VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB REPORT

Artificial Intelligence (23CS5PCAIN)

Submitted by

Madhu Sarika (1BM22CS140)

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

Academic Year 2024 - 25 (odd)

B.M.S. College of Engineering

Bull Temple Road, Bangalore 560019

(Affiliated To Visvesvaraya Technological University, Belgaum)

Department of Computer Science and Engineering

CERTIFICATE

This is to certify that the Lab work entitled "Artificial Intelligence (23CS5PCAIN)" carried out by **MADHU SARIKA (1BM22CS140),** who is bonafide student of **B.M.S. College of Engineering.** It is in partial fulfilment for the award of **Bachelor of Engineering in Computer Science and Engineering** of the Visvesvaraya Technological University, Belgaum. The Laboratory 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	Dr. Kavitha Sooda
Assistant Professor	Professor & HOD
Department of CSE, BMSCE	Department of CSE, BMSCE

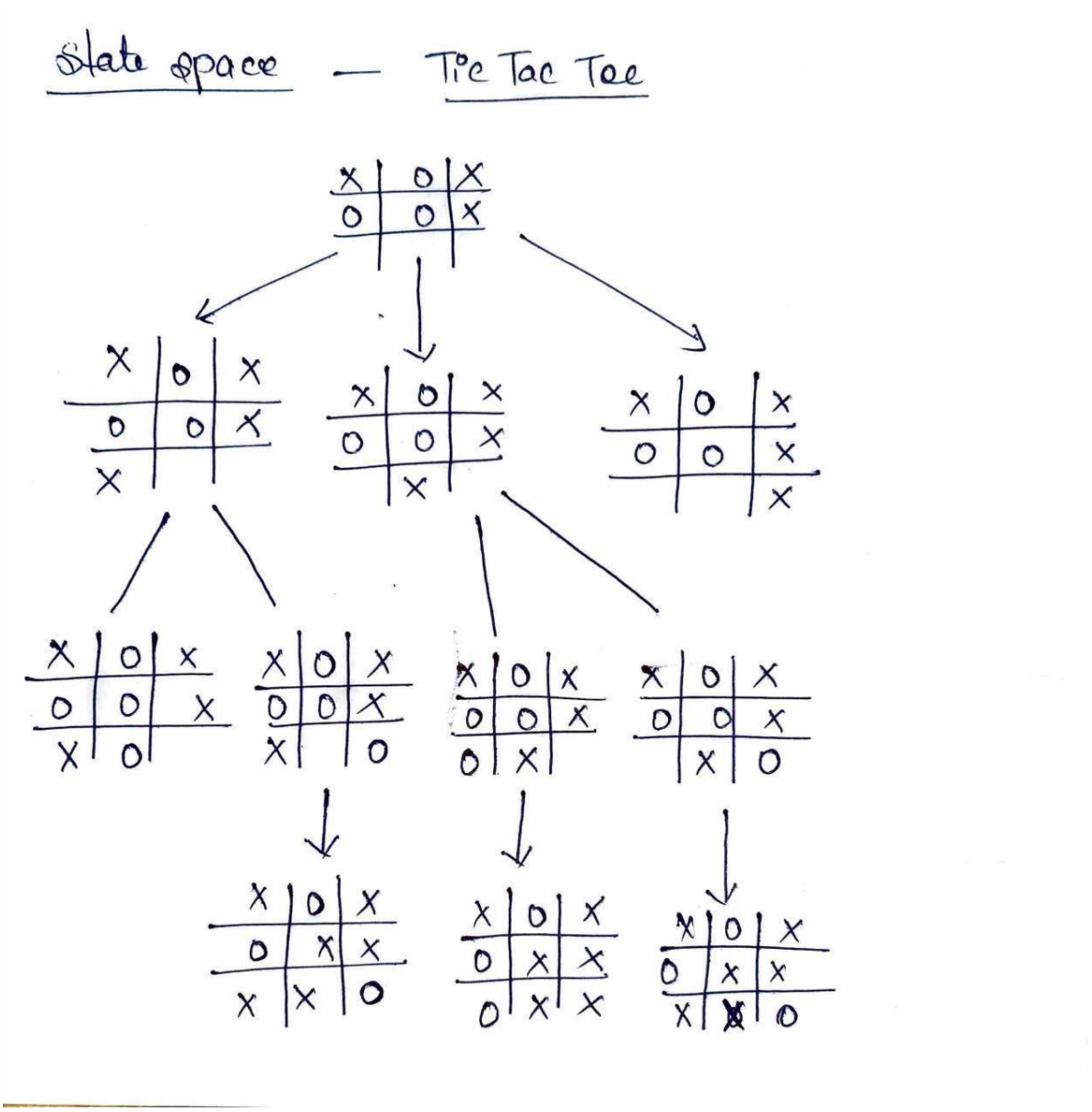
INDEX

Github Link:

https://github.com/madhupandeyy/AI-Lab

LAB PROGRAM 1

Implement Tic Tac Toe game.



Lab - 2 Tictac Toe 1|10|24

Algorithm:

1. Intialize game board – Create a 3×3 board filled with empty spaces.
2. Print the board positions and the corresponding index numbers (0-8) for user input.
3. Set human player as "x" and the computer as "0".
4. Game loop:

   Repeat until the game ends (win or draw)

   Print current board state.

   //check current player

   if the current player is "x" (human), prompt the user to input their move.

   For validating i/p, ensure that user has entered b/w 0 and 8 and also check that the selected cell is not already occupied, if invalid prompt again.

   if the current player is "0" (computer) generate a list of available moves and implement the strategy to choose the best move, win move, block move or random.

5. Place the current player's mark either 0 or x in the selected cell.
6. Call check_winner(board) to check for the winner.
7. Call isboard_full(board) to check for the draw and then exit the loop.
8. Then player can switch from x to 0.

Code:

```
import random

def print_board(board):
    for row in board:
        print(" | ".join(row))
        print("-" * 9)

def check_winner(board):
    for row in board:
        if row[0] == row[1] == row[2] != " ":
            return row[0]

    for col in range(3):
        if board[0][col] == board[1][col] == board[2][col] != " ":
            return board[0][col]

    if board[0][0] == board[1][1] == board[2][2] != " ":
        return board[0][0]
    if board[0][2] == board[1][1] == board[2][0] != " ":
        return board[0][2]

    return None

def is_board_full(board):
    return all(cell != " " for row in board for cell in row)

def get_available_moves(board):
    return [(i, j) for i in range(3) for j in range(3) if board[i][j] == " "]
```

```
def tic_tac_toe():
    board = [[" " for _ in range(3)] for _ in range(3)]
    current_player = "X" # Human player
    computer_player = "O"

    print("Welcome to Tic-Tac-Toe!")
    print("Board positions:")
    print(" 0 | 1 | 2")
    print("...........")
    print(" 3 | 4 | 5")
    print("...........")
    print(" 6 | 7 | 8")

    while True:
        print_board(board)

        if current_player == "X":
            move = input(f"Player {current_player}, enter your move (0-8): ")
            try:
                move = int(move)
                row, col = divmod(move, 3)
                if board[row][col] != " ":
                    print("Invalid move. Cell already occupied. Try again.")
                    continue
            except (ValueError, IndexError):
                print("Invalid input. Please enter a number between 0 and 8.")
                continue
        else:
            # Computer's turn
            available_moves = get_available_moves(board)
```

