**Artificial Intelligence Lab Report**

****

***Submitted by***

**Sumith UN(1BM22CS297) Batch: C4**

**Course: Artificial Intelligence Course Code: 23CS5PCAIP Sem & Section: 5F**

**BACHELOR OF ENGINEERING**

***in***

**COMPUTER SCIENCE AND ENGINEERING**

****

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

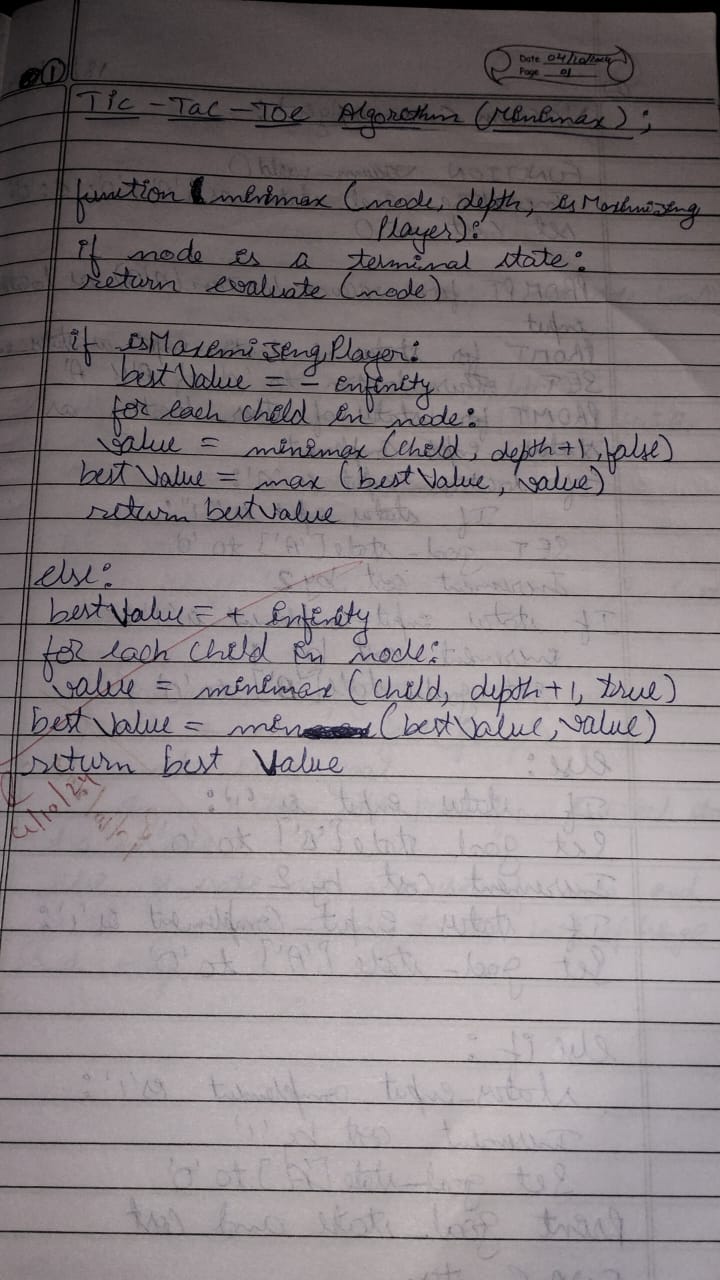
**(Autonomous Institution under VTU) BENGALURU-560019**

**2022-2023**

Table of contents

|  |  |  |
| --- | --- | --- |
| **Program Number** | **Program Title** | **Page Number** |
| **1** | **Tic-Tac\_Toe** | **3-7** |
| **2** | **8-Puzzle BFS** | **8-12** |
| **3** | **8-Puzzle IDS** | **13-17** |
| **4** | **8-Puzzle A\*** | **18-21** |
| **5** | **Vacuum Cleaner** | **22-25** |
| **6** | **Hill climbing N queens** | **26-29** |
| **7** | **Simulated Annealing** | **30-31** |
| **8** | **Knowledge Base** | **32-35** |
| **9** | **Knowledge Base -Resolution** | **36-39** |
| **10** | **Unification in First Order Logic** | **40-44** |
| **11** | **First Order Logic to Conjunctive Normal Form** | **45-49** |
| **12** | **Forward Reasoning** | **50-54** |
|  |  |  |

**Program 1 - Tic Tac toe Algorithm**

****

**Code**

import random class TicTacToe:

def init (self): self.board = []

def create\_board(self): for i in range(3):

row = []

for j in range(3): row.append('-')

self.board.append(row)

def get\_random\_first\_player(self): return random.randint(0, 1)

def fix\_spot(self, row, col, player): self.board[row][col] = player

def is\_player\_win(self, player): win = None

n = len(self.board) for i in range(n):

win = True

for j in range(n):

if self.board[i][j] != player: win = False

break

if win:

return win for i in range(n):

win = True

for j in range(n):

if self.board[j][i] != player: win = False

break if win:

return win win = True

for i in range(n):

if self.board[i][i] != player: win = False

break if win:

return win win = True

for i in range(n):

if self.board[i][n - 1 - i] != player: win = False

break if win:

return win return False

for row in self.board: for item in row:

if item == '-': return False

return True

def is\_board\_filled(self): for row in self.board:

for item in row: if item == '-':

return False return True

def swap\_player\_turn(self, player): return 'X' if player == 'O' else 'O'

def show\_board(self): for row in self.board:

for item in row: print(item, end=" ")

print() def start(self):

self.create\_board()

player = 'X' if self.get\_random\_first\_player() == 1 else 'O' while True:

print(f"Player {player} turn") self.show\_board()

row, col = list(

map(int, input("Enter row and column numbers to fix spot: ").split())) print()

self.fix\_spot(row - 1, col - 1, player) if self.is\_player\_win(player):

print(f"Player {player} wins the game!") break

if self.is\_board\_filled():

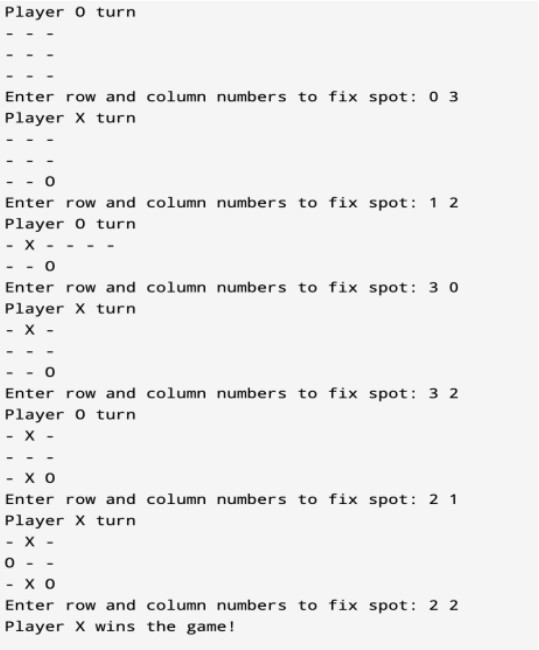
print("Match Draw!") break

player = self.swap\_player\_turn(player) print()

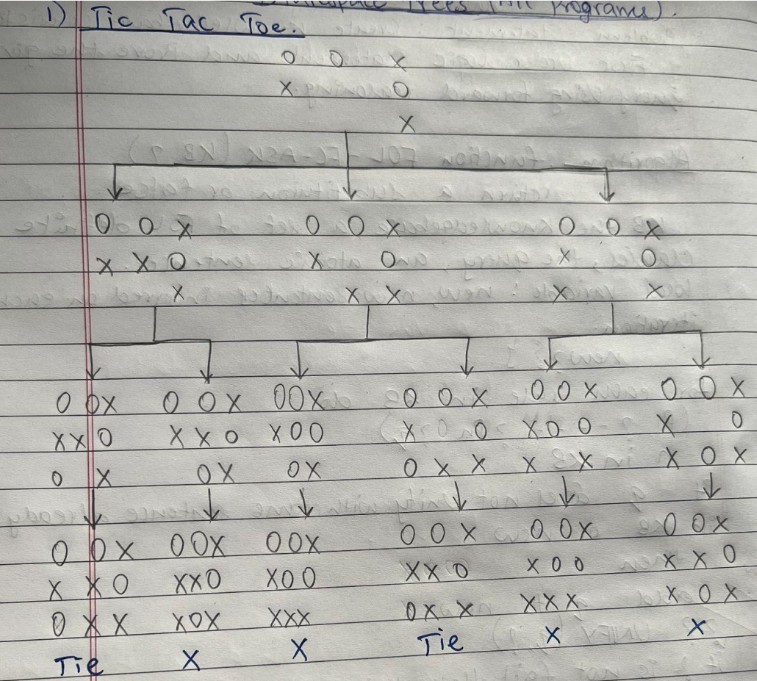
self.show\_board() tic\_tac\_toe = TicTacToe()

tic\_tac\_toe.start()

