VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"Jnana Sangama", Belgaum -590014, Karnataka.

LAB RECORD

on

Artificial Intelligence (23CS5PCAIN)

Submitted by

VANITHA (1BM22CS317)

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
Sep-2024 to Jan-2025

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 " Bio Inspired Systems (23CS5BSBIS)" carried out by **VANITHA (1BM22CS317),** 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. The Lab report has been approved as it satisfies the academic requirements of the above mentioned subject and the work prescribed for the said degree.

Prameetha Pai Assistant Professor Department of CSE, BMSCE	Dr. Kavitha Sooda Professor & HOD Department of CSE, BMSCE

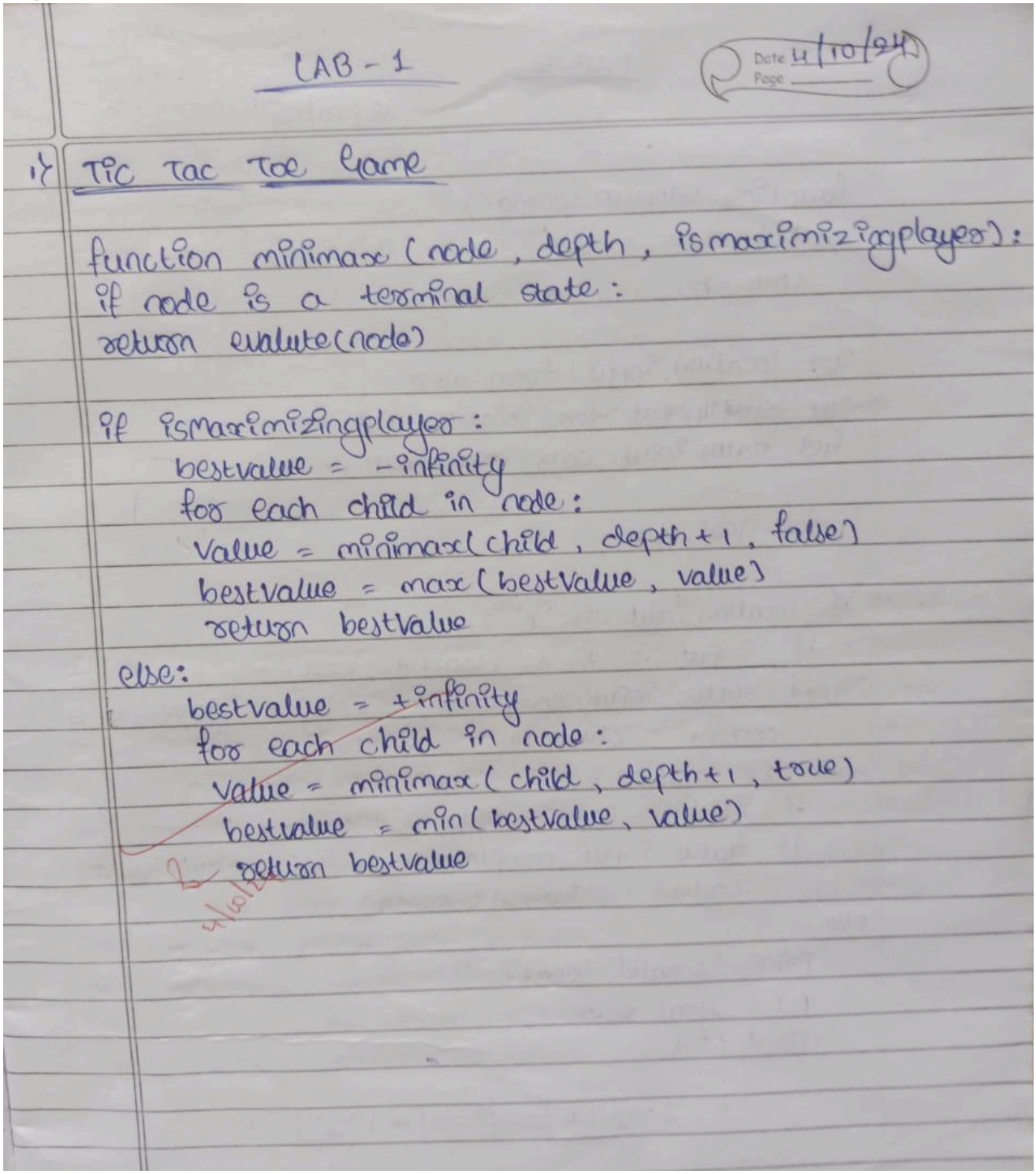
Index

Github Link:
https://github.com/vanithakss/AI_LAB

Program 1

Implement Tic –Tac –Toe Game

Algorithm :



LAB - 1

Date 4/10/24
Page

1) Tic Tac Toe Game

```
function minimax(node, depth, isMaximizingplayer):
if node is a terminal state:
return evalute(node)

if isMaximizingplayer:
    bestvalue = -infinity
    for each child in node:
     value = minimax(child, depth+1, false)
     bestvalue = max(bestvalue, value)
     return bestvalue
else:
    bestvalue = +infinity
    for each child in node:
     value = minimax(child, depth+1, true)
     bestvalue = min(bestvalue, value)
     return bestvalue
```

Code:

```
#Tic Tac Toe Game
board = {
    1: ' ', 2: ' ', 3: ' ',
    4: ' ', 5: ' ', 6: ' ',
    7: ' ', 8: ' ', 9: ' '
}

def printBoard(board):
    print(board[1] + '|' + board[2] + '|' + board[3])
    print('-+-+-')
    print(board[4] + '|' + board[5] + '|' + board[6])
    print('-+-+-')
    print(board[7] + '|' + board[8] + '|' + board[9])
    print('\n')

def spaceFree(pos):
    return board[pos] == ' '

def checkWin():
    winning_combinations = [
        (1, 2, 3), (4, 5, 6), (7, 8, 9),  # rows
        (1, 4, 7), (2, 5, 8), (3, 6, 9),  # columns
        (1, 5, 9), (3, 5, 7)              # diagonals
    ]
    for a, b, c in winning_combinations:
        if board[a] == board[b] == board[c] and board[a] != ' ':
            return True
    return False

def checkDraw():
    return all(board[key] != ' ' for key in board.keys())

def insertLetter(letter, position):
    if spaceFree(position):
        board[position] = letter
        printBoard(board)

        if checkDraw():
            print('Draw!')
            return True  # End the game
        elif checkWin():
            if letter == 'X':
                print('Bot wins!')
            else:
                print('You win!')
            return True  # End the game
    else:
        print('Position taken, please pick a different position.')
        position = int(input('Enter new position: '))
        return insertLetter(letter, position)

    return False  # Game continues

player = 'O'
bot = 'X'

def playerMove():
```

```
    position = int(input('Enter position for O: '))
    return insertLetter(player, position)

def compMove():
    bestScore = -1000
    bestMove = 0
    for key in board.keys():
        if board[key] == ' ':
            board[key] = bot
            score = minimax(board, False)
            board[key] = ' '
            if score > bestScore:
                bestScore = score
                bestMove = key

    return insertLetter(bot, bestMove)

def minimax(board, isMaximizing):
    if checkWin():
        return 1 if isMaximizing else -1
    elif checkDraw():
        return 0

    if isMaximizing:
        bestScore = -1000
        for key in board.keys():
            if board[key] == ' ':
                board[key] = bot
                score = minimax(board, False)
                board[key] = ' '
                bestScore = max(score, bestScore)
        return bestScore
    else:
        bestScore = 1000
        for key in board.keys():
            if board[key] == ' ':
                board[key] = player
                score = minimax(board, True)
                board[key] = ' '
                bestScore = min(score, bestScore)
        return bestScore

# Main game loop
game_over = False
while not game_over:
    compMove()
    game_over = playerMove()

print("Vanitha - 1BM22CS317")
```

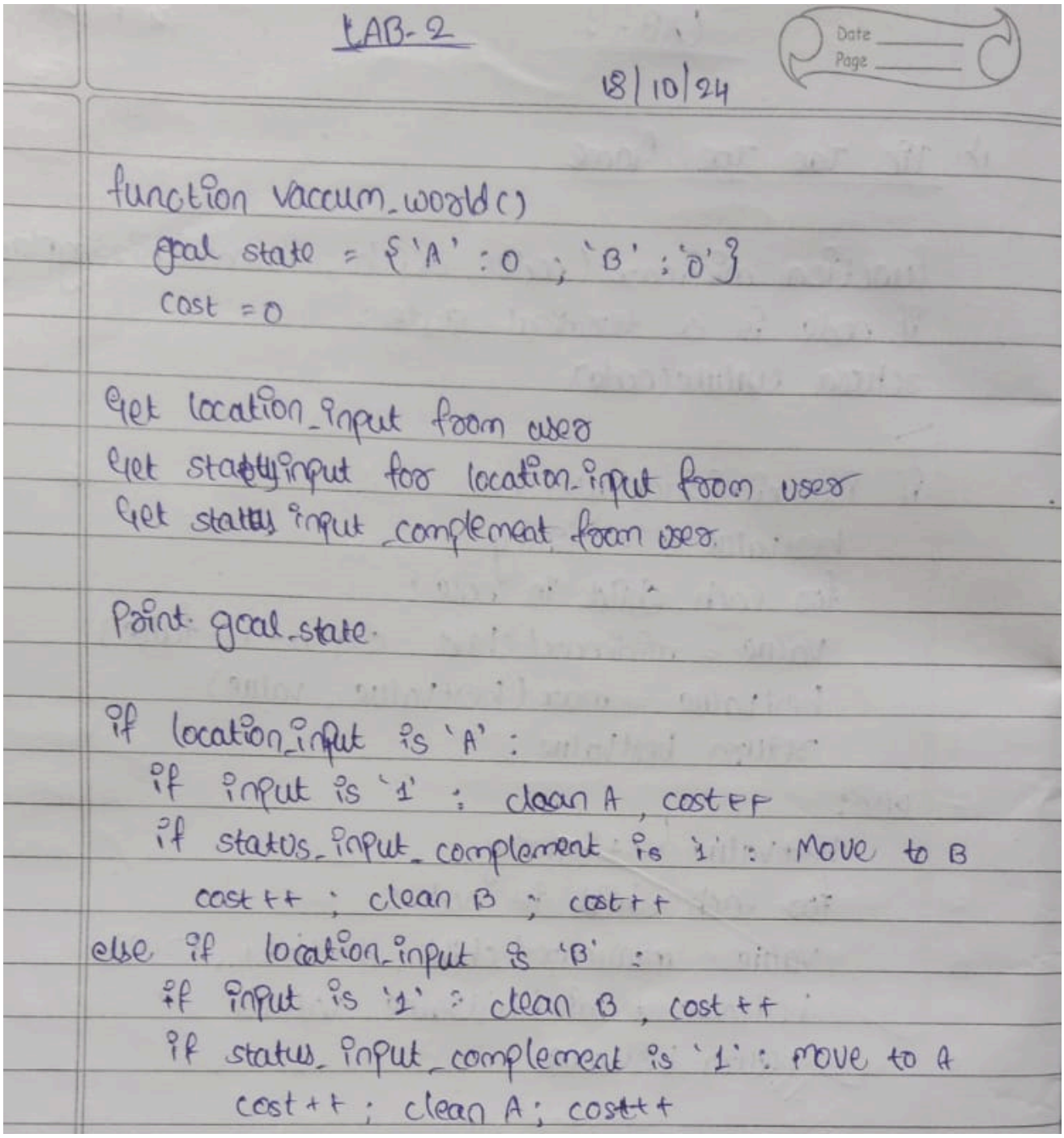
Output:

```
 | |
-+-+-
 |X|
-+-+-
 | |


Enter position for O: 9
 | |
-+-+-
 |X|
-+-+-
 | |O


X| |
-+-+-
 |X|
-+-+-
 | |O


Enter position for O: 3
X| |O
-+-+-
 |X|
-+-+-
 | |O


X|X|O
-+-+-
 |X|
-+-+-
 | |O


Enter position for O: 6
X|X|O
-+-+-
 |X|O
-+-+-
 | |O


You win!
Vanitha - 1BM22CS317
```

Program 2

Implement vacuum cleaner agent.