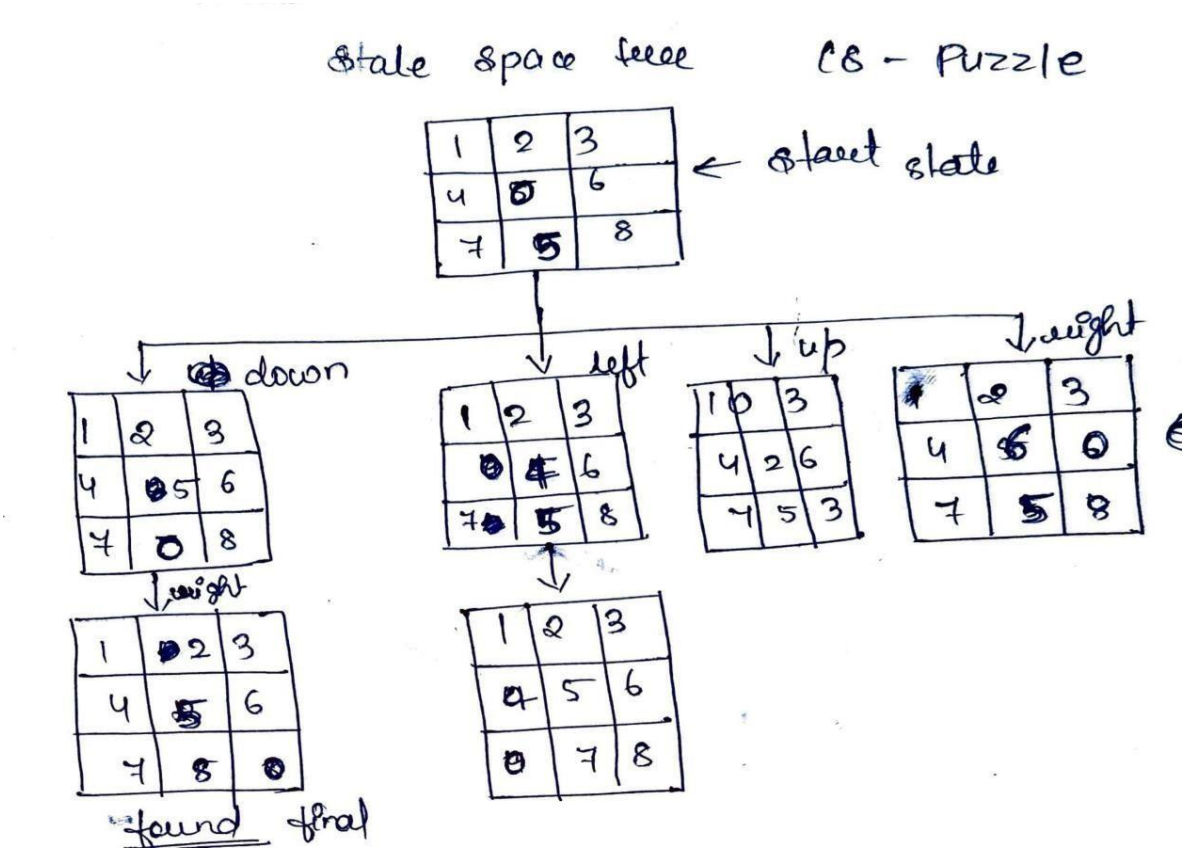
```
            row, col = random.choice(available_moves)
            print(f"Computer ({computer_player}) chooses: {row * 3 + col}")

        board[row][col] = current_player
        winner = check_winner(board)

        if winner:
            print_board(board)
            print(f"Player {winner} wins!")
            break

        if is_board_full(board):
            print_board(board)
            print("It's a draw!")
            break

        # Switch players
        current_player = computer_player if current_player == "X" else "X"

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

Output:

```
Welcome to Tic-Tac-Toe!
Board positions:
 0 | 1 | 2
-----------
 3 | 4 | 5
-----------
 6 | 7 | 8
  |   |
---------
  |   |
---------
  |   |
---------
Player X, enter your move (0-8): 4
  |   |
---------
  | X |
---------
  |   |
---------
Computer (O) chooses: 8
  |   |
---------
  | X |
---------
  |   | O
---------
Player X, enter your move (0-8): 5
  |   |
---------
  | X | X
---------
  |   | O
---------
Computer (O) chooses: 2
  |   | O
---------
  | X | X
---------
  |   | O
---------
Player X, enter your move (0-8): 3
  |   | O
---------
X | X | X
---------
  |   | O
---------
Player X wins!
```

LAB PROGRAM 2

Solve 8 – Puzzle problems.



LAB-2

8 Puzzle Problem

```
Import numpy as np

def bfs(src, target):
    queue = [(src, None)] # state and last move
    visited = set()
    state_count = 0  # Initialize state count

    while queue:
        state, last_move = queue.pop(0)
        state_tuple = tuple(state) # Convert state to
                                   tuple for set
        if state_tuple not in visited: operations
            visited.add(state_tuple)
            state_count += 1 # Increment the state
                               count

            print_board(state)
            if last_move:
                print(f"Current move: {last_move}\n")

            if state == target:
                print("Goal state achieved!")
                break

            for move, direction in possible_moves
                                   (state):
                if tuple(move) not in visited:
                    queue.append((move, direction))

print(f"Total unique states explored: {state_
                                   count}")
```

```
def possible_moves(state):
    b = state.index(0)
    directions = []
    if b not in [0,1,2]: directions.append('u')
    if b not in [6,7,8]: directions.append('d')
    if b not in [0,3,6]: directions.append('l')
    if b not in [2,5,8]: directions.append('r')
    return [(gen(state, d, b), d) for d in directions]

def gen(state, direction, b):
    temp = state.copy()
    if direction == 'u': temp[b], temp[b-3] = temp[b-3], temp[b]
    if direction == 'd': temp[b], temp[b+3] = temp[b+3], temp[b]
    if direction == 'l': temp[b], temp[b-1] = temp[b-1], temp[b]
    if direction == 'r': temp[b], temp[b+1] = temp[b+1], temp[b]

def print_board(state):
    board = np.array(state).reshape(3,3)
    print(board)

# Intial and target configuration
src = [1,2,3,0,4,5,6,7,8]
target = [1,2,3,4,5,6,7,8,0]

# Run bfs to solve the puzzle
bfs(src, target)
```

Code:

```
import numpy as np



def bfs(src, target):
    queue = [(src, None)] # State and last move
    visited = set()
    state_count = 0  # Initialize state count



    while queue:
        state, last_move = queue.pop(0)
        state_tuple = tuple(state)  # Convert state to tuple for set operations

        if state_tuple not in visited:
            visited.add(state_tuple)
            state_count += 1 # Increment the state count



            print_board(state)
            if last_move:
                print(f"Current move: {last_move}\n")



            if state == target:
                print("Goal state achieved!")
                break



            for move, direction in possible_moves(state):
```

```
            if tuple(move) not in visited:
                queue.append((move, direction))

    print(f"Total unique states explored: {state_count}")

def possible_moves(state):
    b = state.index(0)
    directions = []

    if b not in [0, 1, 2]: directions.append('u')
    if b not in [6, 7, 8]: directions.append('d')
    if b not in [0, 3, 6]: directions.append('l')
    if b not in [2, 5, 8]: directions.append('r')

    return [(gen(state, d, b), d) for d in directions]

def gen(state, direction, b):
    temp = state.copy()
    if direction == 'u': temp[b], temp[b - 3] = temp[b - 3], temp[b]
    if direction == 'd': temp[b], temp[b + 3] = temp[b + 3], temp[b]
    if direction == 'l': temp[b], temp[b - 1] = temp[b - 1], temp[b]
    if direction == 'r': temp[b], temp[b + 1] = temp[b + 1], temp[b]
    return temp

def print_board(state):
```

```
    board = np.array(state).reshape(3, 3)
    print(board)