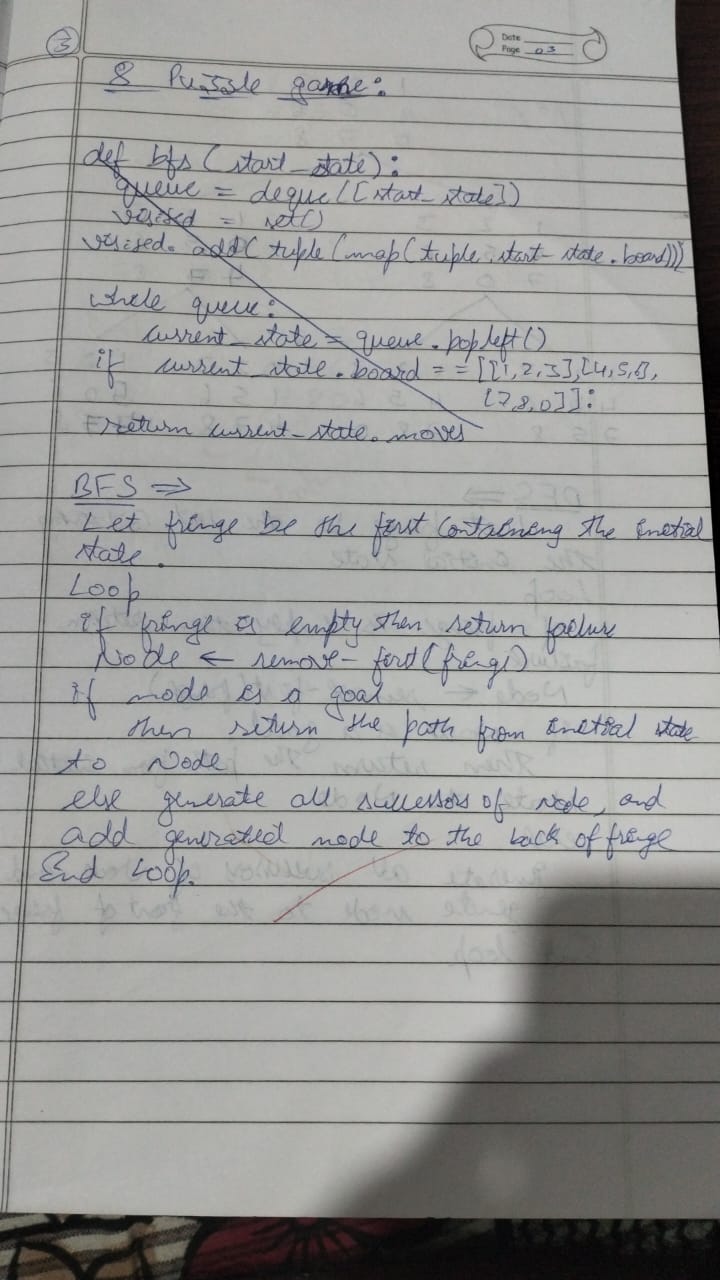
**Output Snapshot**

****

**State Space Tree**



**Program 2 - 8 Puzzle Using BFS Algorithm**

****

**Code**

import sys

import numpy as np class Node:

def init (self, state, parent, action): self.state = state

self.parent = parent self.action = action

class StackFrontier:

def init (self):

self.frontier = [] def add(self, node):

self.frontier.append(node) def contains\_state(self, state):

return any((node.state[0] == state[0]).all() for node in self.frontier) def empty(self):

return len(self.frontier) == 0 def remove(self):

if self.empty():

raise Exception("Empty Frontier")

else:

node = self.frontier[-1] self.frontier = self.frontier[:-1] return node

class QueueFrontier(StackFrontier): def remove(self):

if self.empty():

raise Exception("Empty Frontier")

class Puzzle:

else:

node = self.frontier[0] self.frontier = self.frontier[1:] return node

def init (self, start, startIndex, goal, goalIndex): self.start = [start, startIndex]

self.goal = [goal, goalIndex] self.solution = None

def neighbors(self, state): mat, (row, col) = state

results = [] if row > 0:

mat1 = np.copy(mat) mat1[row][col] = mat1[row - 1][col] mat1[row - 1][col] = 0

results.append(('up', [mat1, (row - 1, col)])) if col > 0:

mat1 = np.copy(mat) mat1[row][col] = mat1[row][col - 1] mat1[row][col - 1] = 0

results.append(('left', [mat1, (row, col - 1)])) if row < 2:

mat1 = np.copy(mat)

mat1[row][col] = mat1[row + 1][col] mat1[row + 1][col] = 0

results.append(('down', [mat1, (row + 1, col)])) if col < 2:

mat1 = np.copy(mat)

mat1[row][col] = mat1[row][col + 1] mat1[row][col + 1] = 0

results.append(('right', [mat1, (row, col + 1)])) return results

def print(self):

solution = self.solution if self.solution is not None else None

print("Start State:\n", self.start[0], "\n") print("Goal State:\n", self.goal[0], "\n")

print("\nStates Explored: ", self.num\_explored, "\n") print("Solution:\n ")

for action, cell in zip(solution[0], solution[1]): print("action: ", action, "\n", cell[0], "\n")

print("Goal Reached!!")

def does\_not\_contain\_state(self, state): for st in self.explored:

if (st[0] == state[0]).all():

return False return True

def solve(self):

self.num\_explored = 0

start = Node(state=self.start, parent=None, action=None)

frontier = QueueFrontier() frontier.add(start) self.explored = []

while True:

if frontier.empty():

raise Exception("No solution") node = frontier.remove() self.num\_explored += 1

if (node.state[0] == self.goal[0]).all(): actions = []

cells = []

while node.parent is not None: actions.append(node.action) cells.append(node.state) node = node.parent

actions.reverse() cells.reverse()

self.solution = (actions, cells) return

self.explored.append(node.state)

for action, state in self.neighbors(node.state):

if not frontier.contains\_state(state) and self.does\_not\_contain\_state(state): child =

Node(state=state, parent=node, action=action) frontier.add(child)

start = np.array([[1, 2, 3], [8, 0, 4], [7, 6, 5]])

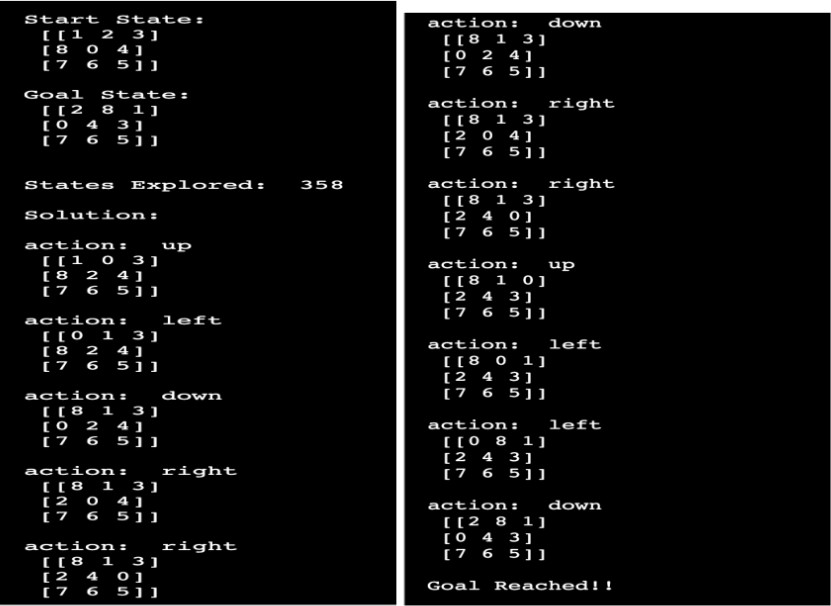
goal = np.array([[2, 8, 1], [0, 4, 3], [7, 6, 5]])

startIndex = (1, 1)

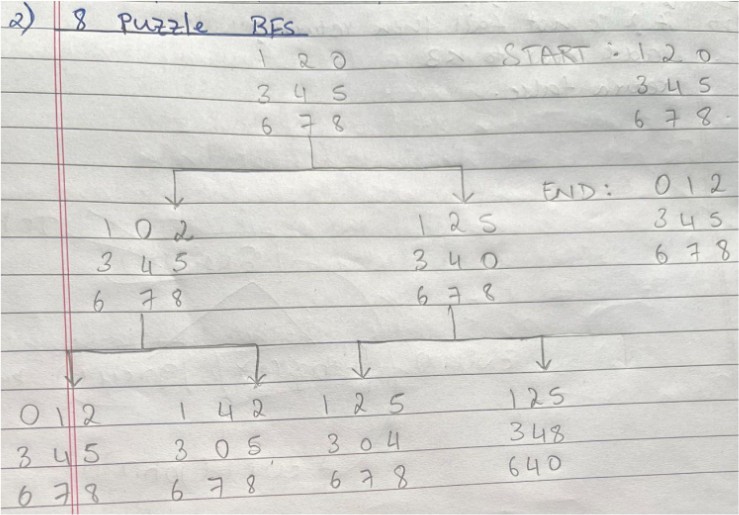
goalIndex = (1, 0)

p = Puzzle(start, startIndex, goal, goalIndex) p.solve() p.print()