Algorithm



LAB-2

18/10/24

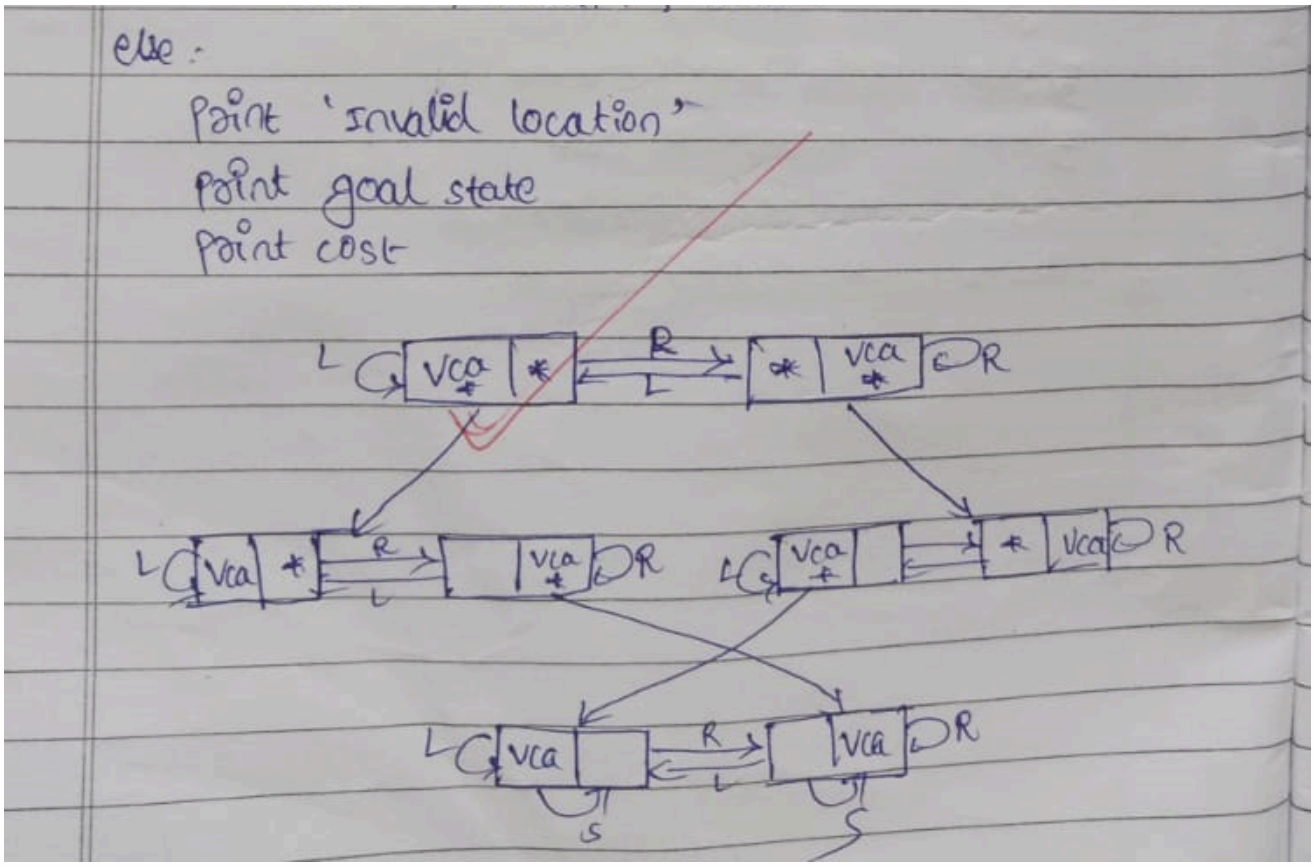
Date
Page

function vaccum_world()
goal state = {'A' : 0 , 'B' : 0}
cost = 0

Get location_input from user
Get status input for location_input from user
Get status input_complement from user.

Print goal_state.

if location_input is 'A' :
if input is '1' : clean A, cost++
if status_input_complement is '1' : Move to B
cost++ ; clean B ; cost++
else if location_input is 'B' :
if input is '1' : clean B, cost++
if status_input_complement is '1' : move to A
cost++ ; clean A; cost++



Code:

```
#Vacuum Cleaner Agent
def vacuum_world():
    # Initializing goal_state: 0 indicates Clean, 1 indicates Dirty
    goal_state = {'A': '0', 'B': '0'}
    cost = 0

    # User inputs
    location_input = input("Enter Location of Vacuum (A or B): ")
     status_input = input(f"Enter status of {location_input} (0 for Clean, 1 for
Dirty): ")
     status_input_complement = input("Enter status of other room (0 for Clean, 1
for Dirty): ")

    print("Initial Location Condition:", goal_state)

    # Function to clean a location
    def clean(location):
        nonlocal cost
        goal_state[location] = '0'
        cost += 1
        print(f"Location {location} has been Cleaned. Cost for CLEANING: {cost}")

    # Function to move to a location
    def move(location):
        nonlocal cost
```

```
        cost += 1
        print(f"Moving to Location {location}. Cost for moving: {cost}")

    if location_input == 'A':
        if status_input == '1':
            print("Location A is Dirty.")
            clean('A')
            if status_input_complement == '1':
                print("Location B is Dirty.")
                move('B')
                clean('B')
        else:
            print("Location A is already clean.")
            if status_input_complement == '1':
                print("Location B is Dirty.")
                move('B')
                clean('B')

    else:  # Vacuum is placed in location B
        if status_input == '1':
            print("Location B is Dirty.")
            clean('B')
            if status_input_complement == '1':
                print("Location A is Dirty.")
                move('A')
                clean('A')
        else:
            print("Location B is already clean.")
            if status_input_complement == '1':
                print("Location A is Dirty.")
                move('A')
                clean('A')

    # Final output
    print("GOAL STATE:", goal_state)
    print("Performance Measurement:", cost)

# Call the function to run it
vacuum_world()

print("Vanitha - 1BM22CS317")
```

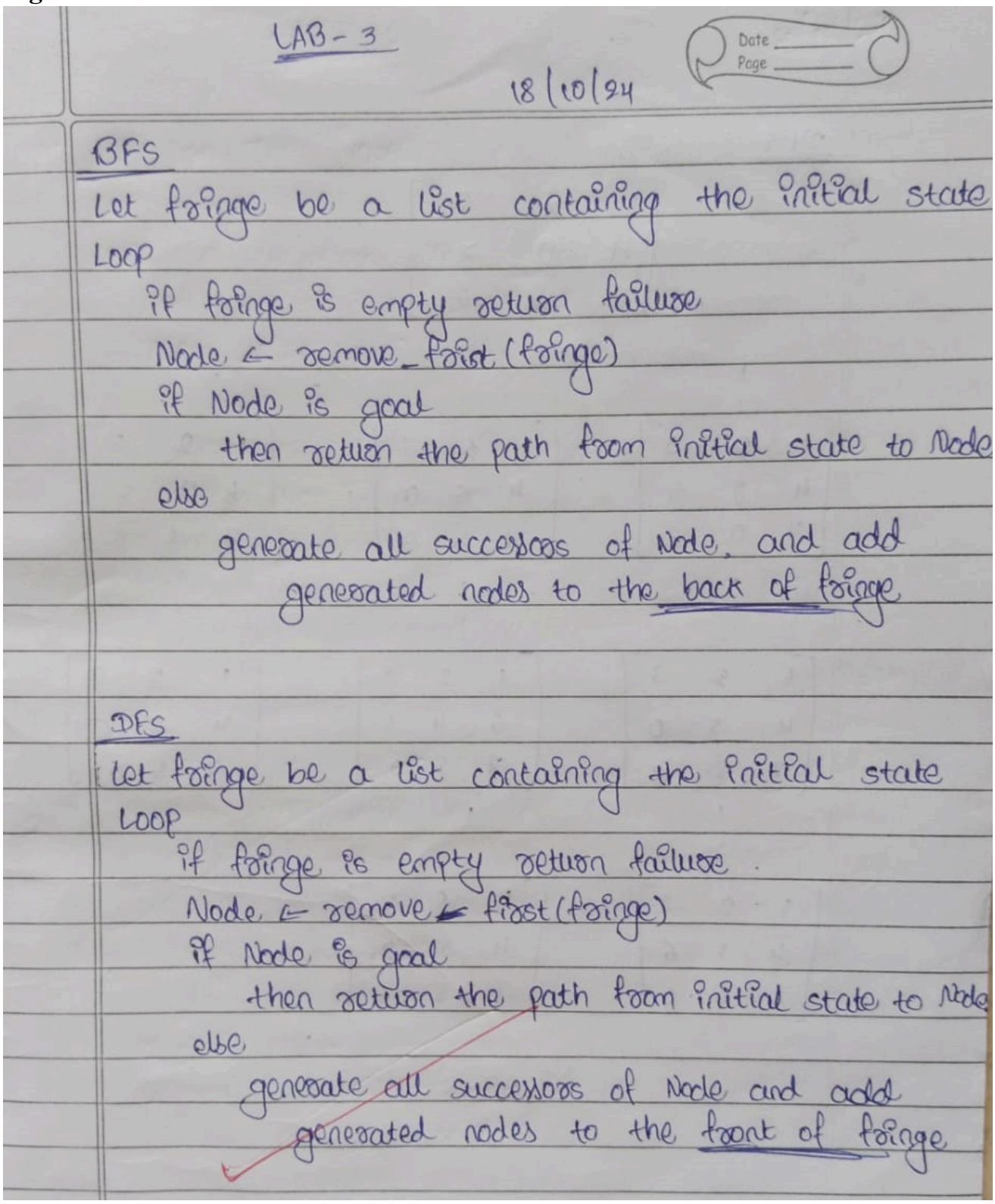
Output:

```
Enter Location of Vacuum (A or B): B
Enter status of B (0 for Clean, 1 for Dirty): 1
Enter status of other room (0 for Clean, 1 for Dirty): 0
Initial Location Condition: {'A': '0', 'B': '0'}
Location B is Dirty.
Location B has been Cleaned. Cost for CLEANING: 1
GOAL STATE: {'A': '0', 'B': '0'}
Performance Measurement: 1
Vanitha - 1BM22CS317
```

Program 3

Solve 8 puzzle problems using DFS an BFS.

Algorithm :



LAB - 3

18/10/24

Date
Page

BFS

Let fringe be a list containing the initial state
Loop
  if fringe is empty return failure
  Node ← remove_first(fringe)
  if Node is goal
    then return the path from initial state to Node
  else
    generate all successors of Node, and add
      generated nodes to the back of fringe

DFS

Let fringe be a list containing the initial state
Loop
  if fringe is empty return failure
  Node ← remove_first(fringe)
  if Node is goal
    then return the path from initial state to Node
  else
    generate all successors of Node and add
      generated nodes to the front of fringe

Code:

```
from copy import deepcopy

DIRECTIONS = [(-1, 0), (1, 0), (0, -1), (0, 1)]

class PuzzleState:
    def __init__(self, board, parent=None, move=""):
        self.board = board
        self.parent = parent
        self.move = move

    def get_blank_position(self):
        for i in range(3):
            for j in range(3):

                if self.board[i][j] == 0:
                    return i, j

    def generate_successors(self):
        successors = []
        x, y = self.get_blank_position()

        for dx, dy in DIRECTIONS:
            new_x, new_y = x + dx, y + dy

            if 0 <= new_x < 3 and 0 <= new_y < 3:
                new_board = deepcopy(self.board)
                                    new_board[x][y],  new_board[new_x][new_y]  =
new_board[new_x][new_y], new_board[x][y]

                successors.append(PuzzleState(new_board, parent=self))

        return successors

    def is_goal(self, goal_state):
        return self.board == goal_state

    def __str__(self):
        return "\n".join([" ".join(map(str, row)) for row in self.board])

def depth_limited_search(current_state, goal_state, depth):
    if depth == 0 and current_state.is_goal(goal_state):
        return current_state

    if depth > 0:
        for successor in current_state.generate_successors():
            found = depth_limited_search(successor, goal_state, depth - 1)
            if found:
                return found
    return None

def iterative_deepening_search(start_state, goal_state, max_depth):
    for depth in range(max_depth + 1):

        print(f"\nSearching at depth level: {depth}")

        result = depth_limited_search(start_state, goal_state, depth)
```

```
        if result:
            return result
    return None

def get_user_input():
    print("Enter the start state (use 0 for the blank):")
    start_state = []

    for _ in range(3):
        row = list(map(int, input().split()))
        start_state.append(row)

    print("Enter the goal state (use 0 for the blank):")
    goal_state = []

    for _ in range(3):
        row = list(map(int, input().split()))
        goal_state.append(row)

    max_depth = int(input("Enter the maximum depth for search: "))

    return start_state, goal_state, max_depth

def main():
    start_board, goal_board, max_depth = get_user_input()
    start_state = PuzzleState(start_board)
    goal_state = goal_board

    result = iterative_deepening_search(start_state, goal_state, max_depth)

    if result:
        print("\nGoal reached!")
        path = []
        while result:
            path.append(result)
            result = result.parent

        path.reverse()
        for state in path:
            print(state, "\n")
    else:
        print("Goal state not found within the specified depth.")

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

Output: (DFS)

```
Enter the start state (use 0 for the blank):
2 8 3
1 6 4
7 0  5
Enter the goal state (use 0 for the blank):
1 2 3
8 0 4
7 6 5
Enter the maximum depth for search: 5

Searching at depth level: 0

Searching at depth level: 1

Searching at depth level: 2

Searching at depth level: 3

Searching at depth level: 4

Searching at depth level: 5

Goal reached!
2 8 3
1 6 4
7 0 5

2 8 3
1 0 4
7 6 5

2 0 3
1 8 4
7 6 5

0 2 3
1 8 4
7 6 5

1 2 3
0 8 4
7 6 5

1 2 3
8 0 4
7 6 5
```

Code: (BFS)

