge base: (pvq)^(~rvp)
Enter the query: p^r

-----------

. . [True, True, True] :knowledgeBase= True :query= True
. . [True, True, False] :knowledgeBase= True :query= False
 doesnt entail!!!
```

Lab-Program-7

AIM: Create a knowledgebase using prepositional logic and prove the given query using resolution.

Objective: A knowledge base is a collection of sentences α1,α2,...,αk that we know are true, i.e. α1∧α2∧... ∧αk. The sentences in the knowledge base allows us to compactly specify the allowable states of the world.

Code:

```
import re
def negation(term):
    return f'~{term}' if term[0] != '~' else term[1]

def revv(clause):
    if len(clause) > 2:
        t = split_exp(clause)
        return f'{t[1]}v{t[0]}'
    return

def split_exp(rule):
    exp = '(~*[PQRS])'
    terms = re.findall(exp, rule)
    return terms

split_exp('~PvR')

def con(goal, clause):
    cons = [ f'{goal}v{negation(goal)}', f'{negation(goal)}v{goal}']
    return clause in cons or revv(clause) in cons

def solve(rules, goal):
    temp = rules.copy()
    temp += [negation(goal)]
    steps = dict()
    for rule in temp:
        steps[rule] = 'Given.'
    steps[negation(goal)] = 'negationd conclusion.'
    i = 0
    while i < len(temp):
        n = len(temp)
        j = (i + 1) % n
        clauses = []
```

```
        while j != i:
            terms1 = split_exp(temp[i])
            terms2 = split_exp(temp[j])
            for c in terms1:
                if negation(c) in terms2:
                    t1 = [t for t in terms1 if t != c]
                    t2 = [t for t in terms2 if t != negation(c)]
                    gen = t1 + t2
                    if len(gen) == 2:
                        if gen[0] != negation(gen[1]):
                            clauses += [f'{gen[0]}v{gen[1]}']
                        else:
                            if con(goal,f'{gen[0]}v{gen[1]}'):
                                temp.append(f'{gen[0]}v{gen[1]}')
                                steps['] = f"Solved {temp[i]} and {temp[j]} to {temp[-1]}, hence is null.
\
                                \ncon when {negation(goal)} is true. So {goal} is true."
                                return steps
                    elif len(gen) == 1:
                        clauses += [f'{gen[0]}']
                    else:
                        if con(goal,f'{terms1[0]}v{terms2[0]}'):
                            temp.append(f'{terms1[0]}v{terms2[0]}')
                            steps['] = f"Solved {temp[i]} and {temp[j]} to {temp[-1]}, hence is null. \
                            \ncon when {negation(goal)} is true. So {goal} is true."
                            return steps
            for clause in clauses:
                if clause not in temp and clause != revv(clause) and revv(clause) not in temp:
                    temp.append(clause)
                    steps[clause] = f'Solved from {temp[i]} and {temp[j]}.'
            j = (j + 1) % n
        i += 1
    return steps

def main(rules, goal):
    rules = rules.split(' ')
    steps = solve(rules, goal)
    print('\nStep\t|Clause\t|Derivation\t')
    print('~' * 30)
    i = 1
    for step in steps:
        print(f' {i}.\t| {step}\t| {steps[step]}\t')
        i += 1

rules = input("Enter the knowledge base 'use whitespace' :")
query=input("query: ")
```

main(rules, query)

Output:

```
Enter the knowledge base 'use whitespace' :(pvq) (pvr)
query: (qvr)

Step    |Clause |Derivation
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
 1.     | (pvq) | Given.
 2.     | (pvr) | Given.
 3.     | ~(qvr)        | negationd conclusion.
```

Lab-Program-8

AIM: Implement unification in first order logic.

Objective: The goal of unification is to make two expression look like identical by using substitution.

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()
print(" ")
print("------------------- ")
print(" ")
main()
print(" ")
print("------------------- ")
print(" ")
main()
print(" ")
print("------------------- ")
print(" ")
main()
print("-------------------------- ")
print("--------------------------")
```

Output:

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

------------------

Enter the first expression
Employer(x)
Enter the second expression
Employer(Bob)
The substitutions are:
['Bob / x']

------------------

Enter the first expression
EMployer(y)
Enter the second expression
Employee(Rita)
Cannot be unified as the predicates do not match!
The substitutions are:
[]

------------------

Enter the first expression
KIng(x, Ram)
Enter the second expression
King(y, AshOka)
Cannot be unified as the predicates do not match!
The substitutions are:
[]
-------------------------
-------------------------
```

Lab-Program-9

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

Objective: A sentence represented as a conjunction of clauses is said to be conjunctive normal form or 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 == '∨':
            s[i] = '∧'
        elif c == '∧':
            s[i] = '∨'
    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] + '∨' + 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[+2], '~'
```

```
        statement = ''.join(s)
    statement = statement.replace('~[∀', '[~∀')
    statement = statement.replace('~[∃', '[~∃')
    expr = '(~[∀|∃].)'
    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

# fol = input("Enter F.O.L statement:\n")
out = fol_to_cnf(input("Enter F.O.L statement:\n"))
print("\nThe CNF form is:")
print(Skolemization(out))
```

Output:

```
Enter F.O.L statement:
∀x[study(x)∧play(x)]=>balancedLife(x)

The CNF form is:
[~study(A)∨~play(A)]∨balancedLife(A)
```

Lab-Program-10

AIm: Create a knowledgebase consisting of first order logic statements and prove the given query using forward reasoning.

Objective: The Forward-chaining algorithm starts from known facts, triggers all rules whose premises are satisfied, and add their conclusion to the known facts. This process repeats until the problem is solved.

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 exit to stop)")
    while True:
        t = input()
        if(t == 'exit'):
            break
        kb.tell(t)
    print("Enter Query:")
    q = input()
    kb.query(q)
    kb.display()

main()
```

Output:

```
Enter KB: (Enter exit to stop)
work(x)=>money(x)
work(John)
play(x,Cricket)=>happy(x)
work(x)&play(John,x)=>balanced(x)
exit
Enter Query:
balanced(x)
Querying balanced(x):
        1. balanced(John)
All facts:
        1. money(John)
        2. work(John)
        3. balanced(John)
```