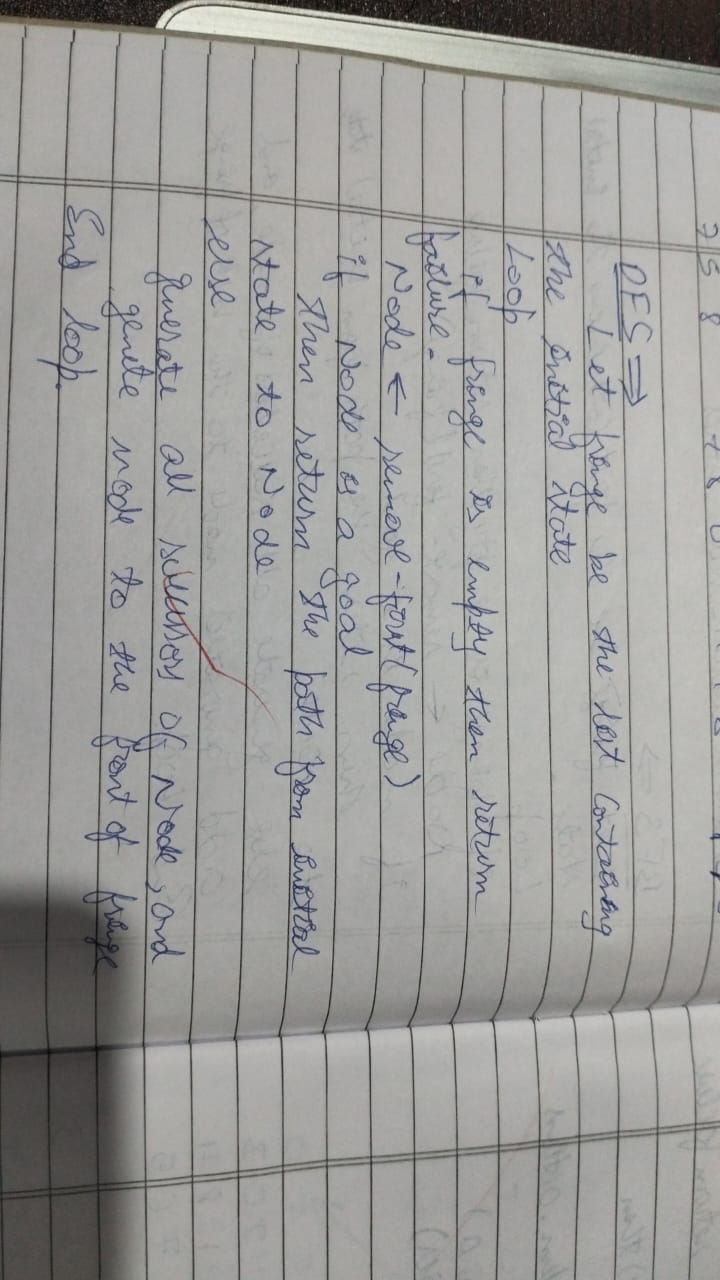
**Output Snapshot**

****

**State Space Tree**

****

**Program 3 - 8 puzzle using DFS Algorithm**

****

**Code**

import copy

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

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

def move(temp, movement): if movement=="up":

for i in range(3):

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

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

return temp

if movement=="down": for i in range(3):

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

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

return temp

if movement=="left": for i in range(3):

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

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

return temp

if movement=="right": for i in range(3):

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

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

return temp def ids():

global inp

global out global flag

for limit in range(100): print('LIMIT -> '+str(limit)) stack=[]

inpx=[inp,"none"] stack.append(inpx) level=0 while(True):

if len(stack)==0:

break puzzle=stack.pop(0) if level<=limit:

print(str(puzzle[1])+" --> "+str(puzzle[0])) if(puzzle[0]==out):

print("Found")

print('Path cost='+str(level)) flag=True

return else:

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

if(up!=puzzle[0]):

upx=[up,"up"] stack.insert(0, upx)

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

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

downx=[down,"down"] stack.insert(0, downx)

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

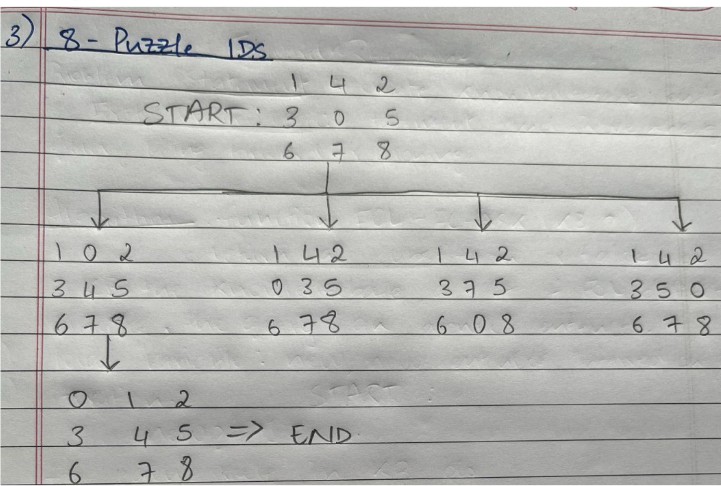
right=move(temp, "right") if(right!=puzzle[0]): rightx=[right,"right"] stack.insert(0, rightx)

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

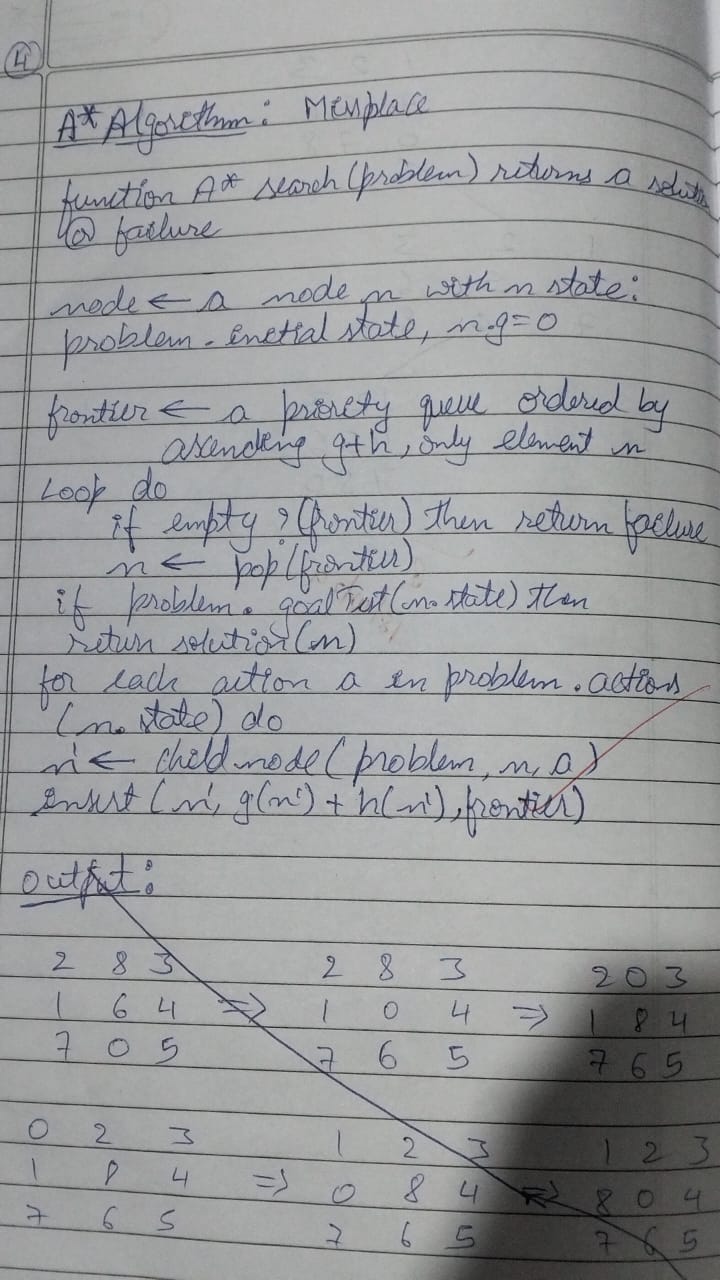
**Output Snapshot**

****

**State Space Tree**

****

**Program 04 - 8 Puzzle Using A\* Algorithm**

****

**Code**

def print\_b(src): state = src.copy()

state[state.index(-1)] = ' ' print(

f"""

{state[0]} {state[1]} {state[2]}

{state[3]} {state[4]} {state[5]}

{state[6]} {state[7]} {state[8]} “””

)

def h(state, target): count = 0

i = 0

for j in state:

if state[i] != target[i]: count = count+1

return count

def astar(state, target): states = [src]

g = 0 visited\_states = [] while len(states):

print(f"Level: {g}") moves = []

for state in states: visited\_states.append(state) print\_b(state)

if state == target: print("Success") return

moves += [move for move in possible\_moves( state, visited\_states) if move not in moves]

costs = [g + h(move, target) for move in moves] states = [moves[i]

for i in range(len(moves)) if costs[i] == min(costs)] g += 1

print("Fail")

def possible\_moves(state, visited\_state):

b = state.index(-1) d = []

if b - 3 in range(9): d.append('u')

if b not in [0, 3, 6]:

d.append('l')

if b not in [2, 5, 8]:

d.append('r')

if b + 3 in range(9): d.append('d')

pos\_moves = [] for m in d:

pos\_moves.append(gen(state, m, b))

return [move for move in pos\_moves if move not in visited\_state] def gen(state, m, b):

temp = state.copy() if m == 'u':

temp[b - 3], temp[b] = temp[b], temp[b - 3] if m == 'l':

temp[b - 1], temp[b] = temp[b], temp[b - 1] if m == 'r':

temp[b + 1], temp[b] = temp[b], temp[b + 1] if m == 'd':