```
#8 puzzle problem
from collections import deque

class PuzzleState:
    def __init__(self, board, zero_position, path=[]):
        self.board = board
        self.zero_position = zero_position
        self.path = path

    def is_goal(self):
        return self.board == [1, 2, 3, 4, 5, 6, 7, 8, 0]

    def get_possible_moves(self):
        moves = []
        row, col = self.zero_position
        directions = [(0, 1), (1, 0), (0, -1), (-1, 0)]  # Right, Down, Left, Up

        for dr, dc in directions:
            new_row, new_col = row + dr, col + dc
            if 0 <= new_row < 3 and 0 <= new_col < 3:
                new_board = self.board[:]
                # Swap zero with the adjacent tile
                    new_board[row * 3 + col], new_board[new_row * 3 + new_col] =
new_board[new_row * 3 + new_col], new_board[row * 3 + col]
                moves.append(PuzzleState(new_board, (new_row, new_col), self.path
+ [new_board]))

        return moves

def bfs(initial_state):
    queue = deque([initial_state])
    visited = set()

    while queue:
        current_state = queue.popleft()

        # Show the current board
        print("Current Board State:")
        print_board(current_state.board)
        print()

        if current_state.is_goal():
            return current_state.path

        visited.add(tuple(current_state.board))

        for next_state in current_state.get_possible_moves():
            if tuple(next_state.board) not in visited:
                queue.append(next_state)

    return None

def print_board(board):
    for i in range(3):
        print(board[i * 3:i * 3 + 3])
```

```
def main():
     print("Enter the initial state of the 8-puzzle (use 0 for the blank tile,
e.g., '1 2 3 4 5 6 7 8 0'): ")
    user_input = input()
    initial_board = list(map(int, user_input.split()))

    if len(initial_board) != 9 or set(initial_board) != set(range(9)):
        print("Invalid input! Please enter 9 numbers from 0 to 8.")
        return

    zero_position = initial_board.index(0)
     initial_state = PuzzleState(initial_board, (zero_position // 3, zero_position
% 3))

    solution_path = bfs(initial_state)

    if solution_path is None:
        print("No solution found.")
    else:
        print("Solution found in", len(solution_path), "steps.")
        for step in solution_path:
            print_board(step)
            print()

if __name__ == "__main__":
    main()

print("Vanitha - 1BM22CS317")
```

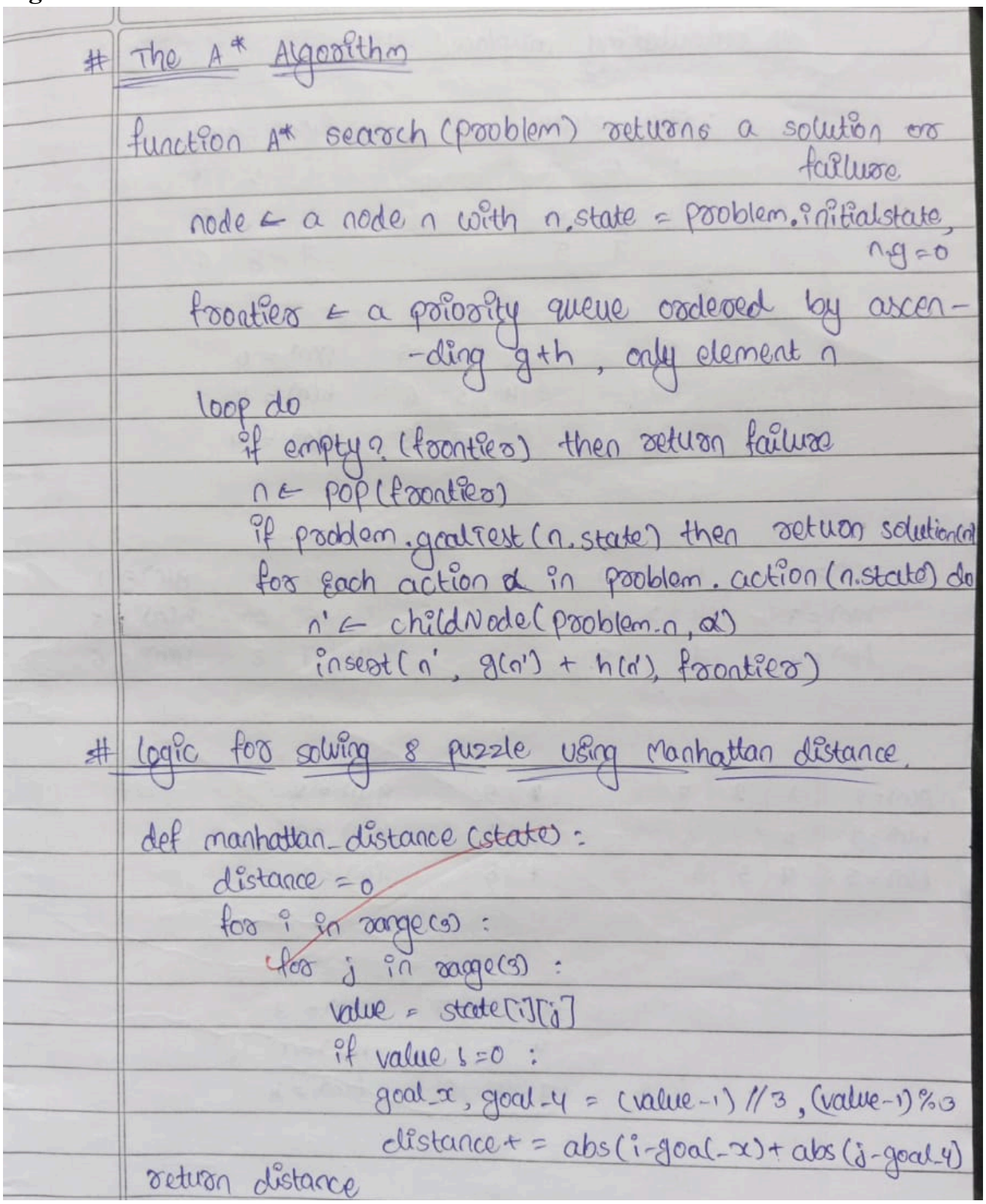
Output: (BFS)

```
Enter the initial state of the 8-puzzle (use 0 for the blank tile, e.g., '1 2 3 4 5 6 7 8 0'):
1 2 3 4  5 6 7 0 8
Current Board State:
[1, 2, 3]
[4, 5, 6]
[7, 0, 8]

Current Board State:
[1, 2, 3]
[4, 5, 6]
[7, 8, 0]

Solution found in 1 steps.
[1, 2, 3]
[4, 5, 6]
[7, 8, 0]

Vanitha - 1BM22CS317
```

Program 4

Implement A* search algorithm.

Algorithm :



The A* Algorithm

function A* search (problem) returns a solution or failure

node ← a node n with n.state = problem.initialstate, n.g = 0

frontier ← a priority queue ordered by ascending g+h, only element n

loop do

if empty? (frontier) then return failure

n ← pop(frontier)

if problem.goalTest(n.state) then return solution(n)

for each action a in problem.action(n.state) do

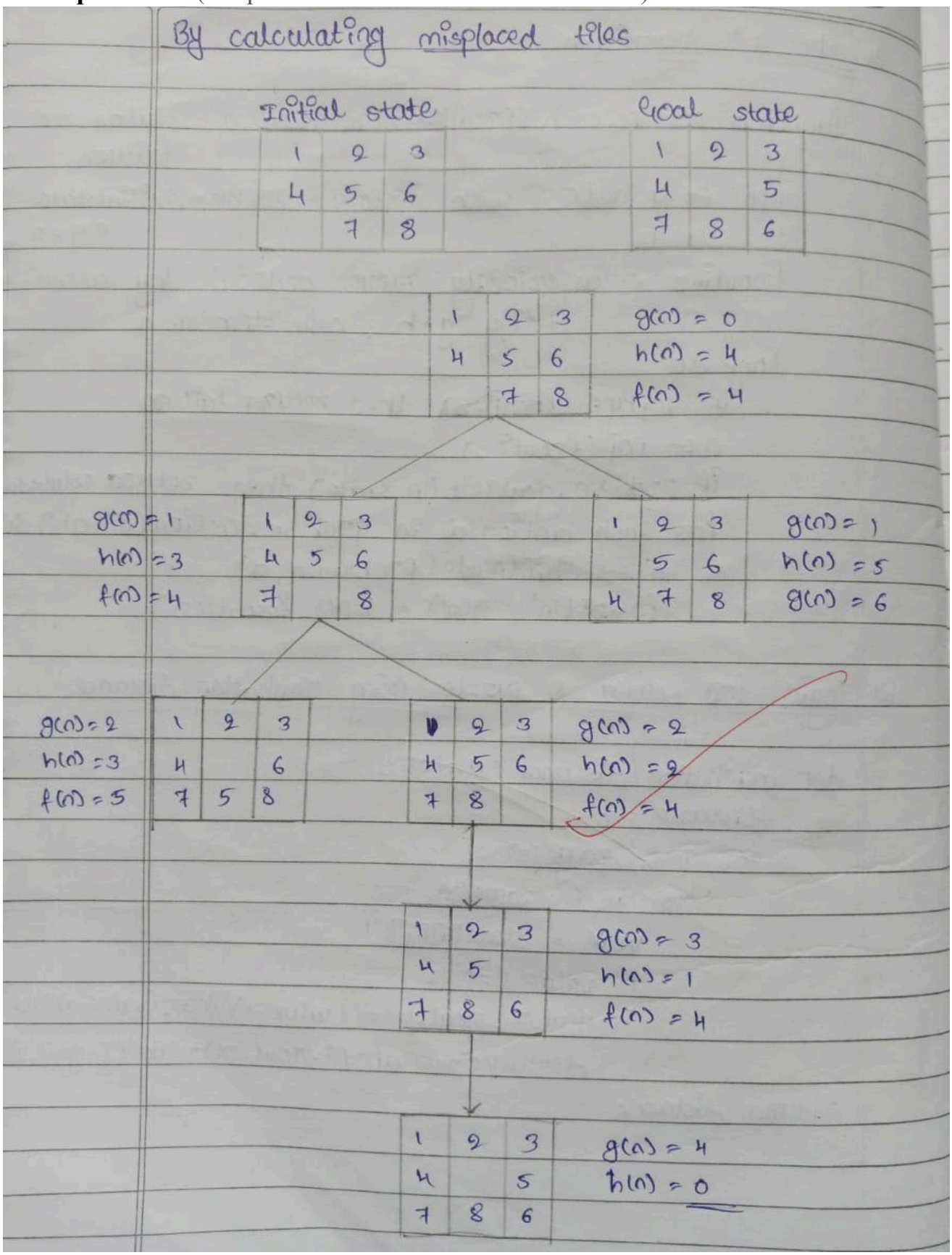
n' ← childNode(problem, n, a)

insert(n', g(n') + h(n'), frontier)

logic for solving 8 puzzle using manhattan distance

```
def manhattan_distance(state):
    distance = 0
    for i in range(3):
        for j in range(3):
            value = state[i][j]
            if value != 0:
                goal_x, goal_y = (value-1)//3, (value-1)%3
                distance += abs(i-goal_x) + abs(j-goal_y)
return distance
```

State space tree: (Misplaced Tiles and Manhattan distance)



Date
Page

By calculating manhattan distance

Initial state

1	2	3
4	5	6
	7	8

Goal state

1	2	3
4		5
7	8	6

1	2	3
4	5	6
	7	8

$g(n) = 0$
$h(n) = 1+1+1+1 = 4$
$f(n) = 4$

$g(n) = 1$
$h(n) = 1+1+1$
$f(n) = 4$

1	2	3
4	5	6
	7	8

1	2	3
	5	6
4	7	8

$g(n) = 1$
$h(n) = 1+1+1+1$
$f(n) = 6$

$g(n) = 2$
$h(n) = 3$
$f(n) = 5$

1	2	3
4		6
7	5	8

1	2	3
4	5	6
7	8	

$g(n) = 2$
$h(n) = 1+1 = 2$
$f(n) = 4$

25/10/24

1	2	3
4	5	
7	8	6

$g(n) = 3$
$h(n) = 1$
$f(n) = 4$

1	2	3
4		5
7	8	6

$g(n) = 4$
$h(n) = 0$

Code: (Misplaced Tiles)