# Initial configuration and target configuration
src = [1, 2, 3, 0, 4, 6, 7, 5, 8]
target = [1, 2, 3, 4, 5, 6, 7, 8, 0]


# Run BFS to solve the puzzle
bfs(src, target)
```

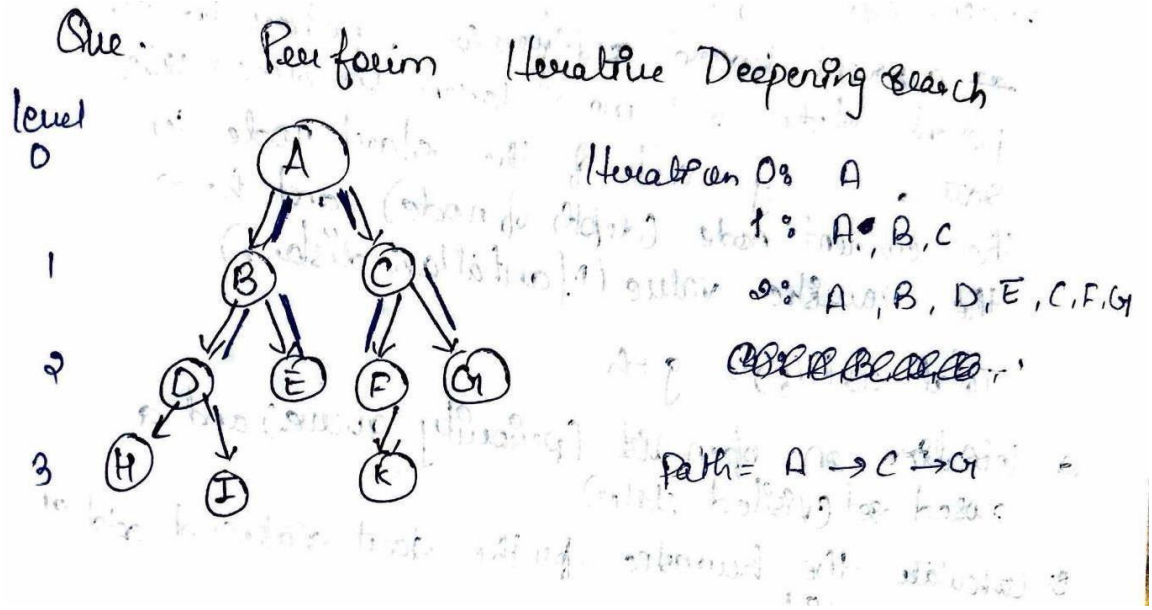
Output:

```
[[1 2 3]
 [0 4 6]
 [7 5 8]]
[[0 2 3]
 [1 4 6]
 [7 5 8]]
Current move: u

[[1 2 3]
 [7 4 6]
 [0 5 8]]
Current move: d

[[1 2 3]
 [4 0 6]
 [7 5 8]]
Current move: r

[[2 0 3]
 [1 4 6]
 [7 5 8]]
Current move: r

[[1 2 3]
 [7 4 6]
 [5 0 8]]
Current move: r

[[1 0 3]
 [4 2 6]
 [7 5 8]]
Current move: u

[[1 2 3]
 [4 5 6]
 [7 0 8]]
Current move: d

[[1 2 3]
 [4 6 0]
 [7 5 8]]
Current move: r
```

```
[[2 4 3]
 [1 0 6]
 [7 5 8]]
Current move: d

[[2 3 0]
 [1 4 6]
 [7 5 8]]
Current move: r

[[1 2 3]
 [7 0 6]
 [5 4 8]]
Current move: u

[[1 2 3]
 [7 4 6]
 [5 8 0]]
Current move: r

[[0 1 3]
 [4 2 6]
 [7 5 8]]
Current move: l

[[1 3 0]
 [4 2 6]
 [7 5 8]]
Current move: r

[[1 2 3]
 [4 5 6]
 [0 7 8]]
Current move: l

[[1 2 3]
 [4 5 6]
 [7 8 0]]
Current move: r

Goal state achieved!
Total unique states explored: 17
```

LAB PROGRAM 3

Implement Iterative deepening search algorithm.

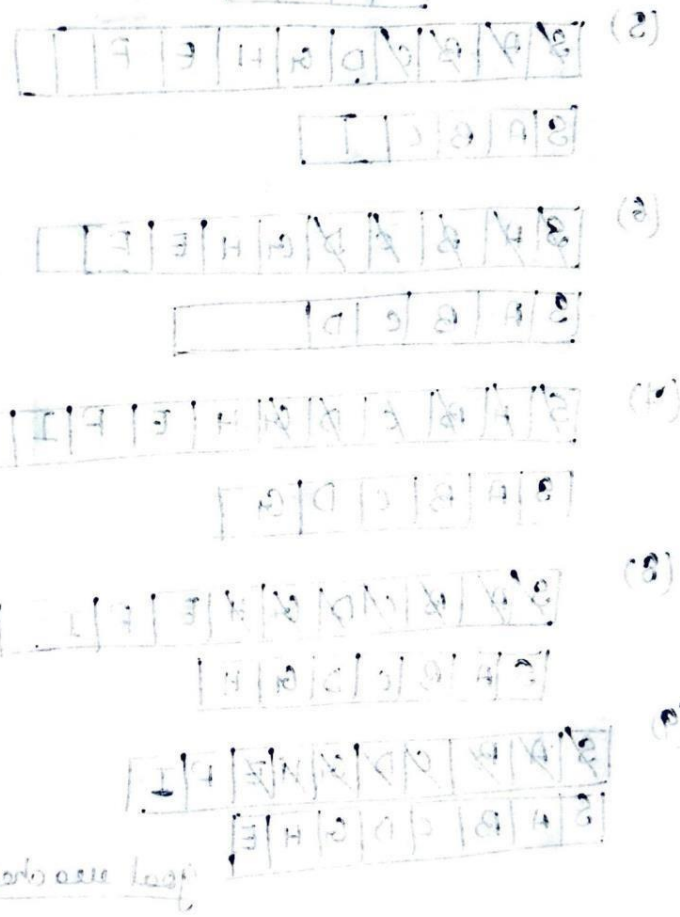


Iterative Deepening DFS:

```
graph = {
    'A' : ['B','C'],
    'B' : ['D','E'],
    'C' : ['G'],
    'D' : [],
    'E' : ['F'],
    'G' : [],
    'F' : []
}

def DFS(curNode, dest, graph, maxDepth, curList):
    print("Checking for destination", curNode)
    curList.append(curNode)
    if curNode == dest:
        return True
    if maxDepth <= 0:
        return False
    for node in graph[curNode]:
        if DFS(node, dest, graph, maxDepth-1, curList):
            return True
    curList.pop()  # Back track if no path found at this depth
    return False