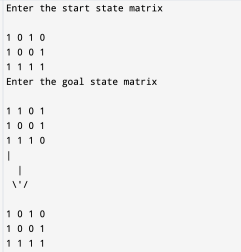
temp[b + 3], temp[b] = temp[b], temp[b + 3] return temp

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

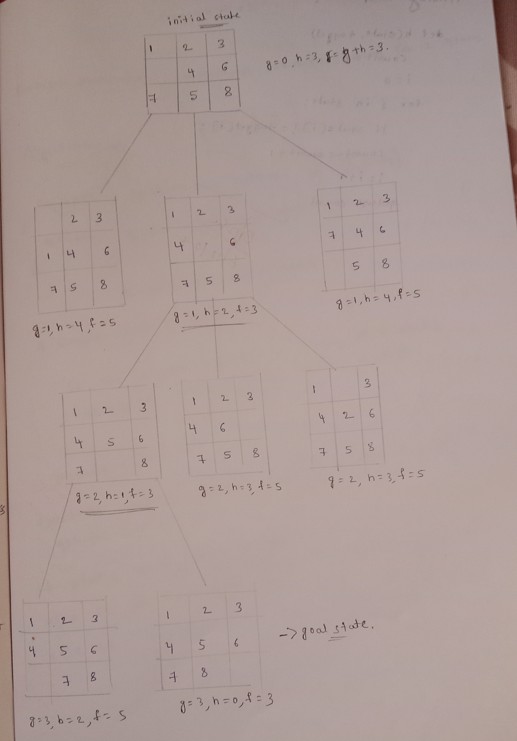
target = [1, 2, 3, 4, 5,6, 7, 8,-1]

astar(src, target)

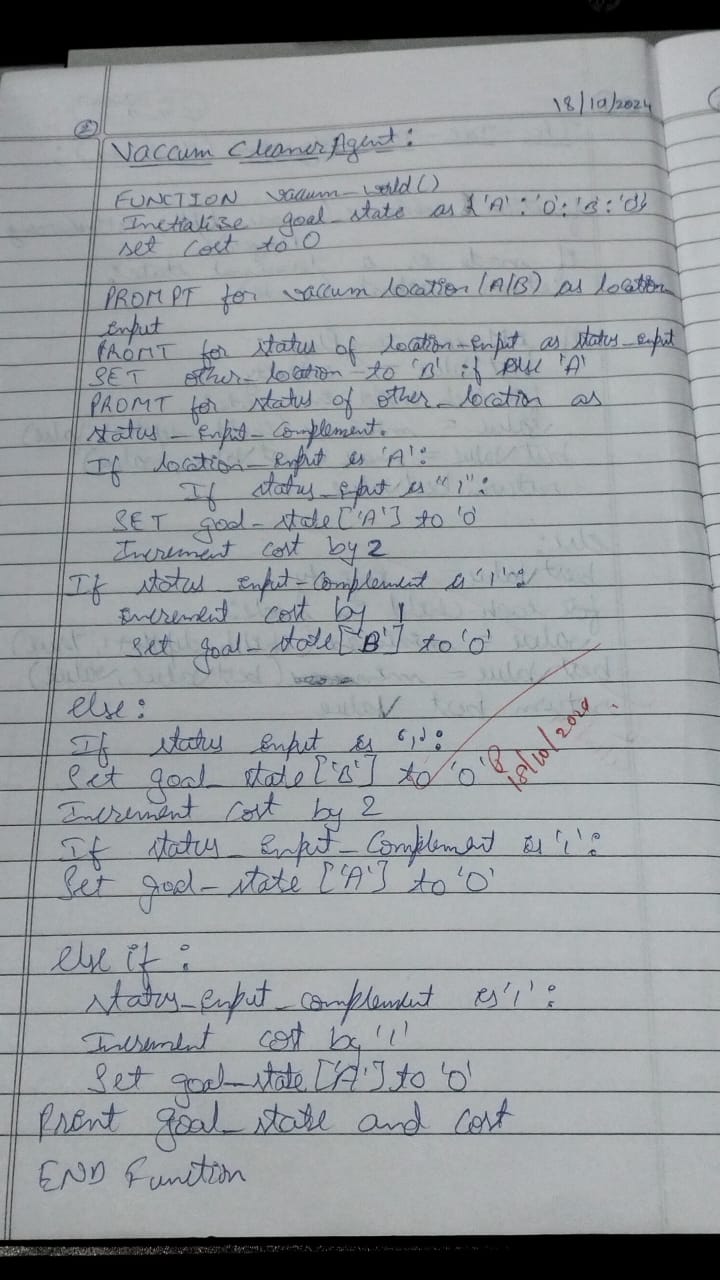
**Output Snapshot**

****

**State Space Tree**

****

**Program 5 - Vacuum Cleaner Algorithm**

****

**Code**

def vacuum\_world():

goal\_state = {'A': '0', 'B': '0'}

cost = 0

location\_input = input("Enter Location of Vacuum: ") status\_input = input("Enter status of " + location\_input+ " : ") status\_input\_complement = input("Enter status of other room : ")

print("Initial Location Condition {A : " + str(status\_input\_complement) + ", B : " + str(status\_input) + " }"

if location\_input == 'A':

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

print("Location A is Dirty.") 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': print("Location B is Dirty.") print("Moving right to the Location B. ") cost += 1

print("COST for moving RIGHT " + str(cost)) goal\_state['B'] = '0'

cost += 1

print("COST for SUCK " + str(cost)) print("Location B has been Cleaned. ")

else:

print("No action" + str(cost)) print("Location B is already clean.")

if status\_input == '0':

print("Location A is already clean ") if status\_input\_complement == '1':

print("Location B is Dirty.")

print("Moving RIGHT to the Location B. ") cost += 1

print("COST for moving RIGHT " + str(cost)) goal\_state['B'] = '0'

cost += 1

print("Cost for SUCK" + str(cost)) print("Location B has been Cleaned. ")

else:

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

print("Location B is already clean.")

else:

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

print("Location B is Dirty.") goal\_state['B'] = '0'

cost += 1

print("COST for CLEANING " + str(cost))

print("Location B has been Cleaned.") if status\_input\_complement == '1':

print("Location A is Dirty.") print("Moving LEFT to the Location A. ") cost += 1

print("COST for moving LEFT " + str(cost)) goal\_state['A'] = '0'

cost += 1

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

else:

print(cost)

print("Location B is already clean.")

if status\_input\_complement == '1': print("Location A is Dirty.") print("Moving LEFT to the Location A. ") cost += 1

print("COST for moving LEFT " + str(cost)) goal\_state['A'] = '0'

cost += 1

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

else:

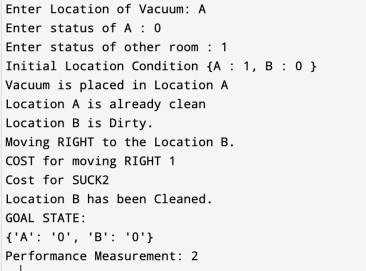
print("No action " + str(cost)) print("Location A is already clean.")

print("GOAL STATE: ")

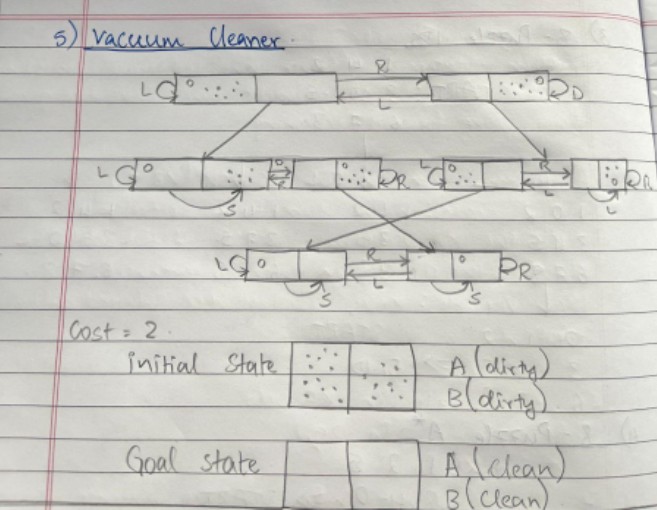
print(goal\_state)