```
#A* Search Algorithm- Misplace Tiles
import heapq

# A* search algorithm to solve the 8-puzzle problem
class Puzzle:
    def __init__(self, board, goal):
        self.board = board
        self.goal = goal
        self.n = len(board)

    def find_zero(self, state):
        # Find the index of the empty tile (0)
        for i in range(self.n):
            for j in range(self.n):
                if state[i][j] == 0:
                    return i, j

    def is_goal(self, state):
        return state == self.goal

    def possible_moves(self, state):
        # Generate all possible moves (up, down, left, right)
        moves = []
        i, j = self.find_zero(state)
        if i > 0:  # Up
            new_state = [row[:] for row in state]
                        new_state[i][j], new_state[i-1][j] = new_state[i-1][j],
new_state[i][j]
            moves.append(new_state)
        if i < self.n - 1:  # Down
            new_state = [row[:] for row in state]
                        new_state[i][j], new_state[i+1][j] = new_state[i+1][j],
new_state[i][j]
            moves.append(new_state)
        if j > 0:  # Left
            new_state = [row[:] for row in state]
                        new_state[i][j], new_state[i][j-1] = new_state[i][j-1],
new_state[i][j]
            moves.append(new_state)
        if j < self.n - 1:  # Right
            new_state = [row[:] for row in state]
                        new_state[i][j], new_state[i][j+1] = new_state[i][j+1],
new_state[i][j]
            moves.append(new_state)
        return moves

    def h(self, state):
        # Heuristic: Number of misplaced tiles
        misplaced = 0
        for i in range(self.n):
            for j in range(self.n):
                if state[i][j] != 0 and state[i][j] != self.goal[i][j]:
                    misplaced += 1
        return misplaced

    def astar(self):
```

```
        # A* search algorithm
        frontier = []
           heapq.heappush(frontier, (self.h(self.board), 0, self.board, []))   #
(f(n), g(n), state, path)
        explored = set()

        while frontier:
            f, g, state, path = heapq.heappop(frontier)

            if self.is_goal(state):
                return path + [state]  # Return the solution path

            explored.add(str(state))

            for move in self.possible_moves(state):
                if str(move) not in explored:
                     heapq.heappush(frontier, (g + 1 + self.h(move), g + 1, move,
path + [state]))

        return None

def print_puzzle(path):
    for step in path:
        for row in step:
            print(row)
        print()

# Initial state of the 8-puzzle
initial_state = [
    [2, 8, 3],
    [6, 4, 1],
    [7, 0, 5]
]

# Final state (goal state)
goal_state = [
    [1, 2, 3],
    [8, 0, 4],
    [7, 6, 5]
]

# Solve the puzzle
puzzle = Puzzle(initial_state, goal_state)
solution = puzzle.astar()

print("Vanitha - 1BM22CS317\n")

if solution:
    print("Solution found!")
    print_puzzle(solution)
     print(f"Number of steps to solution: {len(solution) - 1}")   # Exclude the
initial state from the count
else:
    print("No solution found.")
```

Output: (Misplaced Tiles)

```
Vanitha - 1BM22CS317

Solution found!
[2, 8, 3]
[6, 4, 1]
[7, 0, 5]

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

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

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

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

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

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

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

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

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

Number of steps to solution: 19
```

Code: (Manhattan distance)

```
#A* Search Algorithm- Manhattan Distance

import heapq

# A* search algorithm to solve the 8-puzzle problem using Manhattan Distance
class Puzzle:
    def __init__(self, board, goal):
        self.board = board
        self.goal = goal
        self.n = len(board)

    def find_zero(self, state):
        # Find the index of the empty tile (0)
        for i in range(self.n):
            for j in range(self.n):
                if state[i][j] == 0:
                    return i, j

    def is_goal(self, state):
        return state == self.goal

    def possible_moves(self, state):
        # Generate all possible moves (up, down, left, right)
        moves = []
        i, j = self.find_zero(state)
        if i > 0:  # Up
            new_state = [row[:] for row in state]
                        new_state[i][j], new_state[i-1][j] = new_state[i-1][j],
new_state[i][j]
            moves.append(new_state)
        if i < self.n - 1:  # Down
            new_state = [row[:] for row in state]
                        new_state[i][j], new_state[i+1][j] = new_state[i+1][j],
new_state[i][j]
            moves.append(new_state)
        if j > 0:  # Left
            new_state = [row[:] for row in state]
                        new_state[i][j], new_state[i][j-1] = new_state[i][j-1],
new_state[i][j]
            moves.append(new_state)
        if j < self.n - 1:  # Right
            new_state = [row[:] for row in state]
                        new_state[i][j], new_state[i][j+1] = new_state[i][j+1],
new_state[i][j]
            moves.append(new_state)
        return moves

    def manhattan_distance(self, state):
        # Heuristic: Manhattan Distance
        distance = 0
        for i in range(self.n):
            for j in range(self.n):
                if state[i][j] != 0:
                    # Find the goal position of the current tile
                    goal_i, goal_j = divmod(self.goal[i][j], self.n)
                    distance += abs(i - goal_i) + abs(j - goal_j)
```

```
        return distance

    def astar(self):
        # A* search algorithm
        frontier = []
              heapq.heappush(frontier,  (self.manhattan_distance(self.board),  0,
self.board, []))  # (f(n), g(n), state, path)
        explored = set()

        while frontier:
            f, g, state, path = heapq.heappop(frontier)

            if self.is_goal(state):
                return path + [state]  # Return the solution path

            explored.add(str(state))

            for move in self.possible_moves(state):
                if str(move) not in explored:
                                            heapq.heappush(frontier,  (g  +  1  +
self.manhattan_distance(move), g + 1, move, path + [state]))

        return None

def print_puzzle(path):
    for step in path:
        for row in step:
            print(row)
        print()

# Initial state of the 8-puzzle
initial_state = [
    [2, 8, 3],
    [1, 6, 4],
    [7, 0, 5]
]

# Final state (goal state)
goal_state = [
    [1, 2, 3],
    [8, 0, 4],
    [7, 6, 5]
]

# Solve the puzzle
puzzle = Puzzle(initial_state, goal_state)
solution = puzzle.astar()

print("Vanitha - 1BM22CS317\n")

if solution:
    print("Solution found!")
    print_puzzle(solution)
     print(f"Number of steps to solution: {len(solution) - 1}")  # Exclude the
initial state from the count
else:
    print("No solution found.")
```

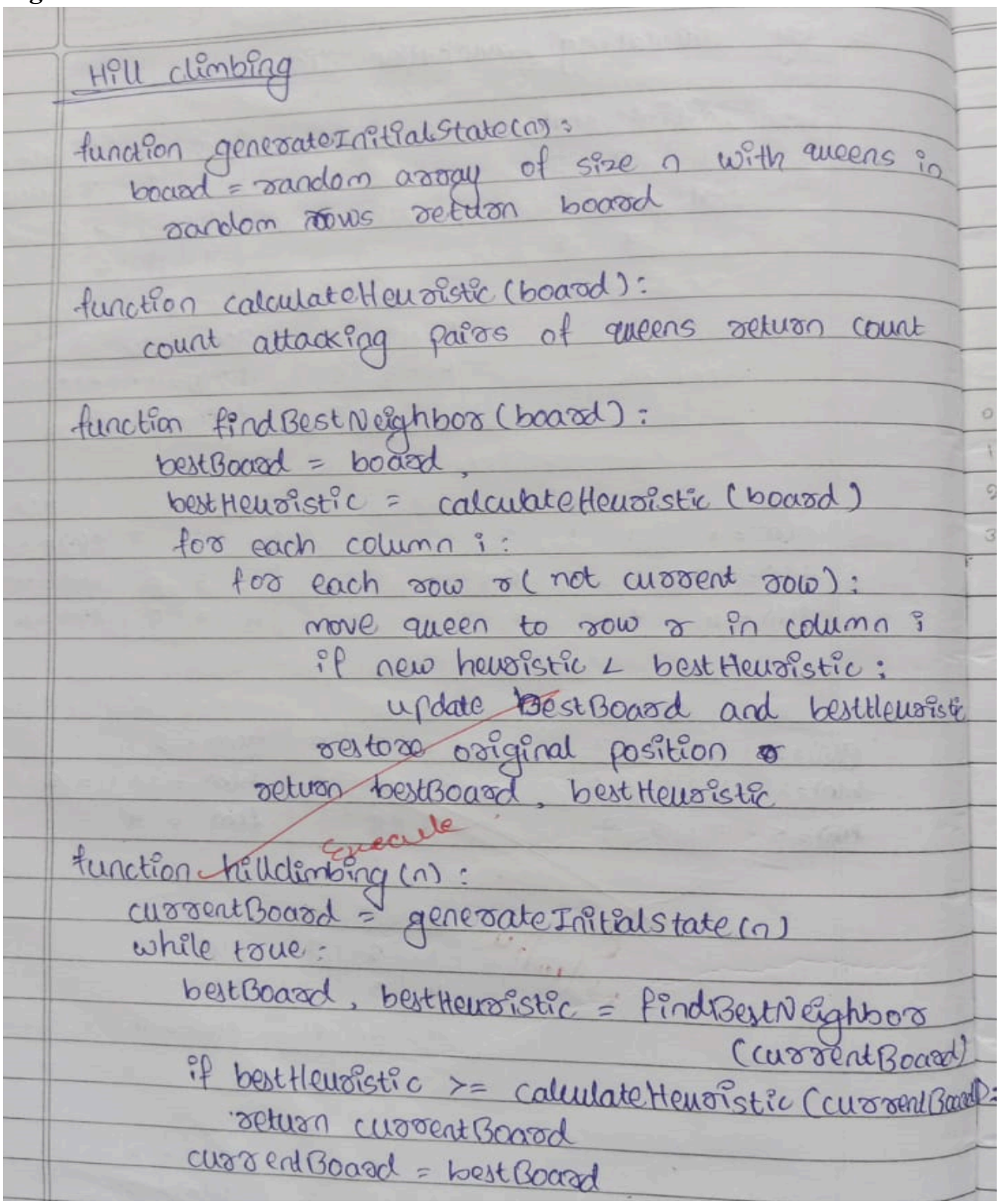
Output:

```
Vanitha - 1BM22CS317

Solution found!
[2, 8, 3]
[1, 6, 4]
[7, 0, 5]

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

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

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

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

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

Number of steps to solution: 5
```

Program 5

Implement Hill Climbing Algorithm.

Algorithm :

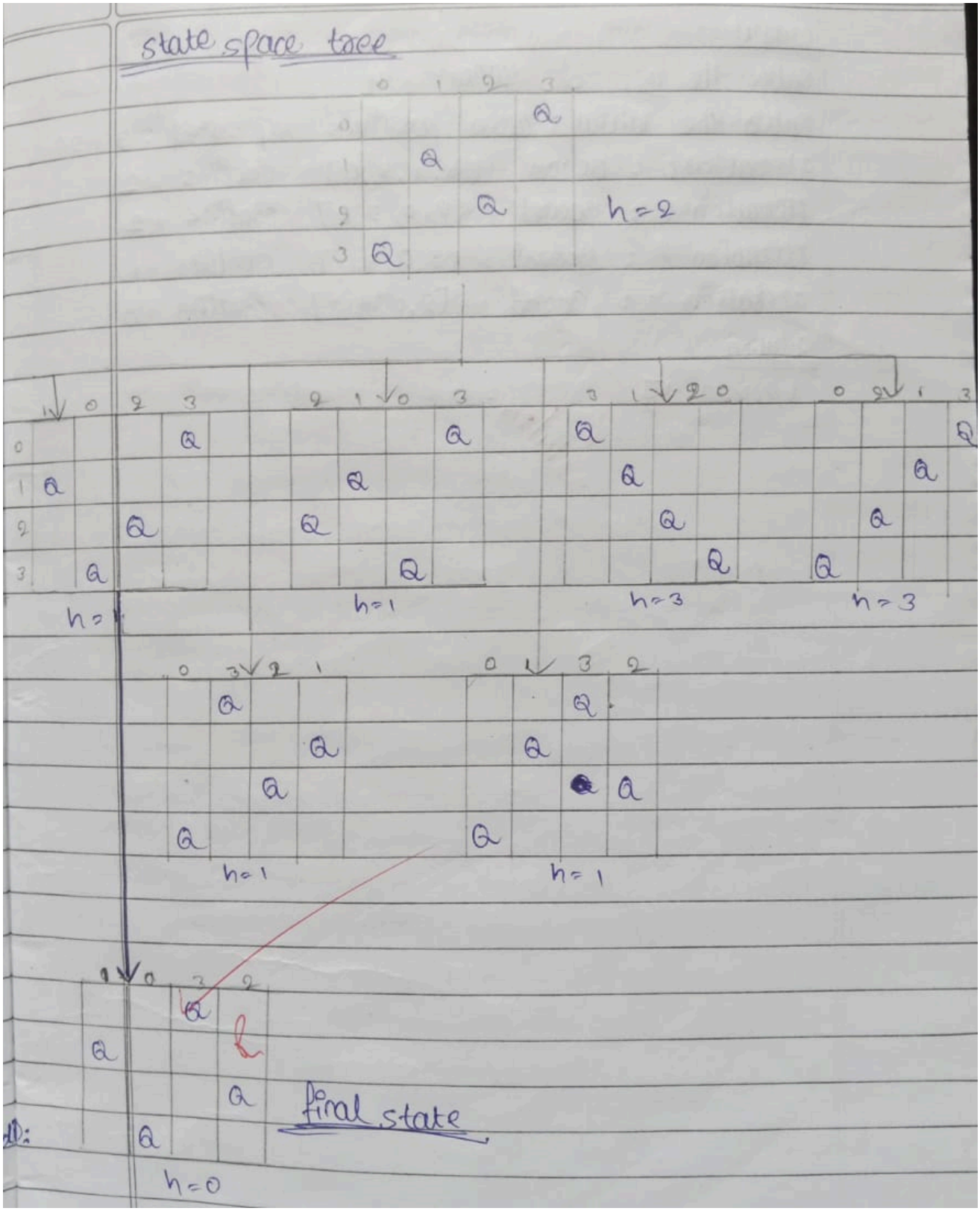


Hill climbing