def IterativeDFS(curNode, dest, graph, maxDepth):
    for i in range(maxDepth):
        print(f"\n --- Iteration with depth level {i} ---")
        curList = []
```

if DFS(curNode, dest, graph, s, curPath):
    print("Yes, path exists")
    print(curPath)
    return True
print(f"Completed level {i}, no path found at this depth.\n")
print("Path not available.")
return False

iterativeDDFS('A', 'E', graph, 4)



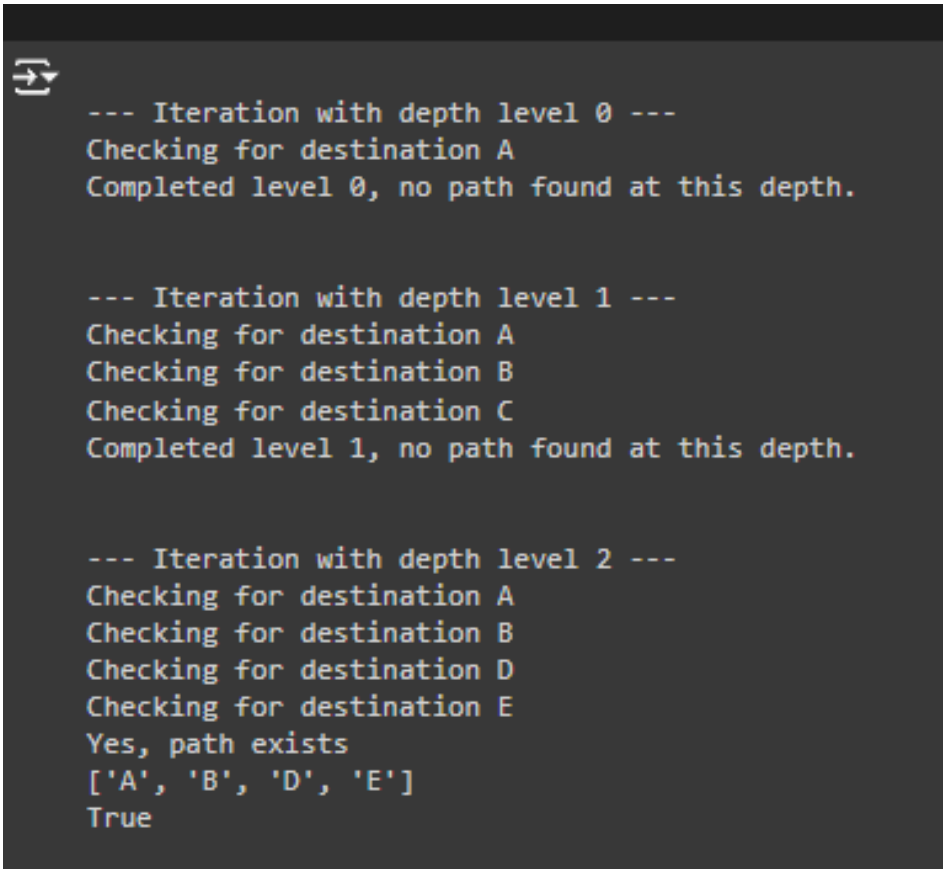
```
def possible_moves(state):
    b = state.index(0)
    directions = []
    if b not in [0,1,2]: directions.append('u')
    if b not in [6,7,8]: directions.append('d')
    if b not in [0,3,6]: directions.append('l')
    if b not in [2,5,8]: directions.append('r')
    return [(gen(state, d, b), d) for d in directions]

def gen(state, direction, b):
    temp = state.copy()
    if direction=='u': temp[b], temp[b-3] = temp[b-3], temp[b]
    if direction=='d': temp[b], temp[b+3] = temp[b+3], temp[b]
    if direction=='l': temp[b], temp[b-1] = temp[b-1], temp[b]
    if direction=='r': temp[b], temp[b+1] = temp[b+1], temp[b]

def print_board(state):
    board = np.array(state).reshape(3,3)
    print(board)

#Intial and target configuration
src = [1,2,3,0,4,5,6,7,8]
target = [1,2,3,4,5,6,7,8,0]

#Run bfs to solve the puzzle
bfs(src, target)
```

Code:

```
graph = {
    'A': ['B', 'C'],
    'B': ['D','E'],
    "C": ['G'],
    'D': [],
    'E': ['F'],
    'G': [],
    'F':[]
}

def DFS(currentNode, destination, graph, maxDepth, curList):
    print("Checking for destination", currentNode)
    curList.append(currentNode)
    if currentNode == destination:
        return True
    if maxDepth <= 0:
        return False
    for node in graph[currentNode]:
        if DFS(node, destination, graph, maxDepth - 1, curList):
            return True
    curList.pop() # Backtrack if no path is found at this depth
    return False

def iterativeDDFS(currentNode, destination, graph, maxDepth):
    for i in range(maxDepth):
        print(f"\n--- Iteration with depth level {i} ---")
        curList = []
        if DFS(currentNode, destination, graph, i, curList):
            print("Yes, path exists")
```

```
            print(curList)
            return True
        print(f"Completed level {i}, no path found at this depth.\n")
    print("Path is not available")
    return False

# Calling the function
iterativeDDFS('A', 'E', graph, 4)
```

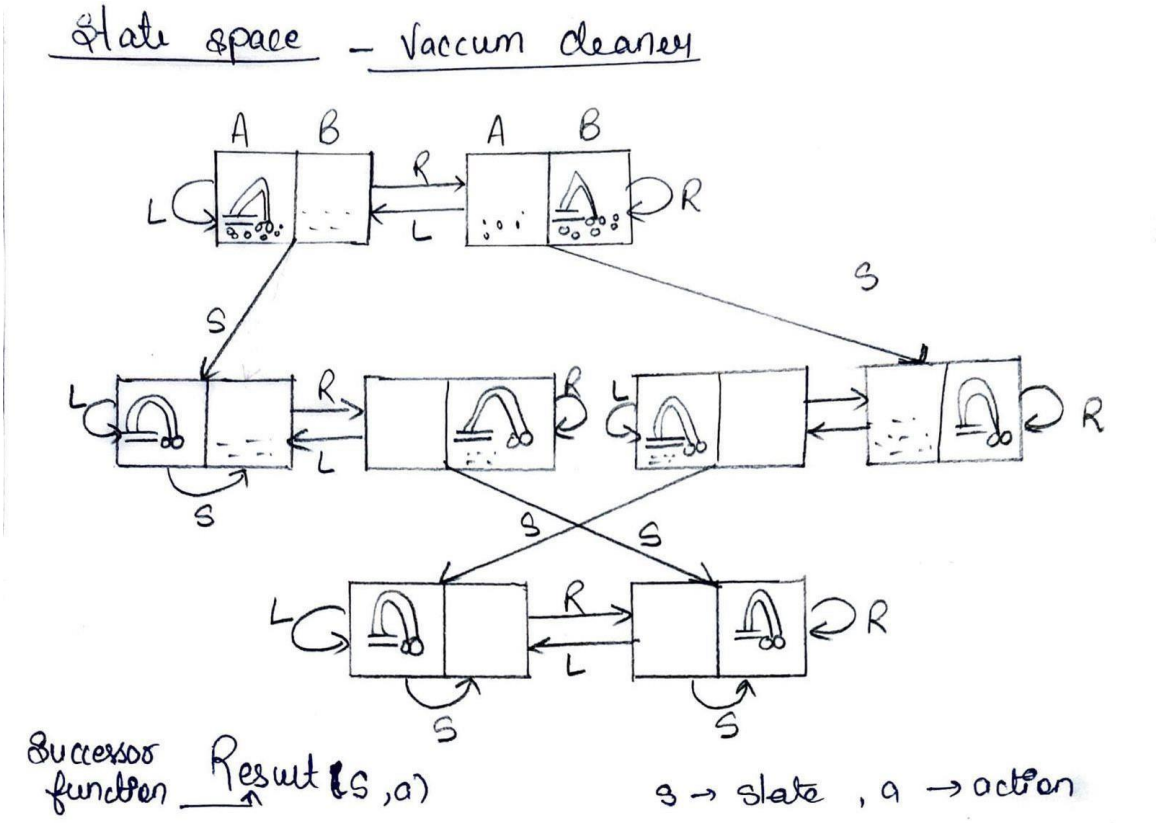
Output:



```
--- Iteration with depth level 0 ---
Checking for destination A
Completed level 0, no path found at this depth.


--- Iteration with depth level 1 ---
Checking for destination A
Checking for destination B
Checking for destination C
Completed level 1, no path found at this depth.