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

**Output Snapshot**

****

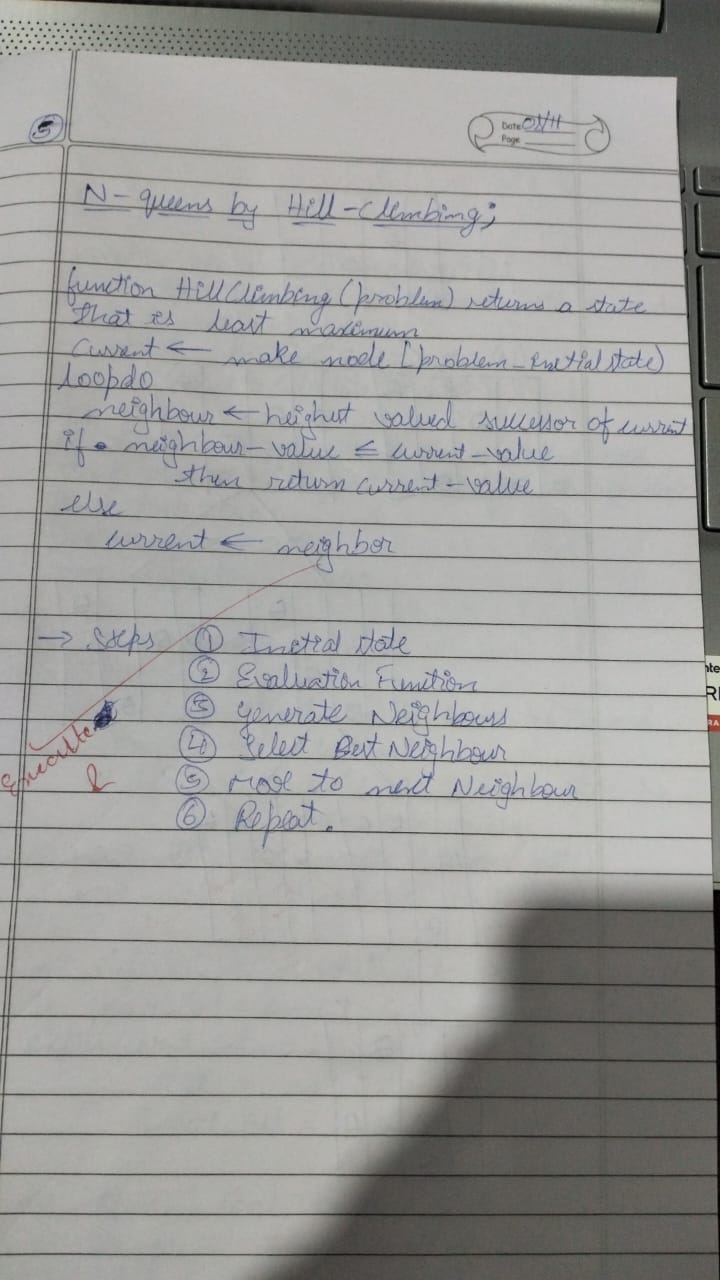
**State Space Tree**

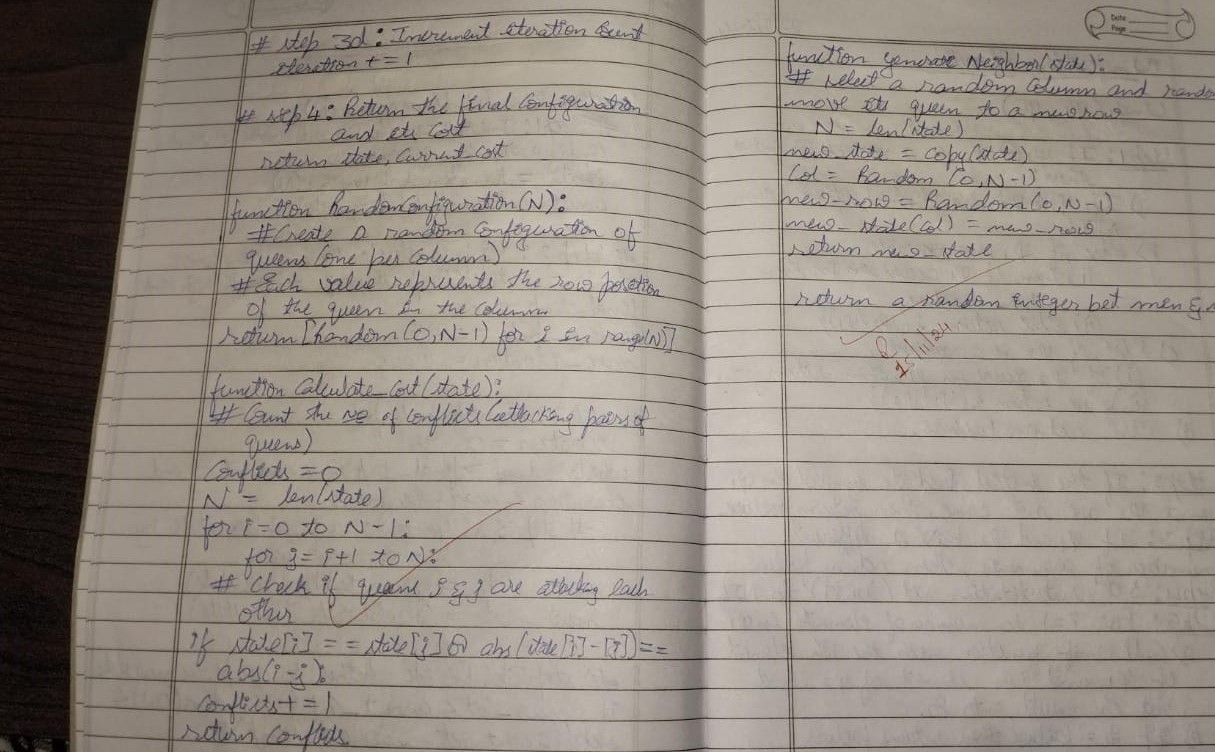
****

**Program 6:**

**Hill climbing N Queens**

**Algorithm:**

****



Code:

import random

def print\_board(board):

"""Prints the chess board."""

for row in board:

print(" ".join("Q" if col == 1 else "." for col in row))

print()

def generate\_board(n):

"""Generates a random N-Queens board."""

return [random.randint(0, n-1) for \_ in range(n)]

def board\_to\_matrix(queens):

"""Converts a board representation to a matrix for visualization."""

n = len(queens)

matrix = [[0] \* n for \_ in range(n)]

for col, row in enumerate(queens):

matrix[row][col] = 1

return matrix

def calculate\_conflicts(queens):

"""Calculates the number of conflicts in the current board."""

n = len(queens)

conflicts = 0

for i in range(n):

for j in range(i + 1, n):

if queens[i] == queens[j] or abs(queens[i] - queens[j]) == abs(i - j):

conflicts += 1

return conflicts

def find\_best\_move(queens):

"""Finds the best move for a queen to reduce conflicts."""

n = len(queens)

current\_conflicts = calculate\_conflicts(queens)

best\_queens = queens[:]

for col in range(n):

original\_row = queens[col]

for row in range(n):

if row == original\_row:

continue

queens[col] = row

new\_conflicts = calculate\_conflicts(queens)

if new\_conflicts < current\_conflicts:

current\_conflicts = new\_conflicts

best\_queens = queens[:]

queens[col] = original\_row

return best\_queens, current\_conflicts

def hill\_climbing(n):

"""Hill climbing algorithm to solve N-Queens."""

queens = generate\_board(n)

steps = 0

while True:

current\_conflicts = calculate\_conflicts(queens)

if current\_conflicts == 0:

return queens, steps

next\_queens, next\_conflicts = find\_best\_move(queens)

if next\_conflicts >= current\_conflicts:

return None, steps

queens = next\_queens

steps += 1

if \_\_name\_\_ == "\_\_main\_\_":

n = int(input("Enter the number of queens (N): "))

solution, steps = hill\_climbing(n)

if solution:

print(f"Solution found in {steps} steps:")

print\_board(board\_to\_matrix(solution))

else:

print("No solution **found.")**

**Output:**

**Solution found in X steps:**

**. Q . . . . . .**

**. . . Q . . . .**

**Q . . . . . . .**

**. . . . . Q . .**

**. . Q . . . . .**

**. . . . . . . Q**

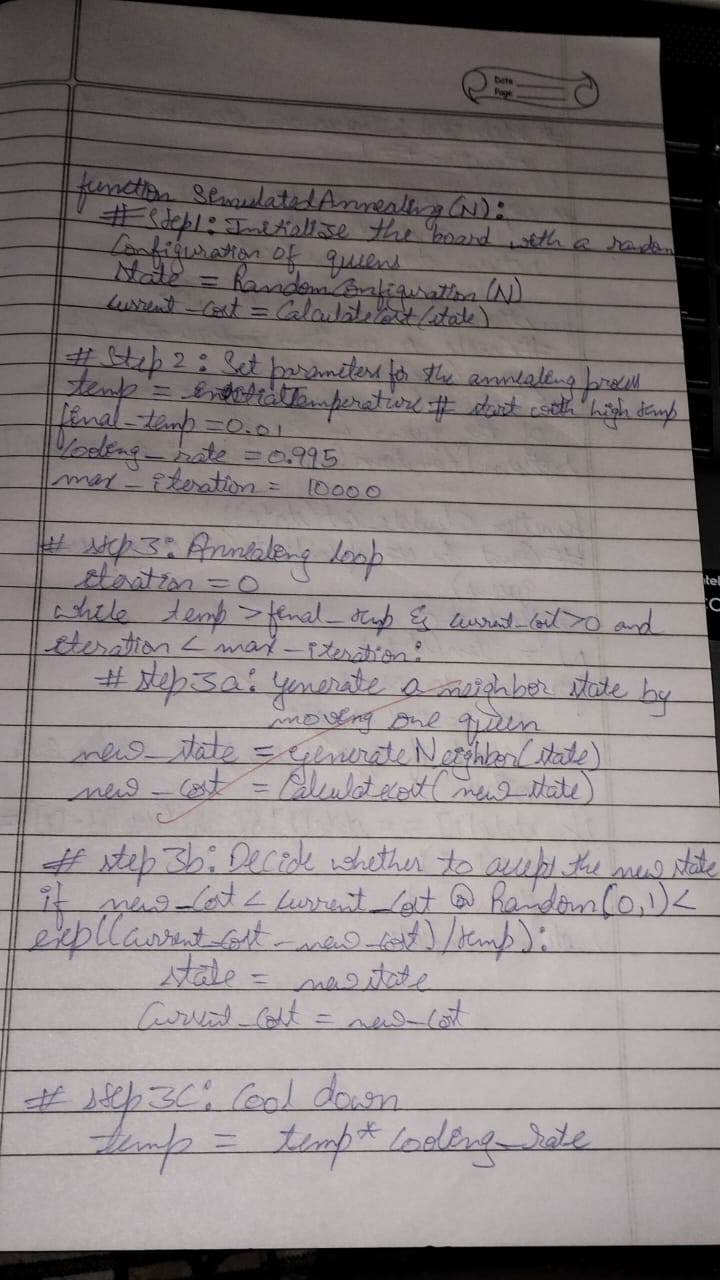
**. . . . Q . . .**

**. . . . . . Q .**

**Program 7**

**Simulated Annealing problem :**

**Algorithm:**

****

**Code:**

import mlrose

import numpy as np

*# Define the fitness function for the N-Queens problem*

def queens\_max(position):

no\_attack\_on = 0

queen\_not\_attacking = 0

n = len(position)

for i in range(n):

for j in range(i + 1, n):

*# Check if queens are not in the same row or diagonal*

if position[i] != position[j] and abs(position[i] - position[j]) != abs(i - j):

no\_attack\_on += 1

queen\_not\_attacking = no\_attack\_on

return queen\_not\_attacking

*# Define the problem*

objective = mlrose.CustomFitness(queens\_max)

problem = mlrose.DiscreteOpt(length=8, fitness\_fn=objective, maximize=True, max\_val=8)

*# Define initial state and simulated annealing parameters*

initial\_position = np.array([4, 0, 1, 5, 2, 6, 3, 7])

schedule = mlrose.ExpDecay()

max\_attempts = 500

max\_iters = 5000

*# Run simulated annealing*

best\_position, best\_objective = mlrose.simulated\_annealing(

problem=problem,

schedule=schedule,

max\_attempts=max\_attempts,

max\_iters=max\_iters,

init\_state=initial\_position

)

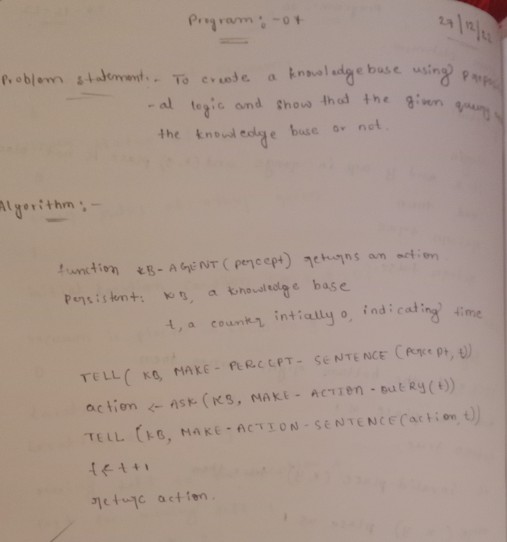
*# Print the results*

print("The best position found is:", best\_position)

print("The number of queens that are not attacking each other is:", best\_objective)

**output:** Best position found: [2 4 7 3 0 6 1 5]

Number of attacks: 0

**Program-07 KnowledgeBase Algorithm**

**Code**

combinations=[(True,True, True),(True,True,False),(True,False,True),(True,False, False),(False,True, True),(False,True, False),(False, False,True),(False,False, False)] variable={'p':0,'q':1, 'r':2}

kb=''

q=''

priority={'~':3,'v':1,'^':2} def input\_rules():

global kb, q

kb = (input("Enter rule: "))

q = input("Enter the Query: ") def entailment():

global kb, q

print(''\*10+"Truth Table Reference"+''\*10) print('kb','alpha')

print('\*'\*10)

for comb in combinations:

s = evaluatePostfix(toPostfix(kb), comb) f = evaluatePostfix(toPostfix(q), comb) print(s, f)

print('-'\*10) if s and not f:

return False return True

def isOperand(c):

return c.isalpha() and c!='v'

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

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

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

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

def hasLessOrEqualPriority(c1, c2): try:

return priority[c1]<=priority[c2] except KeyError:

return False def toPostfix(infix):

stack = [] postfix = '' for c in infix:

if isOperand(c): postfix += c

else:

if isLeftParanthesis(c): stack.append(c)

elif isRightParanthesis(c): operator = stack.pop()

while not isLeftParanthesis(operator): postfix += operator

operator = stack.pop()

else:

while (not isEmpty(stack)) and hasLessOrEqualPriority(c, peek(stack)): postfix += stack.pop()

stack.append(c) while (not isEmpty(stack)):

postfix += stack.pop()

return postfix

def evaluatePostfix(exp, comb): stack = []

for i in exp:

if isOperand(i): stack.append(comb[variable[i]])

elif i == '~':

val1 = stack.pop() stack.append(not val1)

else:

val1 = stack.pop() val2 = stack.pop()

stack.append(\_eval(i,val2,val1)) return stack.pop()

def \_eval(i, val1, val2): if i == '^':

return val2 and val1 return val2 or val1

#Test 1 input\_rules()

ans = entailment() if ans:

print("Knowledge Base entails query") else:

print("Knowledge Base does not entail query") #Test 2

input\_rules()

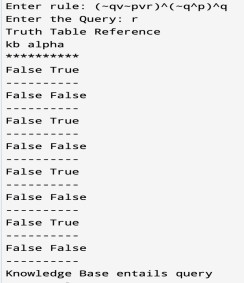
ans = entailment()

if ans:

print("Knowledge Base entails query") else:

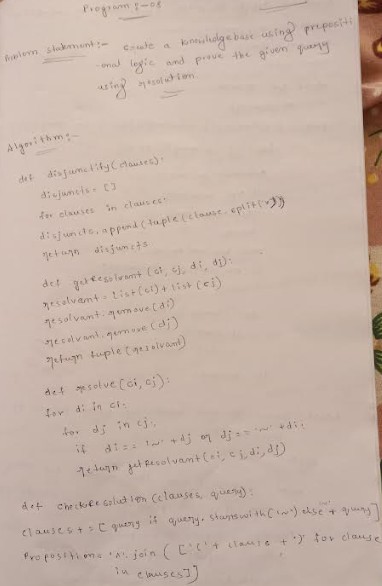
print("Knowledge Base does not entail query")

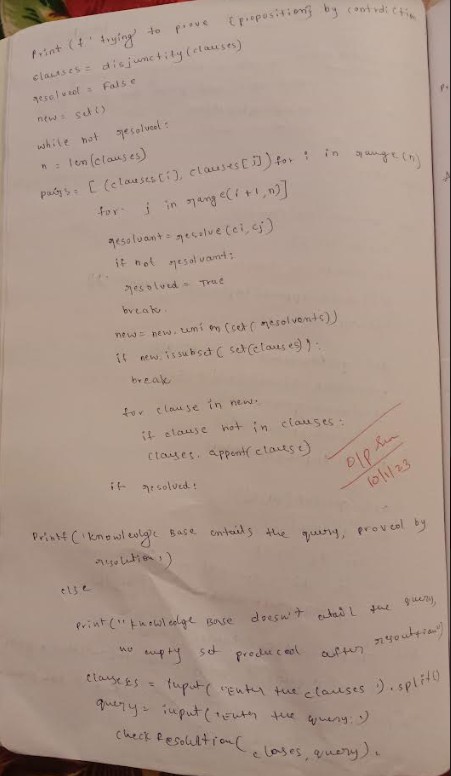
**OUTPUT**

****

**Program-09 KnowledgeBase - Resolution**

**Algorithm**

****



**Code**

def disjunctify(clauses): disjuncts = []

for clause in clauses: disjuncts.append(tuple(clause.split('v')))

return disjuncts

def getResolvant(ci, cj, di, dj): resolvant = list(ci) + list(cj) resolvant.remove(di) resolvant.remove(dj)

return tuple(resolvant)

def resolve(ci, cj): for di in ci:

for dj in cj:

if di == '~' + dj or dj == '~' + di: return getResolvant(ci, cj, di, dj)

def checkResolution(clauses, query):

clauses += [query if query.startswith('~') else '~' + query] proposition = '^'.join(['(' + clause + ')' for clause in clauses]) print(f'Trying to prove {proposition} by contradiction. ')

clauses = disjunctify(clauses) resolved = False

new = set()

while not resolved: n = len(clauses)

pairs = [(clauses[i], clauses[j]) for i in range(n) for j in range(i + 1, n)] for (ci, cj) in pairs:

resolvant = resolve(ci, cj) if not resolvant:

resolved = True break

new = new.union(set(resolvents)) if new.issubset(set(clauses)):

break

for clause in new:

if clause not in clauses: clauses.append(clause)

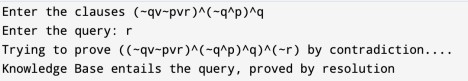
if resolved:

print('Knowledge Base entails the query, proved by resolution')