```
function generateInitialState(n):
    board = random array of size n with queens in
    random rows return board

function calculateHeuristic(board):
    count attacking pairs of queens return count

function findBestNeighbor(board):
    bestBoard = board,
    bestHeuristic = calculateHeuristic(board)
    for each column i:
        for each row r (not current row):
            move queen to row r in column i
            if new heuristic < bestHeuristic:
                update BestBoard and bestHeuristic
            restore original position
        return bestBoard, bestHeuristic

function hillclimbing(n):
    currentBoard = generateInitialState(n)
    while true:
        bestBoard, bestHeuristic = findBestNeighbor
                                   (currentBoard)
        if bestHeuristic >= calculateHeuristic(currentBoard):
            return currentBoard
        currentBoard = bestBoard
```

Execute

State space tree:



Code:

```
import random

# Function to calculate the number of conflicts in the current configuration
def calculate_conflicts(board):
    n = len(board)
    conflicts = 0

    for i in range(n):
        for j in range(i + 1, n):
            if board[i] == board[j]:
                conflicts += 1  # Same column
            elif abs(board[i] - board[j]) == abs(i - j):
                conflicts += 1  # Same diagonal

    return conflicts

# Function to generate a random configuration of queens (if needed)
def random_board(n):
    return [random.randint(0, n - 1) for _ in range(n)]

# Hill climbing algorithm to solve the N-Queens problem
def hill_climbing(n, current_board):
    current_conflicts = calculate_conflicts(current_board)
    iteration = 0

    # Repeat until we reach a solution or cannot improve
    while current_conflicts != 0:
        iteration += 1
        print(f"Iteration {iteration}, Conflicts: {current_conflicts}")
        print_board(current_board)

        neighbors = []

        # Generate neighbors by moving each queen to a different row in its
column
        for i in range(n):
            for row in range(n):
                if row != current_board[i]:
                    new_board = current_board[:]
                    new_board[i] = row
                    neighbors.append(new_board)

        # Find the neighbor with the least conflicts
        best_neighbor = None
        best_conflicts = current_conflicts

        for neighbor in neighbors:
            neighbor_conflicts = calculate_conflicts(neighbor)
            if neighbor_conflicts < best_conflicts:
                best_conflicts = neighbor_conflicts
                best_neighbor = neighbor

        # If no better neighbor is found, we are stuck, return current board
        if best_conflicts >= current_conflicts:
            print("Stuck in local optimum.")
            return current_board
```

```
            # Otherwise, move to the best neighbor
            current_board = best_neighbor
            current_conflicts = best_conflicts

    print(f"Final Iteration {iteration + 1}, Conflicts: {current_conflicts}")
    print_board(current_board)  # Final solution
    return current_board

# Function to print the board in a readable format
def print_board(board):
    n = len(board)
    for i in range(n):
        row = ['Q' if col == board[i] else '.' for col in range(n)]
        print(' '.join(row))
    print()



# Function to get user input for the initial configuration of the queens
def get_user_input(n):
    while True:
        try:
            input_str = input("Enter the list of column positions
(space-separated): ")
            board = [int(x) for x in input_str.split()]

            if len(board) != n:
                continue

            if any(x < 0 or x >= n for x in board):
                continue

            if len(set(board)) != n:
                continue

            return board

        except (ValueError):
            continue

# Main function to handle user input and run the algorithm
def main():
    while True:
        try:
            n = int(input("Enter the size of the board (N): "))
            if n <= 0:
                continue
            else:
                break
        except ValueError:
            continue

    initial_board = get_user_input(n)

    max_restarts = 10
    restart_count = 0

    while restart_count < max_restarts:
```

```
        solution = hill_climbing(n, initial_board)

        if calculate_conflicts(solution) == 0:
            return
        else:
            restart_count += 1
            initial_board = random_board(n)  # Restart with a random
configuration

    print("\nFailed to find a solution after several attempts.")

# Run the program
if __name__ == "__main__":
    main()

print("Vanitha - 1BM22CS317")
```

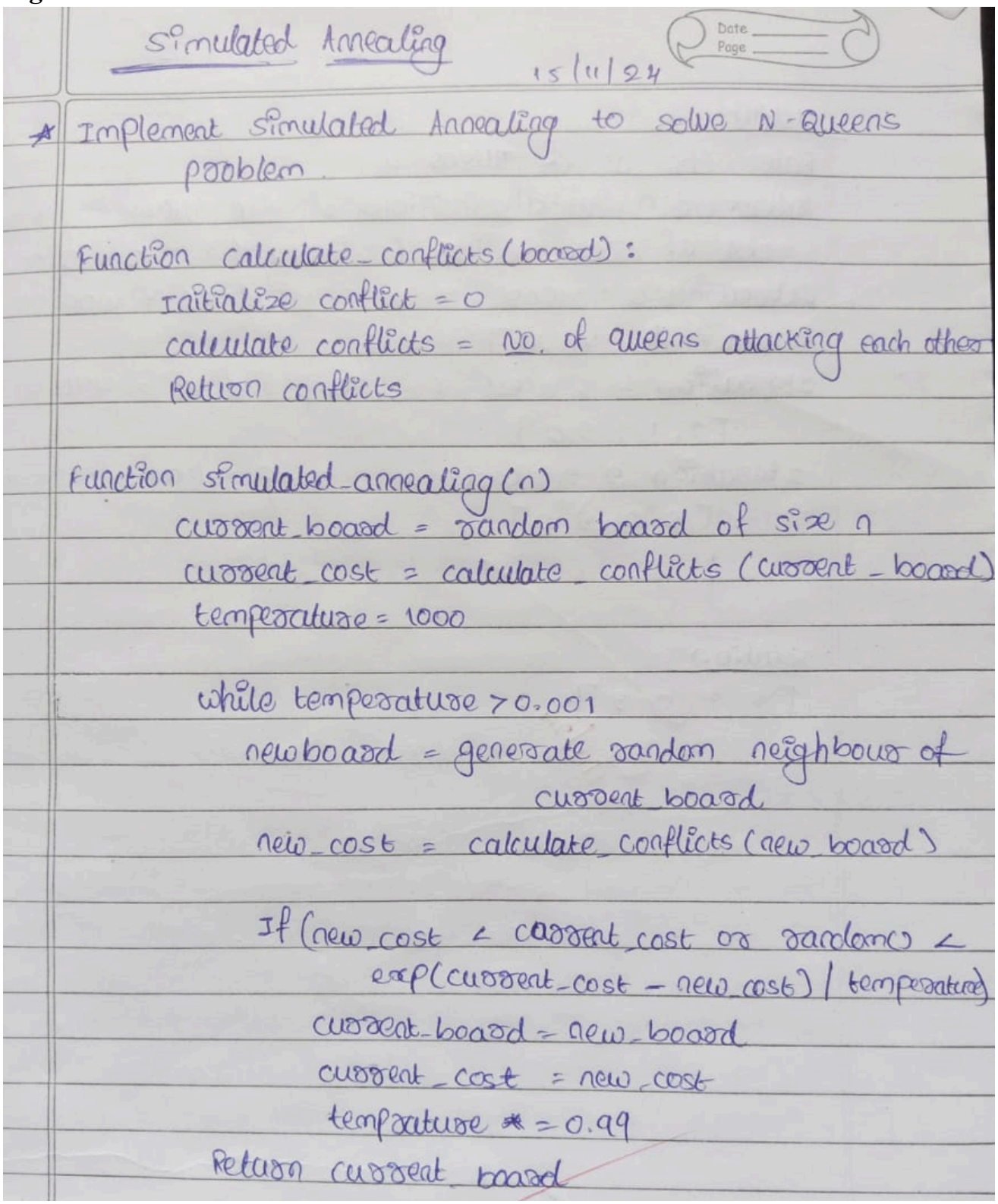
Output:

```
Enter the size of the board (N): 4
Enter the list of column positions (space-separated): 3 1 2 0
Iteration 1, Conflicts: 2
. . . Q
. Q . .
. . Q .
Q . . .

Stuck in local optimum.
Iteration 1, Conflicts: 2
. Q . .
. . . Q
. . Q .
. . Q .

Final Iteration 2, Conflicts: 0
. Q . .
. . . Q
Q . . .
. . Q .

Vanitha - 1BM22CS317
```

Program 6

Write a program to implement Simulated Annealing Algorithm

Algorithm :



Simulated Annealing

15/11/24

Implement Simulated Annealing to solve N-Queens problem.

Function calculate_conflicts(board):
    Initialize conflict = 0
    calculate conflicts = No. of queens attacking each other
    Return conflicts

Function simulated_annealing(n)
    current_board = random board of size n
    current_cost = calculate_conflicts(current_board)
    temperature = 1000

    while temperature > 0.001
        newboard = generate random neighbour of current_board
        new_cost = calculate_conflicts(new_board)

        If (new_cost < current_cost or random() < exp(current_cost - new_cost) / temperature)
            current_board = new_board
            current_cost = new_cost
            temperature *= 0.99
    Return current_board

Code:

```
#Simulated Annealing to solve N Queens problem
import random
import math

def generateRandomBoard(n):
    """Generate a random board represented by a permutation of columns."""
    return random.sample(range(n), n)

def generateNeighbor(state):
    """Generate a neighbor by moving one queen to another row in its column."""
    newState = state[:]
    row = random.randint(0, len(state) - 1)
    newRow = random.randint(0, len(state) - 1)

    # Ensure the queen moves to a different row (no self-move)
    while newRow == newState[row]:
        newRow = random.randint(0, len(state) - 1)

    newState[row] = newRow
    return newState

def calculateEnergy(state):
    """Calculate the number of attacking pairs of queens."""
    conflicts = 0
    n = len(state)

    for i in range(n):
        for j in range(i + 1, n):
            if state[i] == state[j]:  # Same column
                conflicts += 1
            if abs(state[i] - state[j]) == abs(i - j):  # Same diagonal
                conflicts += 1
    return conflicts

def simulatedAnnealing(n, initialTemperature=1000, minimumTemperature=0.01,
alpha=0.95, numberOfIterations=100):
    """Simulated Annealing algorithm to solve the N-Queens problem."""
    # Generate an initial random board
    currentState = generateRandomBoard(n)
    currentEnergy = calculateEnergy(currentState)
    T = initialTemperature

    print("Starting simulated annealing...")

    # Iterate while the temperature is above the minimum
    iteration = 0
    while T > minimumTemperature:
        iteration += 1
        print(f"\nIteration {iteration}, Temperature: {T:.4f}")
        print(f"Current state: {currentState}")
        print(f"Current energy (conflicts): {currentEnergy}")

        for _ in range(numberOfIterations):
            # Generate a random neighbor (new board configuration)
            nextState = generateNeighbor(currentState)
            nextEnergy = calculateEnergy(nextState)
```

```
            # If the next state is better or accepted by probability
            if nextEnergy < currentEnergy:
                currentState = nextState
                currentEnergy = nextEnergy
            else:
                deltaEnergy = nextEnergy - currentEnergy
                probability = math.exp(-deltaEnergy / T)
                if random.random() < probability:
                    currentState = nextState
                    currentEnergy = nextEnergy

        # Cool the temperature
        T *= alpha

        # If a solution with zero conflicts is found, return the solution
        if currentEnergy == 0:
            return currentState

    # Return the best state found if no perfect solution is found
    return currentState

# Function to handle user input for the board configuration
def getUserInput(n):
    """Get user input for the initial configuration of queens."""
    while True:
        try:
            # Ask the user for the queen positions
            user_input = input(f"Enter the initial positions of the queens (0 to
{n-1} for {n} queens, separated by spaces): ")
            positions = list(map(int, user_input.split()))

            # Validate the input
            if len(positions) != n:
                print(f"Error: You must provide exactly {n} positions.")
                continue

            # Ensure no duplicates (no two queens in the same column)
            if len(set(positions)) != n:
                print("Error: Two queens cannot be placed in the same column.")
                continue

            # Ensure the positions are within the valid range (0 to n-1)
            if any(pos < 0 or pos >= n for pos in positions):
                print(f"Error: Queen positions must be between 0 and {n-1}.")
                continue

            return positions
        except ValueError:
            print("Error: Invalid input. Please enter integers separated by
spaces.")


# Main function to get user input and run the algorithm
if __name__ == "__main__":
    n = int(input("Enter the number of queens (n): "))

    # Get user input for the initial positions of the queens
    initial_state = getUserInput(n)
```

```
    # Calculate energy for the initial state
    initial_energy = calculateEnergy(initial_state)
    print(f"Initial board configuration: {initial_state}")
    print(f"Initial energy (conflicts): {initial_energy}")

    # Run the simulated annealing algorithm
    solution = simulatedAnnealing(n)

    # Output the results
    if calculateEnergy(solution) == 0:
        print("\nSolution found!")
        print(solution)
    else:
        print("\nNo perfect solution found. Best found solution:")
        print(solution)

print("Vanitha - 1BM22CS317")
```

Output:

```
Enter the number of queens (n): 4
Enter the initial positions of the queens (0 to 3 for 4 queens, separated by spaces): 3 2 1 0
Initial board configuration: [3, 2, 1, 0]
Initial energy (conflicts): 6
Starting simulated annealing...