--- Iteration with depth level 2 ---
Checking for destination A
Checking for destination B
Checking for destination D
Checking for destination E
Yes, path exists
['A', 'B', 'D', 'E']
True
```

LAB PROGRAM 4

Implement vaccum cleaner agent.



Vaccum cleaner

Pseudocode :

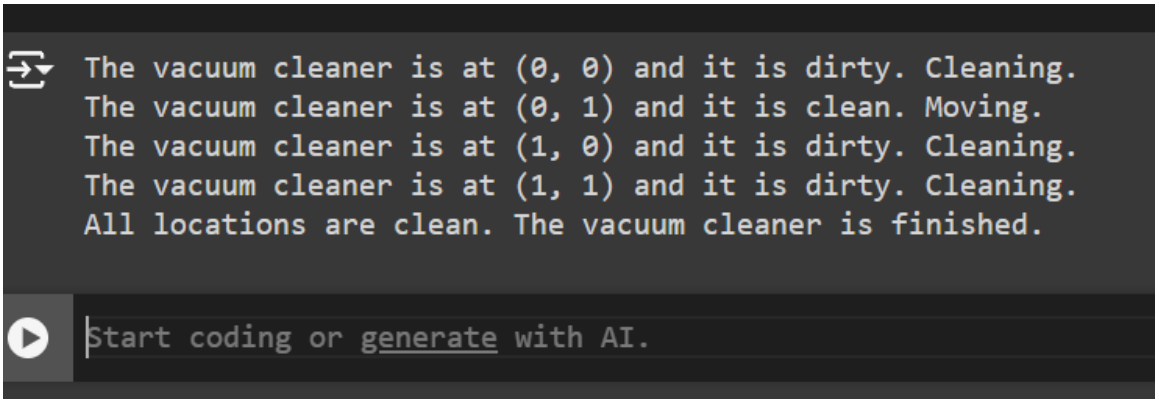
```
def vacuum_cleaner_agent(location, status):
    x, y = location
    if status[x][y] == 'Dirty':
        return f"the vacuum cleaner is at({x},{y}) and it is
        dirty. cleaning."
    else:
        return f"the vacuum cleaner is at ({x},{y} and it is
        clean. Moving."

status = [['Dirty', 'clean'], ['Dirty', 'Dirty']]
location = (0,0)
while True:
    action = vacuum_cleaner_agent(location, status)
    print(action)

    x, y = location
    if status[x][y] == 'Dirty':
        status[x][y] = 'clean'
    if status[0][0] == 'clean' and status[0][1] == 'clean'
    and status[1][0] == 'clean' and status[1][1] == 'clean':
        print("All locations are clean")
        break
    if y < 1:
        location = (x, y+1)
    elif x < 1
        location = (x+1, 0)
```

8a
11/10/24

Code:

```
def vacuum_cleaner_agent(location, status):
    x, y = location
    if status[x][y] == 'Dirty':
        return f"The vacuum cleaner is at ({x}, {y}) and it is dirty. Cleaning."
    else:
        return f"The vacuum cleaner is at ({x}, {y}) and it is clean. Moving."

status = [['Dirty', 'Clean'], ['Dirty', 'Dirty']]
location = (0, 0)

while True:
    action = vacuum_cleaner_agent(location, status)
    print(action)

    x, y = location
    if status[x][y] == 'Dirty':
        status[x][y] = 'Clean'

    if status[0][0] == 'Clean' and status[0][1] == 'Clean' and status[1][0] == 'Clean' and
status[1][1] == 'Clean':
        print("All locations are clean. The vacuum cleaner is finished.")
        break