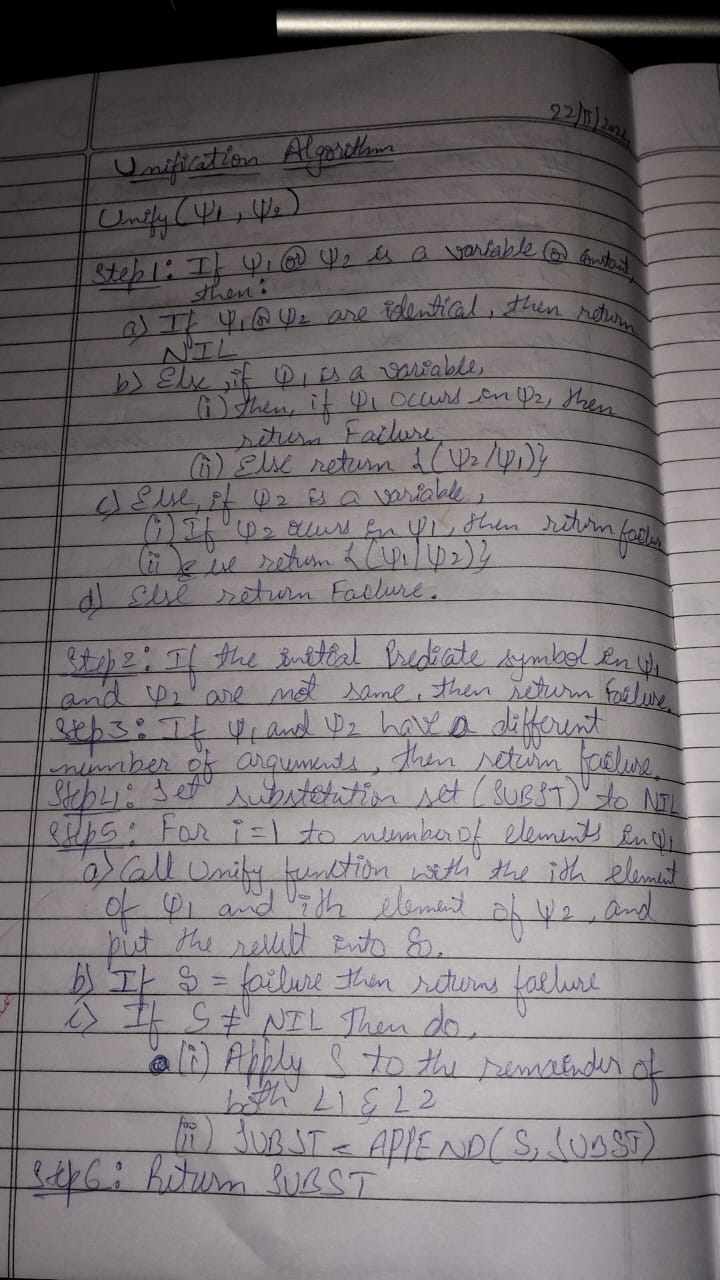
else:

print("Knowledge Base doesn't entail the query, no empty set produced after resolution") clauses

= input('Enter the clauses ').split() query = input('Enter the query: ') checkResolution(clauses, query) **Output Snapshot**

****

**Program-10 Unification in first order logic Algorithm**

****

**Code**

import re

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

return attributes

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

def isConstant(char):

return char.isupper() and len(char) == 1

def isVariable(char):

return char.islower() and len(char) == 1 def replaceAttributes(exp, old, new):

attributes = getAttributes(exp) predicate = getInitialPredicate(exp) for index, val in enumerate(attributes):

if val == old: attributes[index] = new

return predicate + "(" + ",".join(attributes) + ")"

def apply(exp, substitutions):

for substitution in substitutions: new, old = substitution

exp = replaceAttributes(exp, old, new) return exp

def checkOccurs(var, exp): if exp.find(var) == -1:

return False return True

def getFirstPart(expression):

attributes = getAttributes(expression) return attributes[0]

def getRemainingPart(expression):

predicate = getInitialPredicate(expression) attributes = getAttributes(expression)

newExpression = predicate + "(" + ",".join(attributes[1:]) + ")" return newExpression

def unify(exp1, exp2): if exp1 == exp2:

return []

if isConstant(exp1) and isConstant(exp2): if exp1 != exp2:

print(f"{exp1} and {exp2} are constants. Cannot be unified") return []

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

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

if isVariable(exp1):

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

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

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

attributeCount1 = len(getAttributes(exp1)) attributeCount2 = len(getAttributes(exp2)) if attributeCount1 != attributeCount2:

print(f"Length of attributes {attributeCount1} and {attributeCount2} do not match. Cannot be unified")

return []

head1 = getFirstPart(exp1)

head2 = getFirstPart(exp2) initialSubstitution = unify(head1, head2) if not initialSubstitution:

return []

if attributeCount1 == 1: return initialSubstitution

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

if initialSubstitution != []:

tail1 = apply(tail1, initialSubstitution) tail2 = apply(tail2, initialSubstitution)

remainingSubstitution = unify(tail1, tail2) if not remainingSubstitution:

return []

return initialSubstitution + remainingSubstitution if name == " main ":

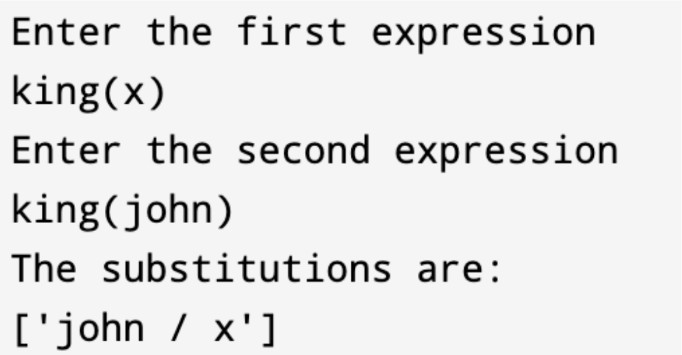
print("Enter the first expression") e1 = input()

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

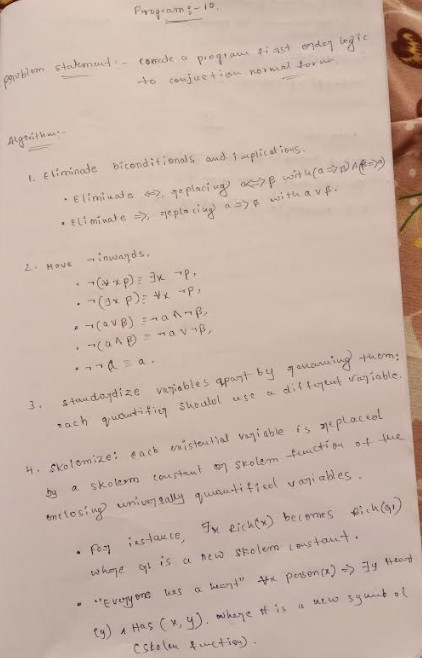
substitutions = unify(e1, e2) print("The substitutions are:")

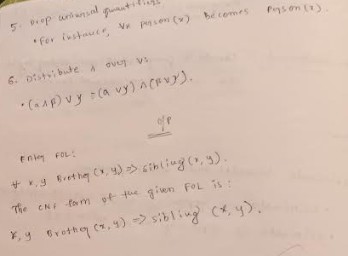
print([' / '.join(substitution) for substitution in substitutions])

**Output Snapshot**



**Program-11 First Order Logic to Conjunctive Normal Form Algorithm**

****



**Code**

import re

def getAttributes(string): expr = '\([^)]+\)'

matches = re.findall(expr, string)

return [m for m in str(matches) if m.isalpha()]

def getPredicates(string):

expr = '[a-z~]+\([A-Za-z,]+\)' return re.findall(expr, string)

def DeMorgan(sentence):

string = ''.join(list(sentence).copy()) string = string.replace('~~','')

flag = '[' in string

string = string.replace('~[','') string = string.strip(']')

for predicate in getPredicates(string):

string = string.replace(predicate, f'~{predicate}') s = list(string)

for i, c in enumerate(string): if c == 'V':

s[i] = '^' elif c == '^':

s[i] = 'V'

string = ''.join(s)

string = string.replace('~~','')

return f'[{string}]' if flag else string

def Skolemization(sentence):