Iteration 1, Temperature: 1000.0000
Current state: [2, 0, 1, 3]
Current energy (conflicts): 1

Iteration 2, Temperature: 950.0000
Current state: [0, 1, 0, 1]
Current energy (conflicts): 5

Iteration 3, Temperature: 902.5000
Current state: [2, 2, 3, 0]
Current energy (conflicts): 3
```



```
          |
          |
Iteration 106, Temperature: 4.5812
Current state: [3, 3, 1, 2]
Current energy (conflicts): 3

Iteration 107, Temperature: 4.3521
Current state: [1, 3, 0, 3]
Current energy (conflicts): 1

Iteration 108, Temperature: 4.1345
Current state: [0, 3, 0, 0]
Current energy (conflicts): 3

Solution found!
[2, 0, 3, 1]
Vanitha - 1BM22CS317
```

Program 7

Implement unification in first order logic.

Algorithm :

Algorithm unify (ψ_1, ψ_2)

1. If ψ_1 or ψ_2 are variable or constant then :
   (a) If ψ_1 or ψ_2 are identical, then return FAILURE
   (b) Else if ψ_1 is variable
      a. then if ψ_1 occurs in ψ_2 then return FAILURE
      b. Else return {(ψ_2/ψ_1)}
   (c) Else if ψ_2 is variable
      a. then if ψ_2 occurs in ψ_1 then return FAILURE
      b. Else return {(ψ_1/ψ_2)}
   (d) Else return FAILURE

2. If the initial predicate symbol in ψ_1 & ψ_2 are not same then return FAILURE

3. If ψ_1 & ψ_2 have different no. of arguments then return FAILURE

4. Set substitution set (SUBST) to NIL

5. For i=1 to the no. of elements in ψ_1
   (a) Call unify function with ith element of ψ_1 & ith element of ψ_2 & put result into S
   (b) If S = failure then returns FAILURE
   (c) If S ≠ NIL then do
      a. Apply S to the remainders of both L1 & L2
      b. SUBST = APPEND (S, SUBST)

6. Return SUBST

Code:

```
def unify(x1, x2):
    """
    Unify two expressions (x1 and x2) based on the given unification algorithm.
    Returns a substitution set (SUBST) or FAILURE if unification is not possible.
    """
    if is_variable_or_constant(x1) or is_variable_or_constant(x2):
        if x1 == x2:
            return []  # Return NIL (empty substitution set)
        elif is_variable(x1):
            if occurs_check(x1, x2):
                return "FAILURE"
            else:
                return [(x2, x1)]  # Return {x2/x1}
        elif is_variable(x2):
            if occurs_check(x2, x1):
                return "FAILURE"
            else:
                return [(x1, x2)]  # Return {x1/x2}
        else:
            return "FAILURE"

    if not is_same_predicate(x1, x2):
        return "FAILURE"

    if len(x1) != len(x2):
        return "FAILURE"

    subst = []  # Substitution set

    for i in range(len(x1)):
        s = unify(x1[i], x2[i])
        if s == "FAILURE":
            return "FAILURE"
        elif s:
            subst.extend(s)
            apply_substitution(s, x1[i+1:])
            apply_substitution(s, x2[i+1:])

    return subst

def is_variable_or_constant(expr):
    """Check if the expression is a variable or a constant."""
    return isinstance(expr, str) and expr.isalnum()

def is_variable(expr):
    """Check if the expression is a variable."""
    return isinstance(expr, str) and expr.islower()

def occurs_check(var, expr):
    """Check if the variable occurs in the expression."""
    if var == expr:
        return True
    elif isinstance(expr, (list, tuple)):
        return any(occurs_check(var, sub_expr) for sub_expr in expr)
    return False
```

```python
def is_same_predicate(x1, x2):
    """Check if the initial predicate symbols of x1 and x2 are the same."""
    if isinstance(x1, (list, tuple)) and isinstance(x2, (list, tuple)):
        return x1[0] == x2[0]
    return False

def apply_substitution(subst, expr):
    """Apply the substitution set to the given expression."""
    for old, new in subst:
        if expr == old:
            return new
        elif isinstance(expr, (list, tuple)):
            return [apply_substitution(subst, sub_expr) for sub_expr in expr]
    return expr

def parse_input(expr):
    """Parse user input into a list or tuple representing the predicate."""
    try:
        return eval(expr)
    except Exception as e:
        print(f"Error in input format: {e}")
        return None


# User Input
print("Vanitha - 1BM22CS317")

print("\nEnter two expressions to unify. Use list/tuple format.")
print("Example: ['P', 'x', 'a'] represents P(x, a)")

expr1_input = input("Enter the first expression: ")
expr2_input = input("Enter the second expression: ")

expr1 = parse_input(expr1_input)
expr2 = parse_input(expr2_input)

if expr1 is not None and expr2 is not None:
    result = unify(expr1, expr2)
    print("Unification Result:", result)
else:
    print("Invalid input format. Please try again.")
```

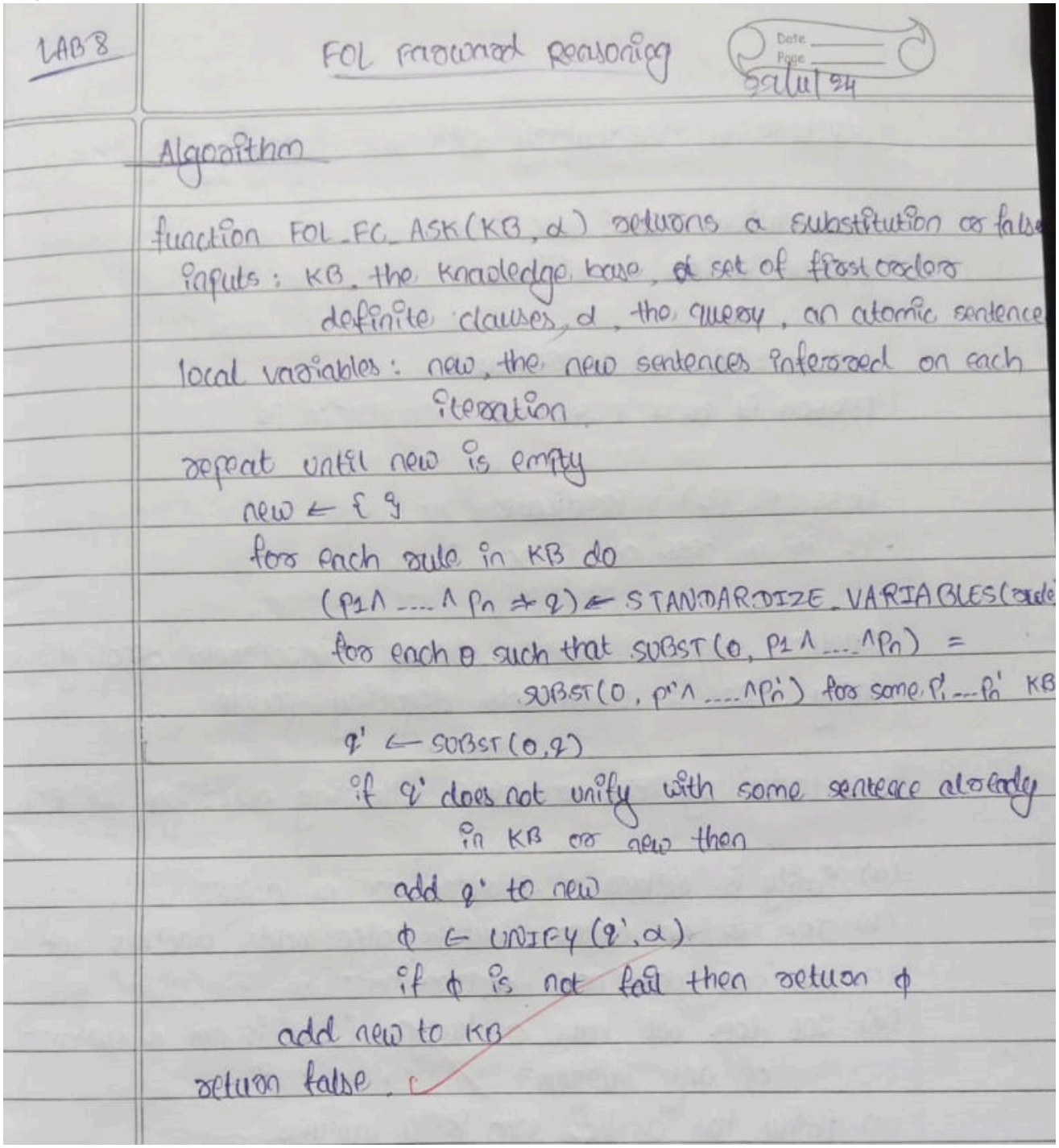
Output:

```
Enter the first expression (e.g., ['P', '?x', 'a']):
['P', 'a', '?X', ['f', '?Y']]
Enter the second expression (e.g., ['P', 'b', 'a']):
['P', '?Z', 'b', ['f', 'c']]
Unification Succeeded
Substitutions:
?Z -> a
?X -> b
?Y -> c
```

Program 8

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

Algorithm :



LAB 8

FOL Forward Reasoning

Date
Page

Algorithm

function FOL_FC_ASK(KB, α) returns a substitution or false

inputs: KB, the knowledge base, a set of first order definite clauses, α, the query, an atomic sentence

local variables: new, the new sentences inferred on each iteration

repeat until new is empty

new ← { }

for each rule in KB do

(p1 ∧ .... ∧ pn ⇒ q) ← STANDARDIZE_VARIABLES(rule

for each θ such that SUBST(θ, p1 ∧ .... ∧ pn) =

SUBST(θ, p'1 ∧ .... ∧ p'n) for some p'1 ... p'n KB

q' ← SUBST(θ, q)

if q' does not unify with some sentence already

in KB or new then

add q' to new

φ ← UNIFY(q', α)

if φ is not fail then return φ

add new to KB

return false

Code:

```
def forward_reasoning_algorithm():
    print("Enter the knowledge base (rules and facts), one per line. Enter 'END'
to finish:")

    # Initialize the knowledge base and facts
    knowledge_base = []
    facts = set()

    # Input: Knowledge base (rules and facts)
    while True:
        line = input().strip()
        if line.upper() == "END":
            break
        if "=>" in line:  # Rule with premises and conclusion
            premises, conclusion = line.split(" => ")
            knowledge_base.append((premises.split(" AND "), conclusion.strip()))
        else:  # Fact
            facts.add(line.strip())

    # Input: Query
    query = input("Enter the query (atomic sentence): ").strip()

    # Forward-chaining algorithm
    inferred = set()  # Store all inferred facts
    new_inferences = True

    while new_inferences:
        new_inferences = False
        for premises, conclusion in knowledge_base:
            # Check if all premises are satisfied in the current set of facts
            if all(p in facts for p in premises) and conclusion not in facts:
                # Infer the conclusion
                facts.add(conclusion)
                inferred.add(conclusion)
                print(f"Inferred: {conclusion}")
                new_inferences = True

        # Break if no new facts are inferred in this iteration
        if not new_inferences:
            break

    # Check if the query can be inferred
    if query in facts:
        print("Query satisfied: YES")
    else:
        print("Query satisfied: NO")