    if y < 1:
        location = (x, y + 1)
    elif x < 1:
        location = (x + 1, 0)
```

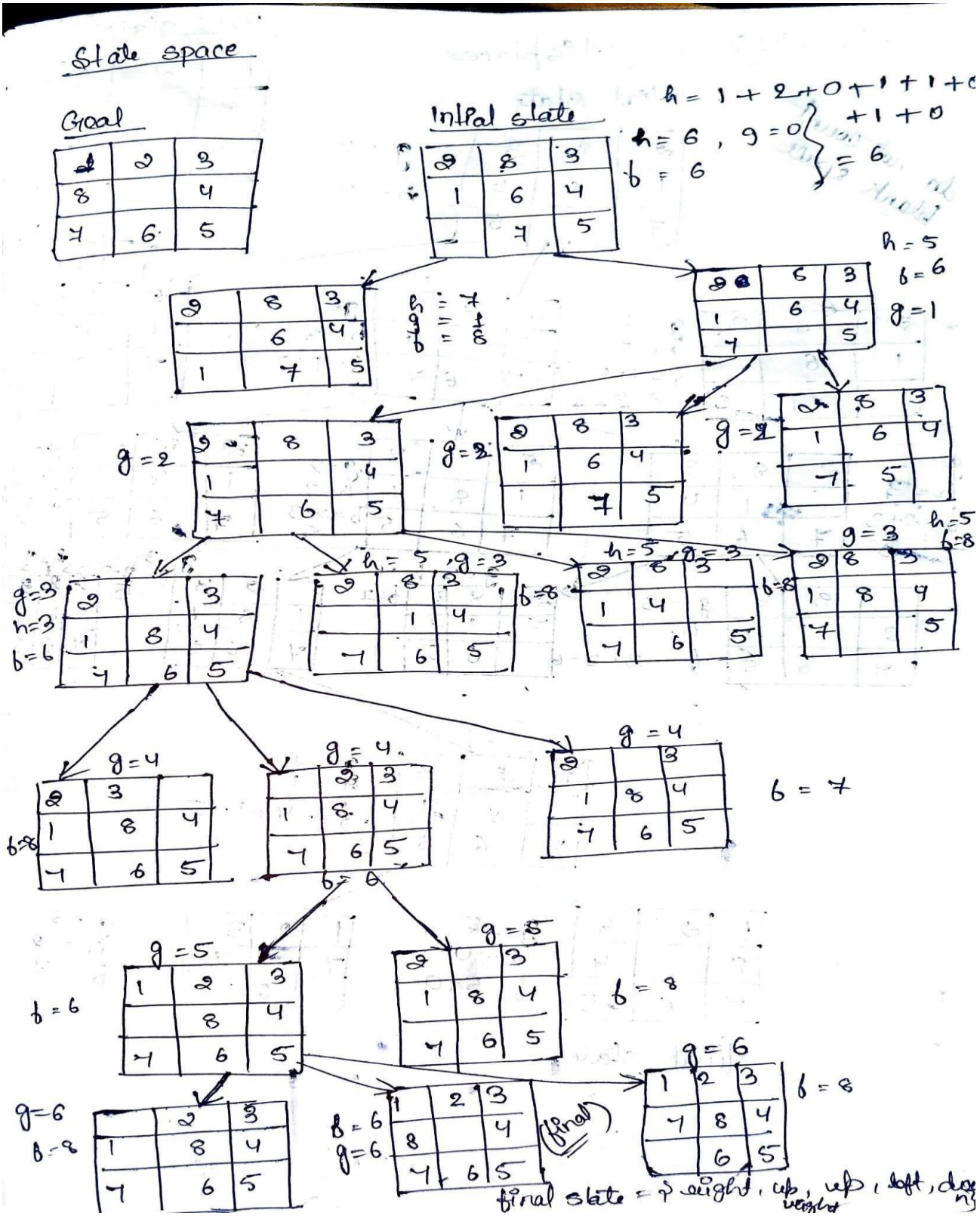
Output:


The vacuum cleaner is at (0, 0) and it is dirty. Cleaning.
The vacuum cleaner is at (0, 1) and it is clean. Moving.
The vacuum cleaner is at (1, 0) and it is dirty. Cleaning.
The vacuum cleaner is at (1, 1) and it is dirty. Cleaning.
All locations are clean. The vacuum cleaner is finished.
Start coding or generate with AI.

LAB PROGRAM 5

Implement A* search algorithm.

Heuristic – Manhattan



15/10/24

Heuristic : Manhattan Algorithm

1. Create a puzzle class to represent the current state of the 8 puzzle. This includes: ~~a~~ current board configuration, pointer to the parent state and move taken to reach this state. g → cost of the start node to the current node (depth of node) and h → the heuristic value (Manhattan distance)

   Total cost (f) = g+h

2. Initialize an open list (priority queue) and a closed set (visited states)
3. Calculate the heuristic for the start state and add it to open list.
4. When the open list is non empty, pop the state with lowest f(n) from the open list.
5. If the current state is the goal, trace the path back to the start and return the moves.
6. Mark the current state as explored and add it to closed set.
7. For each neighbouring state, generate new states and calculate g(n), h(n), and f(n).
8. If the neighbour hasn't been explored add it to the open list.
9. If the goal is found, return the solution path or if the open list has no solution, return that no solution exists.

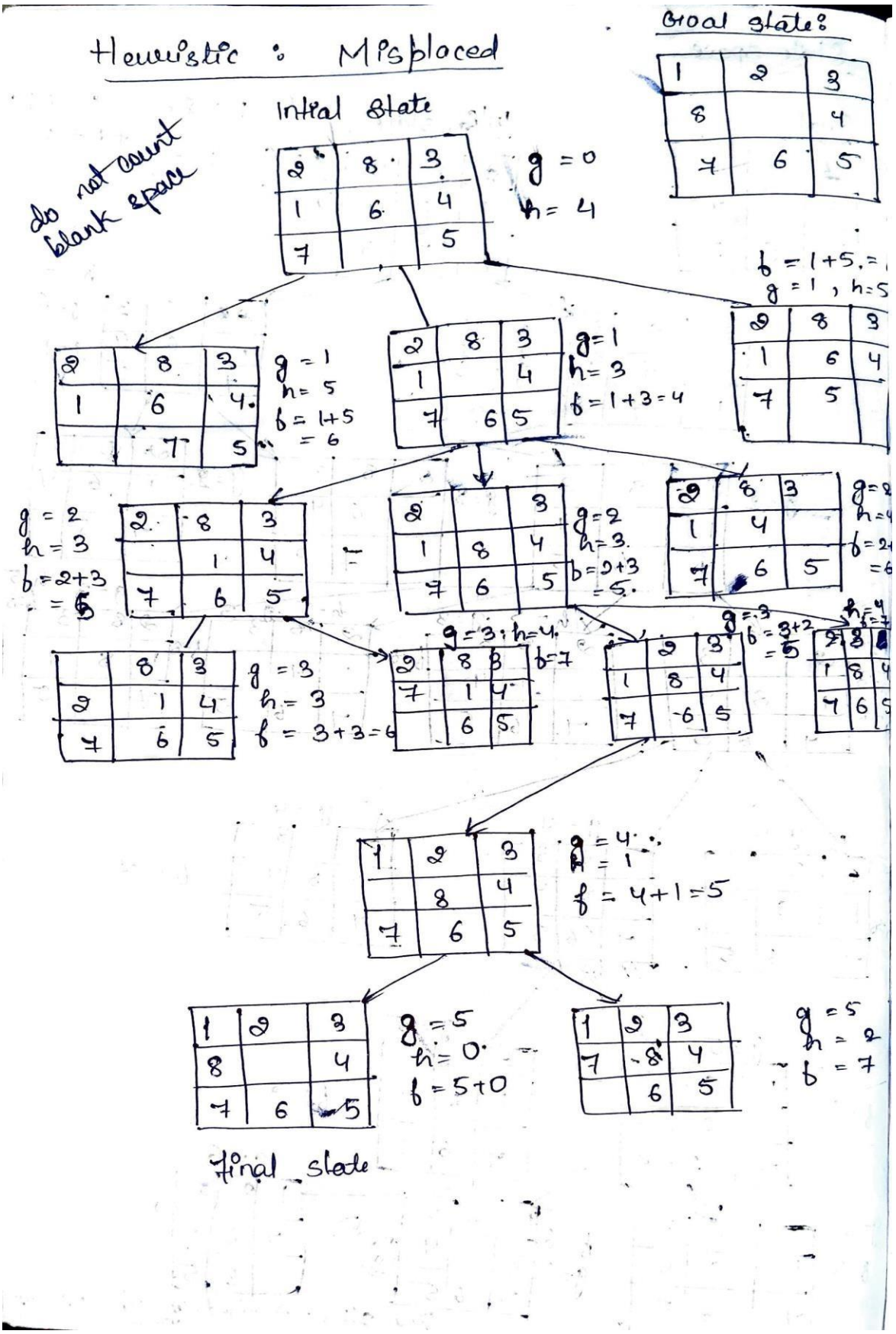
```
import heapq

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

def find_zero(state):
    for i in range(3):
        for j in range(3):
            if state[i][j] == 0:
                return (i, j)

def move(state, direction):
    new_state = [row[:] for row in state]
    i, j = find_zero(state)
    if direction == "up" and i > 0:
        new_state[i][j], new_state[i-1][j] = new_state[i-1][j], new_state[i][j]
    elif direction == "down" and i < 2:
        new_state[i][j], new_state[i+1][j] = new_state[i+1][j], new_state[i][j]
    elif direction == "left" and j > 0:
        new_state[i][j], new_state[i][j-1] = new_state[i][j-1], new_state[i][j]
    elif direction == "right" and j < 2:
        new_state[i][j], new_state[i][j+1] = new_state[i][j+1], new_state[i][j]
    else:
        return None
    return new_state

def is_goal(state):
    return state == goal_state

def manhattan_distance(state):
    distance = 0
    for i in range(3):
        for j in range(3):
            if state[i][j] != 0:
                goal_i, goal_j = divmod(state[i][j] - 1, 3)
                if state[i][j] == 8:
                    goal_i, goal_j = 1, 1
                distance += abs(goal_i - i) + abs(goal_j - j)
    return distance

def a_star(initial_state):
    priority_queue = []
    heapq.heappush(priority_queue, (0, initial_state, [], 0))
    visited = set()

    while priority_queue:
        f, state, path, g = heapq.heappop(priority_queue)
        print("Exploring state in A*:")
        print_state(state)
```

```
        if is_goal(state):
            return path


        visited.add(tuple(map(tuple, state)))

        for direction in ["up", "down", "left", "right"]:
            new_state = move(state, direction)
            if new_state and tuple(map(tuple, new_state)) not in visited:
                h = manhattan_distance(new_state)
                new_g = g + 1
                new_f = new_g + h
                heapq.heappush(priority_queue, (new_f, new_state, path + [direction], new_g))

    return None

def print_state(state):
    for row in state:
        print(row)
    print()

def get_initial_state():
    print("Enter the initial state of the 8-puzzle (0 for empty space):")
    initial_state = []
    for i in range(3):
        row = list(map(int, input(f"Enter row {i+1} (space-separated): ").strip().split()))
        if len(row) != 3:
            raise ValueError("Each row must contain exactly 3 numbers.")
        initial_state.append(row)
    return initial_state

if __name__ == "__main__":
    initial_state = get_initial_state()
    print("Initial State:")
    print_state(initial_state)
    print("Solving using A* search:")
    a_star_solution = a_star(initial_state)

    if a_star_solution:
        print("A* Solution:", a_star_solution)
    else:
        print("No solution found with A*.")
"""
```

Output:

```
Enter the initial state (0 for empty space):
Enter row 1 (space-separated): 2 8 3
Enter row 2 (space-separated): 1 6 4
Enter row 3 (space-separated): 0 7 5
Initial State:
[2, 8, 3]
[1, 6, 4]
[0, 7, 5]

Solving using A* search:
Exploring state in A*:
[2, 8, 3]
[1, 6, 4]
[0, 7, 5]

Exploring state in A*:
[2, 8, 3]
[1, 6, 4]
[7, 0, 5]

Exploring state in A*:
[2, 8, 3]
[1, 6, 4]
[7, 5, 0]

Exploring state in A*:
[2, 8, 3]
[0, 6, 4]
[1, 7, 5]

Exploring state in A*:
[2, 8, 3]
[1, 0, 4]
[7, 6, 5]

Exploring state in A*:
[2, 0, 3]
[1, 8, 4]
[7, 6, 5]

Exploring state in A*:
[0, 2, 3]
[1, 8, 4]
[7, 6, 5]

Exploring state in A*:
[1, 2, 3]
[0, 8, 4]
[7, 6, 5]

Exploring state in A*:
[2, 8, 3]
[1, 4, 0]
[7, 6, 5]

Exploring state in A*:
[2, 8, 3]
[1, 4, 5]
[7, 6, 0]

Exploring state in A*:
[2, 8, 3]
[1, 4, 5]
[7, 0, 6]

Exploring state in A*:
[2, 8, 3]
[1, 6, 0]
[7, 5, 4]

Exploring state in A*:
[2, 8, 3]
[1, 0, 6]
[7, 5, 4]

Exploring state in A*:
[2, 0, 3]
[1, 8, 6]
[7, 5, 4]

Exploring state in A*:
[0, 2, 3]
[1, 8, 6]
[7, 5, 4]

Exploring state in A*:
[1, 2, 3]
[0, 8, 6]
[7, 5, 4]

Exploring state in A*:
[2, 8, 3]
[1, 5, 6]
[7, 0, 4]

Exploring state in A*:
[2, 8, 3]
[1, 5, 6]
[7, 4, 0]

Exploring state in A*:
[0, 8, 3]
[2, 6, 4]
[1, 7, 5]

Exploring state in A*:
[1, 2, 3]
[7, 8, 4]
[0, 6, 5]

Exploring state in A*:
[1, 2, 3]
[7, 8, 6]
[0, 5, 4]

Exploring state in A*:
[1, 2, 3]
[8, 0, 4]
[7, 6, 5]

A* Solution: ['right', 'up', 'up', 'left', 'down', 'right']
"""
```

Heuristic – Mismatched Tiles



Algorithm

1) Create the start node with intial state and set its cost = 0.
2) Create an empty priority queue called frontier and add the start node to it.
3) Create an empty set called explored set to store visited states

while frontier is not empty,

4) pop the node with the lowest cost from the frontier, (this is the current node).
5) If the current node's state matches the goal state, return the path to this node.
6) Add the current node's state to the explored set.

for each neighbour of currentnode:

7) Calculate the depth g as the current node's depth + 1.
8) Calculate the heuristic h by counting the number of misplaced in the neighbour's state.
9) Calculate the total cost f = g + h.
10) If the neighbour's state is not in explored
   (i) Set the neighbor's cost to f and
   (ii) add the neighbour to the frontier.
   else, Add neighbour's state to the explored set.
11) If no sol found, return failure.

15/10/24

```
import heapq

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

def find_zero(state):
    for i in range(3):
        for j in range(3):
            if state[i][j] == 0:
                return (i, j)

def move(state, direction):
    new_state = [row[:] for row in state]
    i, j = find_zero(state)
    if direction == "up" and i > 0:
        new_state[i][j], new_state[i-1][j] = new_state[i-1][j], new_state[i][j]
    elif direction == "down" and i < 2:
        new_state[i][j], new_state[i+1][j] = new_state[i+1][j], new_state[i][j]
    elif direction == "left" and j > 0:
        new_state[i][j], new_state[i][j-1] = new_state[i][j-1], new_state[i][j]
    elif direction == "right" and j < 2:
        new_state[i][j], new_state[i][j+1] = new_state[i][j+1], new_state[i][j]
    else:
        return None
    return new_state

def is_goal(state):
    return state == goal_state

def misplaced_tiles(state):
    return sum(1 for i in range(3) for j in range(3)
               if state[i][j] != goal_state[i][j] and state[i][j] != 0)

def a_star(initial_state):
    priority_queue = []
    heapq.heappush(priority_queue, (0, initial_state, [], 0))
    visited = set()

    while priority_queue:
        f, state, path, g = heapq.heappop(priority_queue)
        print("Exploring state in A*:")
        print_state(state)

        if is_goal(state):
            return path

        visited.add(tuple(map(tuple, state)))
```

```
 for direction in ["up", "down", "left", "right"]:
            new_state = move(state, direction)
            if new_state and tuple(map(tuple, new_state)) not in visited:
                h = misplaced_tiles(new_state)
                new_g = g + 1
                new_f = new_g + h
                heapq.heappush(priority_queue, (new_f, new_state, path + [direction], new_g))

    return None

def print_state(state):
    for row in state:
        print(row)
    print()

def get_initial_state():
    print("Enter the initial state (0 for empty space):")
    initial_state = []
    for i in range(3):
        row = list(map(int, input(f"Enter row {i+1} (space-separated): ").strip().split()))
        if len(row) != 3:
            raise ValueError("Each row must contain exactly 3 numbers.")
        initial_state.append(row)
    return initial_state

if __name___== "__main__":
    initial_state = get_initial_state()
    print("Initial State:")
    print_state(initial_state)
    print("Solving using A* search:")
    a_star_solution = a_star(initial_state)

    if a_star_solution:
        print("A* Solution:", a_star_solution)
    else:
        print("No solution found with A*.")
```

Output:

Enter the initial state (0 for empty space):
Enter row 1 (space-separated): 2 8 3
Enter row 2 (space-separated): 1 6 4
Enter row 3 (space-separated): 0 7 5
Initial State:
[2, 8, 3]
[1, 6, 4]
[0, 7, 5]

Solving using A* search:
Exploring state in A*:
[2, 8, 3]
[1, 6, 4]
[0, 7, 5]

Exploring state in A*:
[2, 8, 3]
[1, 6, 4]
[7, 0, 5]

Exploring state in A*:
[2, 8, 3]
[1, 0, 4]
[7, 6, 5]

Exploring state in A*:
[2, 0, 3]
[1, 8, 4]
[7, 6, 5]

Exploring state in A*:
[0, 2, 3]
[1, 8, 4]
[7, 6, 5]

Exploring state in A*:
[1, 2, 3]
[0, 8, 4]
[7, 6, 5]

Exploring state in A*:
[1, 2, 3]
[8, 0, 4]
[7, 6, 5]

A* Solution: ['right', 'up', 'up', 'left', 'down', 'right']

LAB PROGRAM 5 (B)

Implement Hill Climbing Algorithm.

22/10/24

Lab Program - 5

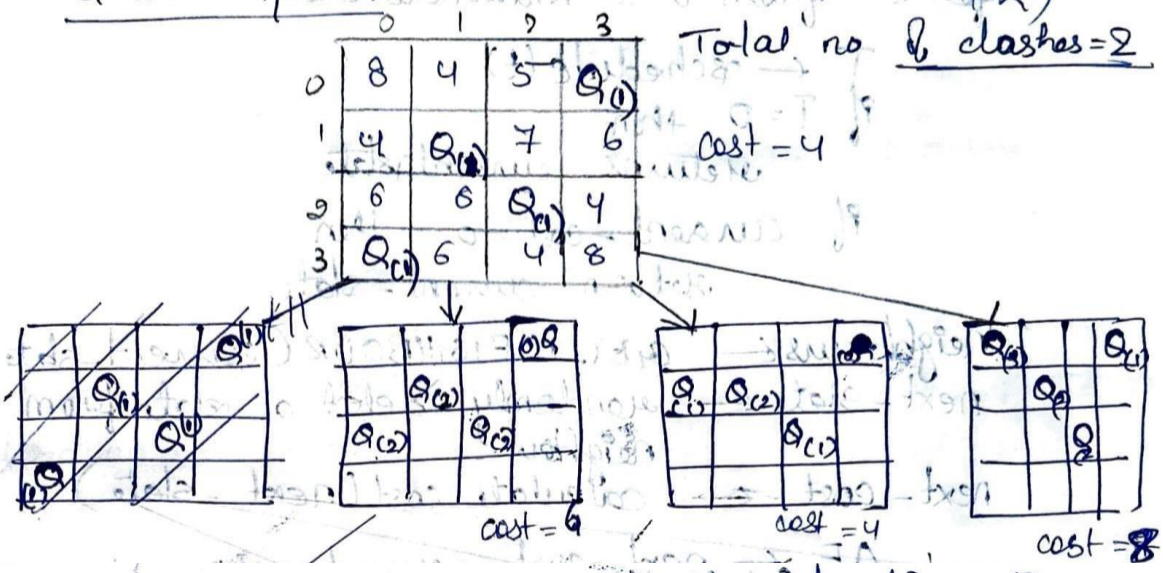
Implement Hill climbing search algorithm to solve n - queen problem.