SKOLEM\_CONSTANTS = [f'{chr(c)}' for c in range(ord('A'), ord('Z')+1)] statement = ''.join(list(sentence).copy())

matches = re.findall('[∀∃].', statement) for match in matches[::-1]:

statement = statement.replace(match, '') statements = re.findall('\[\[[^]]+\]]', statement) for s in statements:

statement = statement.replace(s, s[1:-1]) for predicate in getPredicates(statement):

attributes = getAttributes(predicate) if ''.join(attributes).islower():

statement = statement.replace(match[1],SKOLEM\_CONSTANTS.pop(0)) else:

aL = [a for a in attributes if a.islower()]

aU = [a for a in attributes if not a.islower()][0]

statement = statement.replace(aU, f'{SKOLEM\_CONSTANTS.pop(0)}({aL[0] if len(aL) else match[1]})')

return statement def fol\_to\_cnf(fol):

statement = fol.replace("<=>", "\_") while '\_' in statement:

i = statement.index('\_')

new\_statement = '[' + statement[:i] + '=>' + statement[i+1:] + ']^['+ statement[i+1:] + '=>' + statement[:i] + ']'

statement = new\_statement

statement = statement.replace("=>", "-") expr = '\[([^]]+)\]'

statements = re.findall(expr, statement)

for i, s in enumerate(statements): if '[' in s and ']' not in s:

statements[i] += ']' for s in statements:

statement = statement.replace(s, fol\_to\_cnf(s)) while '-' in statement:

i = statement.index('-')

br = statement.index('[') if '[' in statement else 0 new\_statement = '~' + statement[br:i] + 'V' + statement[i+1:]

statement = statement[:br] + new\_statement if br > 0 else new\_statement while '~∀' in statement:

i = statement.index('~∀')

statement = list(statement)

statement[i], statement[i+1], statement[i+2] = '∃', statement[i+2], '~' statement = ''.join(statement)

while '~∃' in statement:

i = statement.index('~∃') s = list(statement)

s[i], s[i+1], s[i+2] = '∀', s[i+2], '~' statement = ''.join(s)

statement = statement.replace('~[∀','[~∀') statement = statement.replace('~[∃','[~∃') expr = '(~[∀V∃].)'

statements = re.findall(expr, statement) for s in statements:

statement = statement.replace(s, fol\_to\_cnf(s)) expr = '~\[[^]]+\]'

statements = re.findall(expr, statement) for s in statements:

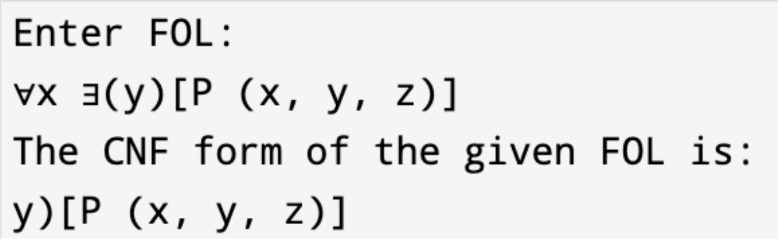
statement = statement.replace(s, DeMorgan(s)) return statement

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

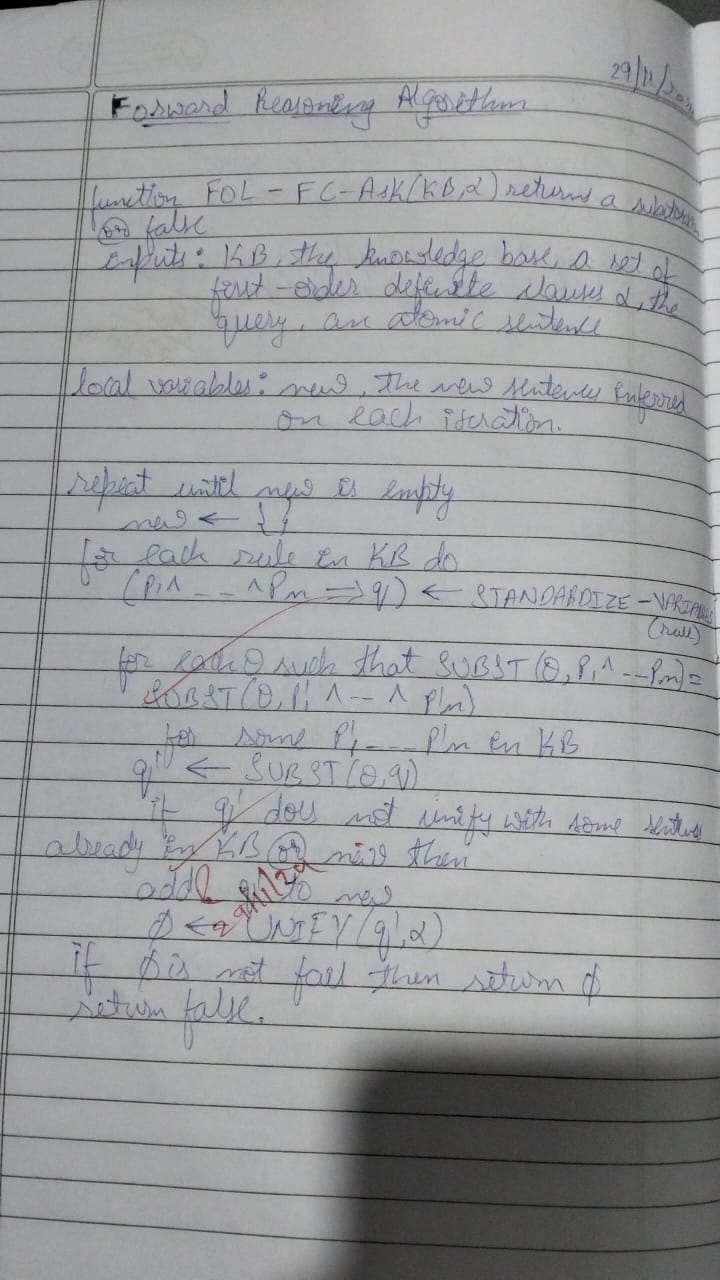
print("The CNF form of the given FOL is: ") print(Skolemization(fol\_to\_cnf(fol)))

main()

**Output Snapshot**

****

**Program-12 Forward Reasoning Algorithm**

****

**Code**

import re

def isVariable(x):

return len(x) == 1 and x.islower() and x.isalpha()

def getAttributes(string): expr = '\([^)]+\)'

matches = re.findall(expr, string) return matches

def getPredicates(string): expr = '([a-z~]+)\([^&|]+\)'

return re.findall(expr, string) class Fact:

def init (self, expression): self.expression = expression

predicate, params = self.splitExpression(expression) self.predicate = predicate

self.params = params

self.result = any(self.getConstants())

def splitExpression(self, expression): predicate = getPredicates(expression)[0]

params = getAttributes(expression)[0].strip('()').split(',')

return [predicate, params]

def getResult(self): return self.result

def getConstants(self):

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

def getVariables(self):

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

def substitute(self, constants): c = constants.copy()

f = f"{self.predicate}({','.join([constants.pop(0) if isVariable(p) else p for p in self.params])})"

return Fact(f) class Implication:

def init (self, expression): self.expression = expression l = expression.split('=>')

self.lhs = [Fact(f) for f in l[0].split('&')] self.rhs = Fact(l[1])

def evaluate(self, facts): constants = {} new\_lhs = []

for fact in facts:

for val in self.lhs:

if val.predicate == fact.predicate:

for i, v in enumerate(val.getVariables()): if v:

constants[v] = fact.getConstants()[i] new\_lhs.append(fact)

predicate, attributes = getPredicates(self.rhs.expression)[0], str(getAttributes(self.rhs.expression)[0])

for key in constants: if constants[key]:

attributes = attributes.replace(key, constants[key])

expr = f'{predicate}{attributes}'

return Fact(expr) if len(new\_lhs) and all([f.getResult() for f in new\_lhs]) else None class KB:

def init (self): self.facts = set() self.implications = set()

def tell(self, e): if '=>' in e:

self.implications.add(Implication(e)) else:

self.facts.add(Fact(e)) for i in self.implications:

res = i.evaluate(self.facts) if res:

self.facts.add(res)

def ask(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 the number of FOL expressions present in KB:") n = int(input())

print("Enter the expressions:") for i in range(n):

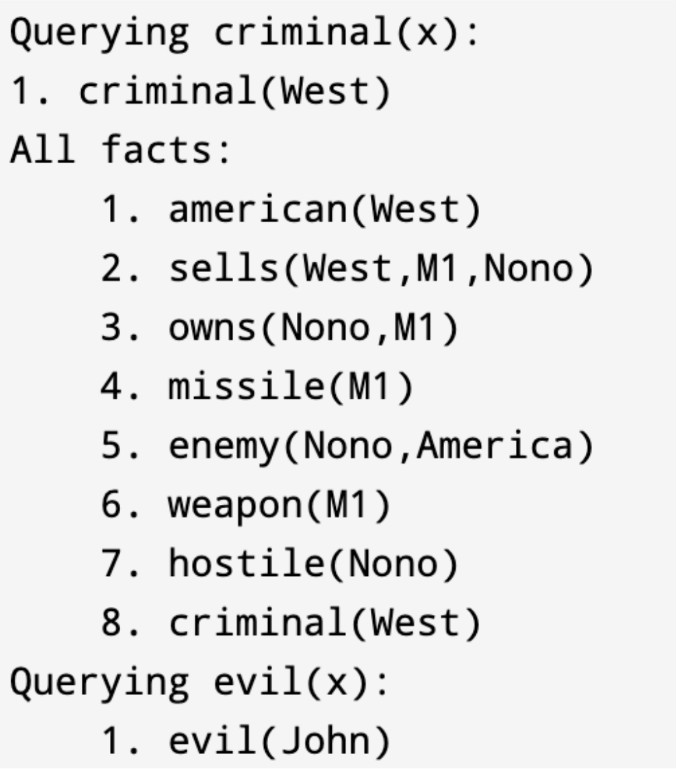
fact = input()

kb.tell(fact)

print("Enter the query:") query = input() kb.ask(query) kb.display()

main()

**Output Snapshot**

****