# Run the algorithm
forward_reasoning_algorithm()
```

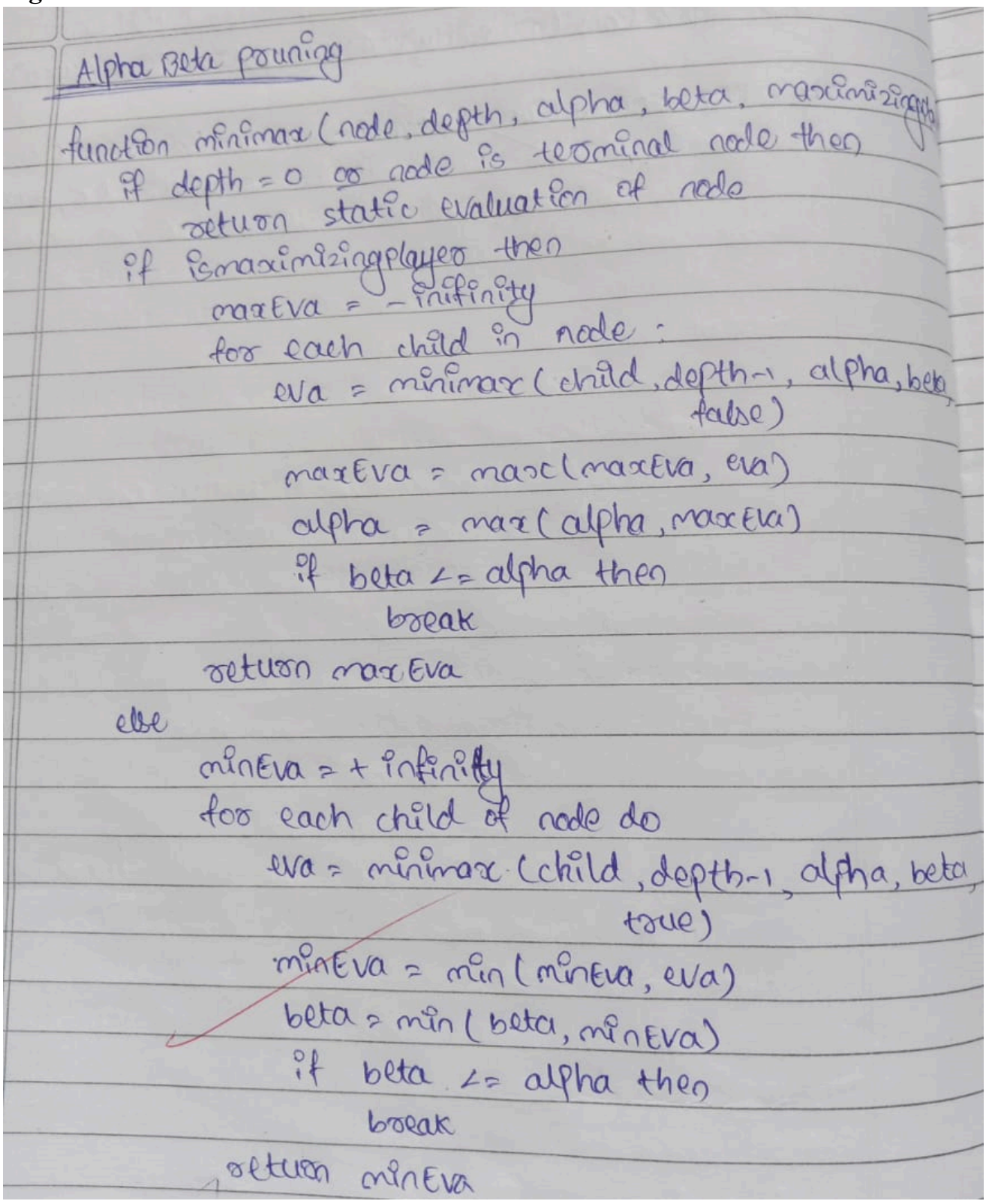
Output:

```
Enter the knowledge base (rules and facts), one per line. Enter 'END' to finish:
fever AND cough => flu
fever AND sore_throat => cold
fever AND headache => flu
fever
cough
END
Enter the query (atomic sentence): flu
Inferred: flu
Query satisfied: YES
```

Program 9

Implement Alpha-Beta Pruning.

Algorithm :



Alpha Beta pruning

```
function minimax(node, depth, alpha, beta, maximizingPlayer)
    if depth = 0 or node is terminal node then
        return static evaluation of node
    if ismaximizingplayer then
        maxEva = -infinity
        for each child in node :
            eva = minimax(child, depth-1, alpha, beta,
                                        false)
            maxEva = max(maxEva, eva)
            alpha = max(alpha, maxEva)
            if beta <= alpha then
                break
        return maxEva
    else
        minEva = +infinity
        for each child of node do
            eva = minimax(child, depth-1, alpha, beta,
                                        true)
            minEva = min(minEva, eva)
            beta = min(beta, minEva)
            if beta <= alpha then
                break
        return minEva
```

Code:

```
# working of Alpha-Beta Pruning
MAX, MIN = 1000, -1000

def minimax(depth, nodeIndex, maximizingPlayer,
                  values, alpha, beta):

            if depth == 3:
            return values[nodeIndex]

      if maximizingPlayer:

            best = MIN

            # Recur for left and right children
            for i in range(0, 2):

                  val = minimax(depth + 1, nodeIndex * 2 + i,
                                    False, values, alpha, beta)
                  best = max(best, val)
                  alpha = max(alpha, best)

                  # Alpha Beta Pruning
                  if beta <= alpha:
                        break

            return best

      else:
            best = MAX

            # Recur for left and
            # right children
            for i in range(0, 2):

                  val = minimax(depth + 1, nodeIndex * 2 + i,
                                      True, values, alpha, beta)
                  best = min(best, val)
                  beta = min(beta, best)

                  # Alpha Beta Pruning
                  if beta <= alpha:
                        break

            return best

# Driver Code
if __name__ == "__main__":

      values = [3, 5, 6, 9, 1, 2, 0, -1]
      print("The optimal value is :", minimax(0, 0, True, values, MIN, MAX))
```

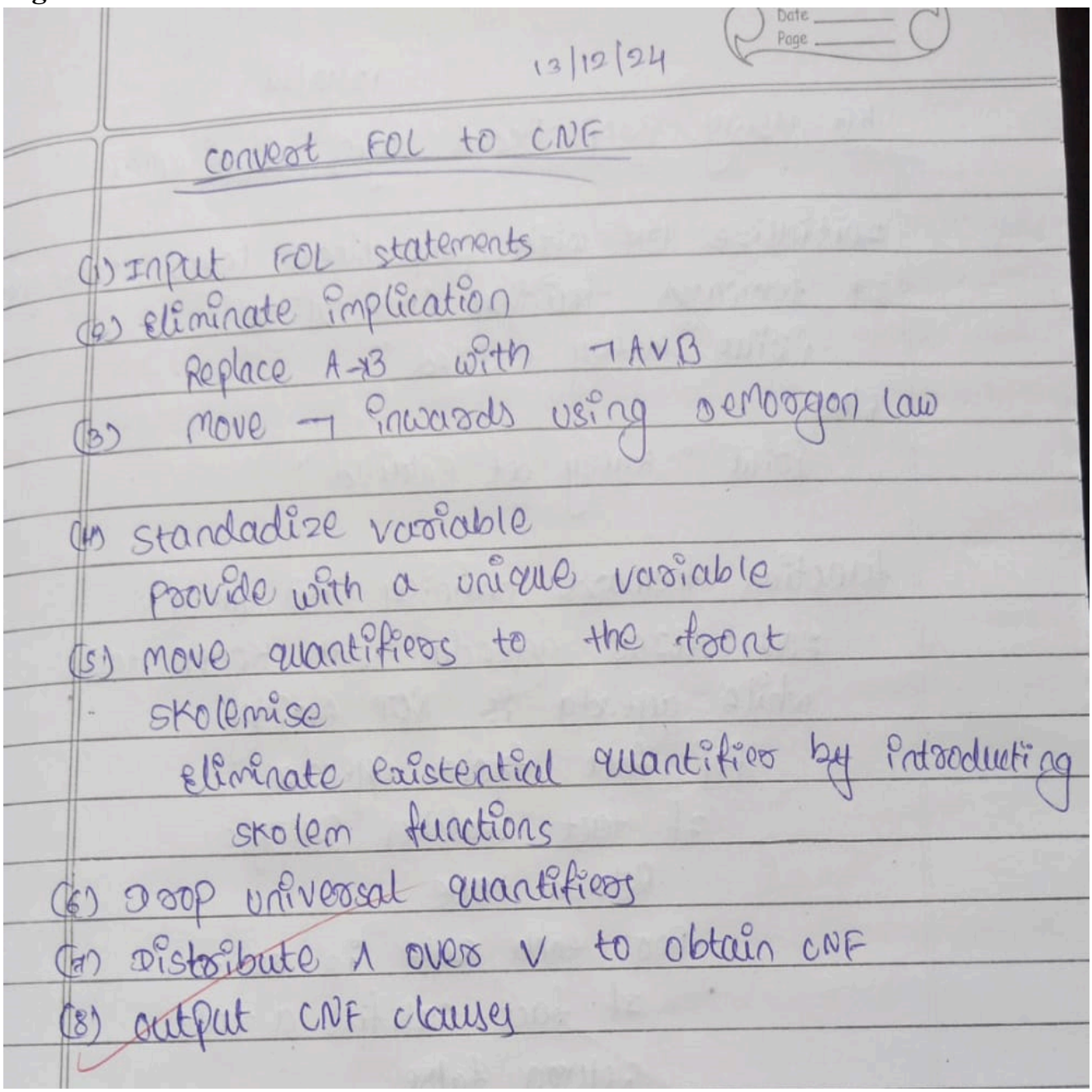
Output:

```
The optimal value is : 5
```

Program 10

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

Algorithm :



Code:

```
from sympy.logic.boolalg import Or, And, Not, Implies, Equivalent
from sympy import symbols

def eliminate_implications(expr):
    """Eliminate implications and equivalences."""
    if isinstance(expr, Implies):
        return Or(Not(eliminate_implications(expr.args[0])),
eliminate_implications(expr.args[1]))

    elif isinstance(expr, Equivalent):
        left = eliminate_implications(expr.args[0])
        right = eliminate_implications(expr.args[1])
        return And(Or(Not(left), right), Or(Not(right), left))

    elif expr.is_Atom:
        return expr
    else:
        return expr.func(*[eliminate_implications(arg) for arg in expr.args])

def push_negations(expr):
    """Push negations inward using De Morgan's laws."""
    if expr.is_Not:
        arg = expr.args[0]

        if isinstance(arg, And):
            return Or(*[push_negations(Not(sub_arg)) for sub_arg in arg.args])
        elif isinstance(arg, Or):
            return And(*[push_negations(Not(sub_arg)) for sub_arg in arg.args])

        elif isinstance(arg, Not):
            return push_negations(arg.args[0])
        else:
            return Not(push_negations(arg))
    elif expr.is_Atom:

        return expr
    else:
        return expr.func(*[push_negations(arg) for arg in expr.args])

def distribute_ands(expr):
    """Distribute AND over OR to obtain CNF."""
    if isinstance(expr, Or):
        and_args = [arg for arg in expr.args if isinstance(arg, And)]

        if and_args:
            first_and = and_args[0]
            rest = [arg for arg in expr.args if arg != first_and]
            return And(*[distribute_ands(Or(arg, *rest)) for arg in
first_and.args])

    elif isinstance(expr, And) or expr.is_Atom or expr.is_Not:
        return expr
    return expr.func(*[distribute_ands(arg) for arg in expr.args])
```

```python
def to_cnf(expr):
    """Convert the given logical expression to CNF."""
    expr = eliminate_implications(expr)
    expr = push_negations(expr)
    expr = distribute_ands(expr)
    return expr

# Example usage:
A, B, C = symbols('A B C')
fol_expr = Implies(A, Or(B, Not(C)))  # Example FOL expression
cnf_expr = to_cnf(fol_expr)

print("FOL expression:", fol_expr)
print("CNF expression:", cnf_expr)
```

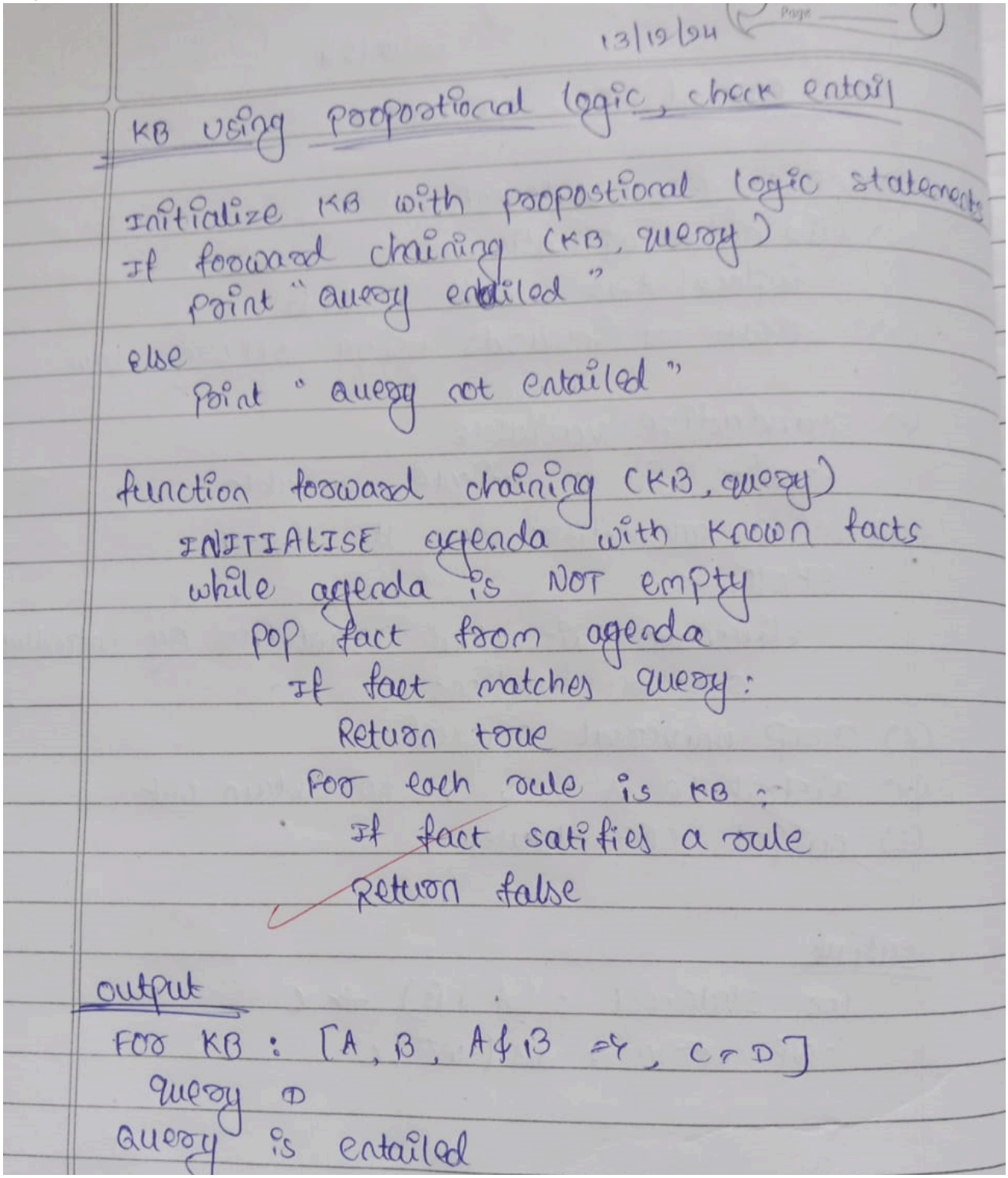
Output:

```
FOL expression: Implies(A, B | ~C)
CNF expression: B | ~A | ~C
```

Program 11

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

Algorithm :



13/12/24

KB using propositional logic, check entail

Initialize KB with propositional logic statements
If forward chaining (KB, query)
  print "query entailed"
else
  print "query not entailed"

function forward chaining (KB, query)
  INITIALISE agenda with known facts
  while agenda is NOT empty
    pop fact from agenda
      If fact matches query:
        Return true
      For each rule in KB:
        If fact satisfies a rule
        Return false

output

FOR KB : [A, B, A & B => C, C => D]
  query D
query is entailed

Code:

```
from itertools import product

# Define a function to evaluate a propositional expression
def evaluate(expr, model):
    """
    Evaluates the given expression based on the values in the model.
    """
    for var, val in model.items():
        expr = expr.replace(var, str(val))
    return eval(expr)

# Define the truth-table enumeration algorithm
def truth_table_entails(KB, query, symbols):
    """
    Checks if KB entails query using truth-table enumeration.
    KB: list of propositional expressions (strings)
    query: propositional expression (string)
    symbols: list of symbols (propositions) in the KB and query
    """
    # Generate all possible truth assignments
    assignments = list(product([False, True], repeat=len(symbols)))

    entailing_models = []

    # Iterate over each assignment to check entailment
    for assignment in assignments:
        model = dict(zip(symbols, assignment))

        # Check if KB is true in this model
        KB_is_true = all(evaluate(expr, model) for expr in KB)

        # If KB is true, check if query is also true
        if KB_is_true:
            query_is_true = evaluate(query, model)
            if query_is_true:
                entailing_models.append(model)  # Store the model
            else:
                return False, []
               # Found a model where KB is true but query is false

    return True, entailing_models  # KB entails query if no counterexample was
found

# Get input from the user
symbols = input("Enter the propositions (symbols) separated by spaces: ").split()
KB = []
n = int(input("Enter the number of statements in the knowledge base: "))

for i in range(n):
    expr = input(f"Enter statement {i + 1} in the knowledge base: ")
    KB.append(expr)

query = input("Enter the query: ")

# Check entailment
result, models = truth_table_entails(KB, query, symbols)
if truth_table_entails(KB, query, symbols):
```

```
    print("KB entails the query.")
    print("Models where KB entails query:")
    for model in models:
        print(model)
else:
    print("KB does not entail the query.")
```

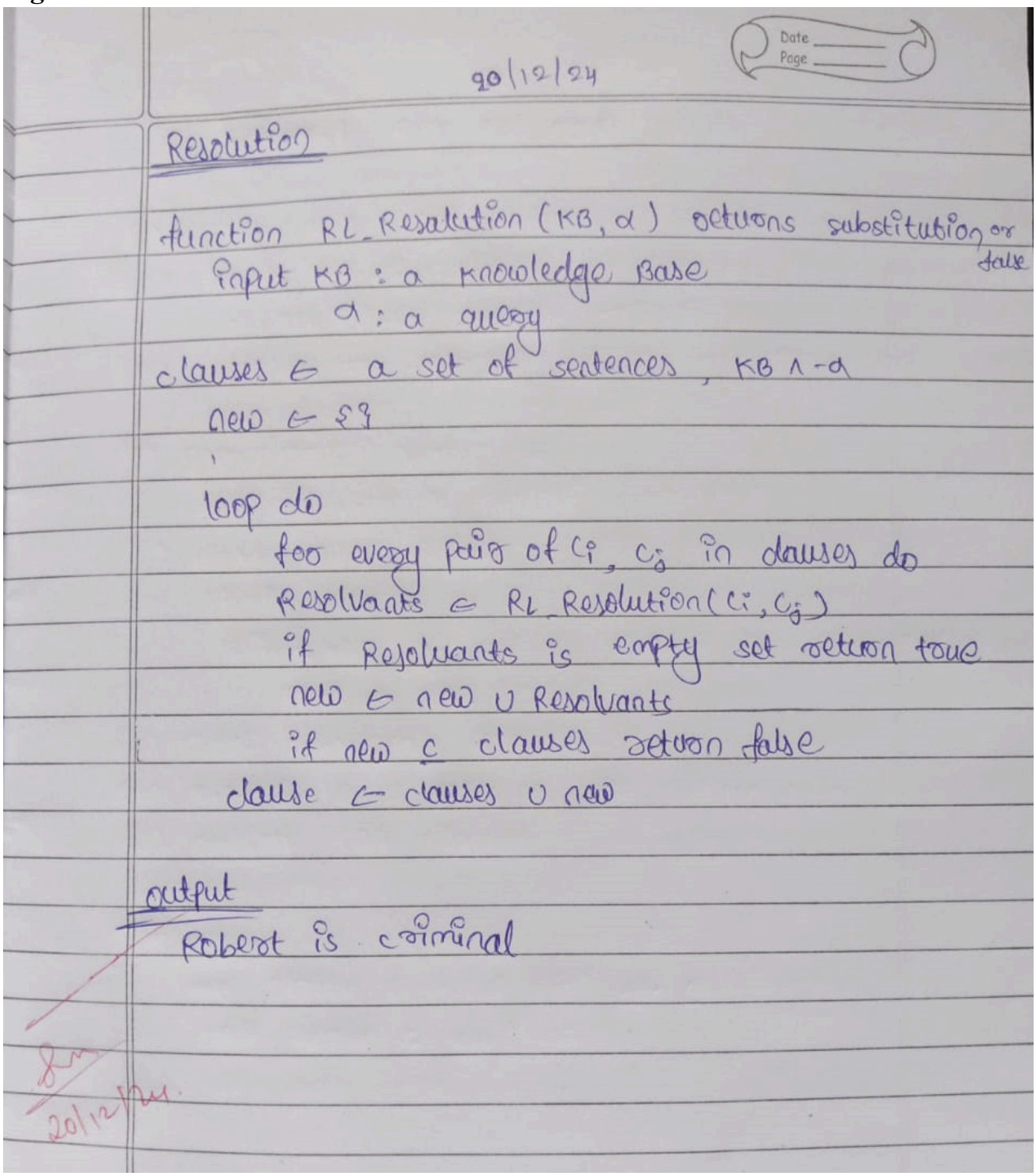
Output:

```
Enter the propositions (symbols) separated by spaces: A B C
Enter the number of statements in the knowledge base: 2
Enter statement 1 in the knowledge base: (A or C)
Enter statement 2 in the knowledge base: (B or not C)
Enter the query: A or B
KB entails the query.
Models where KB entails query:
{'A': False, 'B': True, 'C': True}
{'A': True, 'B': False, 'C': False}
{'A': True, 'B': True, 'C': False}
{'A': True, 'B': True, 'C': True}
```

Program 12

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

Algorithm :



20/12/24

Resolution

function RL_Resolution (KB, α) returns substitution or false

input KB : a knowledge base

α : a query

clauses ← a set of sentences , KB ∧ ¬α

new ← {}

loop do

for every pair of C_i, C_j in clauses do

Resolvants ← RL_Resolution(C_i, C_j)

if Resolvants is empty set return true

new ← new ∪ Resolvants

if new ⊆ clauses return false

clause ← clauses ∪ new

output

Robert is criminal

20/12/24

Code:

```
# Step 1: Helper function to parse user inputs into clauses (with '!' for
negation)
def parse_clause(clause_input):
    """
    Parses a user input string into a tuple of literals for the clause.
    Replaces '!' with '¬' for negation handling.
    Example: "!Food(x), Likes(John, x)" -> ("¬Food(x)", "Likes(John, x)")
    """
    return tuple(literal.strip().replace("!", "¬") for literal in
clause_input.split(","))

# Step 2: Collect knowledge base (KB) from user
def get_knowledge_base():
    print("Enter the premises of the knowledge base, one by one.")
    print("Use ',' to separate literals in a clause. Use '!' for negation.")
    print("Example: !Food(x), Likes(John, x)")
    print("Type 'done' when finished.")

    kb = []
    while True:
        clause_input = input("Enter a clause (or 'done' to finish): ").strip()
        if clause_input.lower() == "done":
            break
        kb.append(parse_clause(clause_input))

    return kb

# Step 3: Add negated conclusion
def get_negated_conclusion():
    print("\nEnter the conclusion to be proved.")
    print("It will be negated automatically for proof by contradiction.")
    conclusion = input("Enter the conclusion (e.g., Likes(John, Peanuts)):
").strip()
    negated = f"!{conclusion}" if not conclusion.startswith("!") else
conclusion[1:]
    return (negated.replace("!", "¬"),)  # Convert '!' to '¬' for consistency

# Step 4: Resolution algorithm
def resolve(clause1, clause2):
    """
    Resolves two clauses and returns the resolvent (new clause).
    If no resolvable literals exist, returns an empty set.
    """
    resolved = set()
    for literal in clause1:
        if "¬" + literal in clause2:
            temp1 = set(clause1)
            temp2 = set(clause2)
            temp1.remove(literal)
            temp2.remove("¬" + literal)
            resolved = temp1.union(temp2)
        elif literal.startswith("¬") and literal[1:] in clause2:
            temp1 = set(clause1)
            temp2 = set(clause2)
            temp1.remove(literal)
            temp2.remove(literal[1:])
            resolved = temp1.union(temp2)
```

```
    return tuple(resolved)

def resolution(kb):
    """
    Runs the resolution algorithm on the knowledge base (KB).
    Returns True if an empty clause is derived (proving the conclusion),
    or False if resolution fails.
    """
    clauses = set(kb)
    new = set()

    while True:
        # Generate all pairs of clauses
        pairs = [(ci, cj) for ci in clauses for cj in clauses if ci != cj]

        for (ci, cj) in pairs:
            resolvent = resolve(ci, cj)
            if not resolvent:  # Empty clause found
                return True
            new.add(resolvent)

        if new.issubset(clauses):  # No new clauses
            return False
        clauses = clauses.union(new)

# Step 5: Main execution
if __name__ == "__main__":
    print("=== Resolution Proof System ===")
    print("Provide the knowledge base and conclusion to prove.")

    # Collect user inputs
    kb = get_knowledge_base()
    negated_conclusion = get_negated_conclusion()
    kb.append(negated_conclusion)

    # Show the knowledge base
    print("\nKnowledge Base (KB):")
    for clause in kb:
        print("  ", " ∨ ".join(clause))  # Join literals with OR for readability

    # Perform resolution
    print("\nStarting resolution process...")
    result = resolution(kb)
    if result:
        print("\nProof complete: The conclusion is TRUE.")
    else:
        print("\nResolution failed: The conclusion could not be proved.")
```

Output:

```
=== Resolution Proof System ===
Provide the knowledge base and conclusion to prove.
Enter the premises of the knowledge base, one by one.
Use ',' to separate literals in a clause. Use '!' for negation.
Example: !Food(x), Likes(John, x)
Type 'done' when finished.
Enter a clause (or 'done' to finish): !Food(x), Likes(John,x)
Enter a clause (or 'done' to finish): Food(Apple)
Enter a clause (or 'done' to finish): Food(Vegetables)
Enter a clause (or 'done' to finish): !Eats(x,y), !Killed(x), Food(y)
Enter a clause (or 'done' to finish): Eats(Anil, Peanuts)
Enter a clause (or 'done' to finish): !Killed(Anil)
Enter a clause (or 'done' to finish): done

Enter the conclusion to be proved.
It will be negated automatically for proof by contradiction.
Enter the conclusion (e.g., Likes(John, Peanuts)): Likes(John, Peanuts)

Knowledge Base (KB):
   ¬Food(x) ∨ Likes(John ∨ x)
   Food(Apple)
   Food(Vegetables)
   ¬Eats(x ∨ y) ∨ ¬Killed(x) ∨ Food(y)
   Eats(Anil ∨ Peanuts)
   ¬Killed(Anil)
   ¬Likes(John, Peanuts)

Starting resolution process...

Proof complete: The conclusion is TRUE.
```