```
# function to calculate number of attacking pair
function cal_attack(board):
    attack = 0
    for each queen i in board:
        for each queen j after i in board:
            if queens i in j are attacking:
                attacks ++;
    return attacks;

function generate_neighbour (board):
    neighbour = copy(board)
    randomly select a column to move
    randomly select a new row for the queen in
        selected col.
    update neighbour with the new queen position
    return neighbour

function hill_climbing_nqueen(n):
    current_board = generate_random_board(n).
    current_attacks = cal_attacks(current_board)
    while cur_attack >0:
        neighbour = generate_neighbour(current_
                                       board)
        neighbour_attacks = cal_attacks(neighbour)

        if neighbour_attacks < current_attacks:
            current_board = neighbour
            current_attacks = neighbour_attacks
```

else consider other moves on explore local minima
return current-board

n = 4
solution = hill climbing - n queens (n)
print sol

State Space

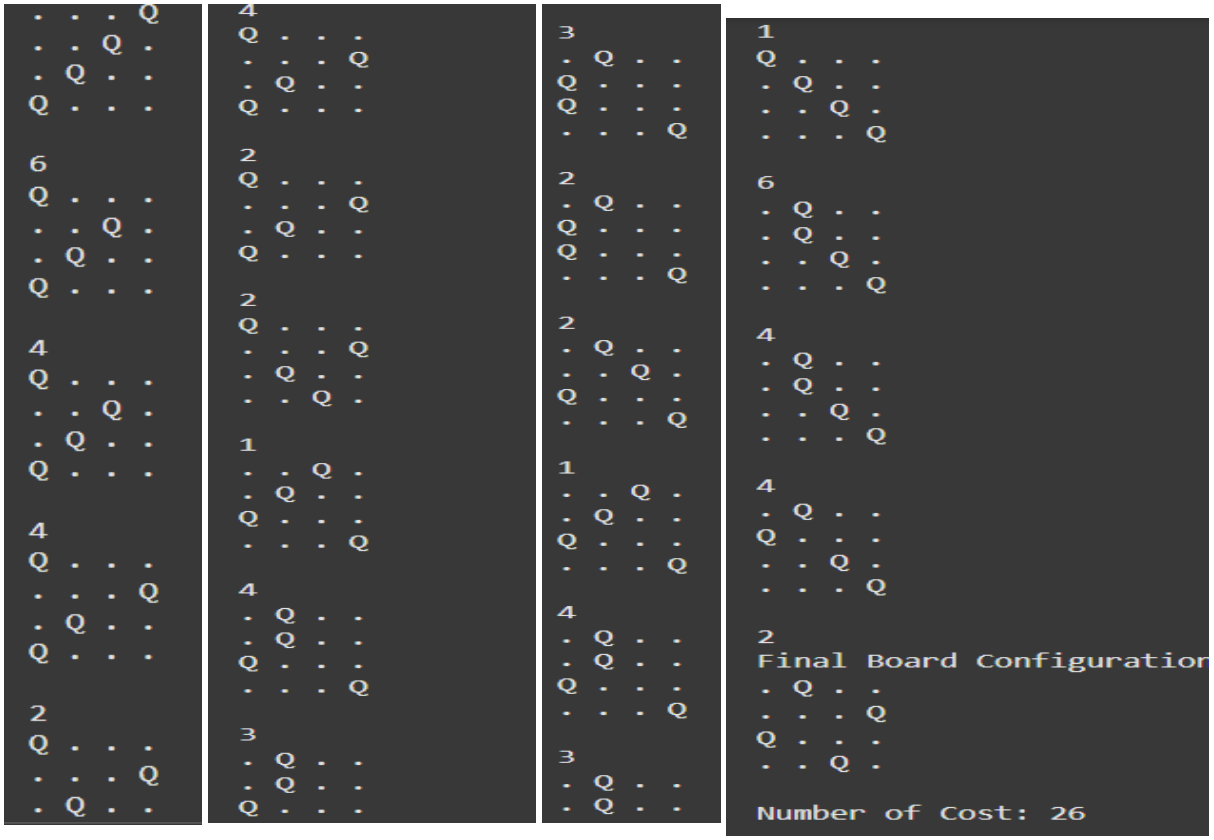


local max^m reached at iteration 0
No solution, only restarting as h ≤ cost

Implement Hill Climbing search algorithm to solve N-Queens problem

```
import random

def is_safe(board, row, col, n):
  """Checks if it's safe to place a queen at the given position."""
  for i in range(row):
    if board[i] == col or \
       abs(board[i] - col) == row - i:
      return False
  return True

def calculate_heuristic(board, n):
  """Calculates the number of pairs of queens attacking each other."""
  heuristic_value = 0
  for i in range(n):
    for j in range(i + 1, n):
      if board[i] == board[j] or abs(board[i] - board[j]) == j - i:
        heuristic_value += 1
  return heuristic_value

def hill_climbing_nqueens(n):
  """Solves the N-Queens problem using hill climbing."""
  current_board = [random.randint(0, n - 1) for _ in range(n)]
  current_heuristic = calculate_heuristic(current_board, n)

  while True:
    neighbors = []
    for row in range(n):
      for col in range(n):
        if current_board[row] != col:
          neighbor_board = current_board[:]
          neighbor_board[row] = col
          neighbors.append(neighbor_board)

    best_neighbor = None
    best_neighbor_heuristic = current_heuristic

    for neighbor in neighbors:
      neighbor_heuristic = calculate_heuristic(neighbor, n)
      if neighbor_heuristic < best_neighbor_heuristic:
      best_neighbor_heuristic = neighbor_heuristic
      best_neighbor = neighbor

    if best_neighbor is None:
      break  # No better neighbor found, local optima reached

    current_board = best_neighbor
    current_heuristic = best_neighbor_heuristic

  return current_board, current_heuristic
```

```
def print_board(board):
  n = len(board)
  for row in range(n):
    line = ""
    for col in range(n):
      if board[row] == col:
        line += "Q "
      else:
        line += ". "
    print(line)

# Example usage for 8-Queens
n = 8
solution, heuristic = hill_climbing_nqueens(n)

if heuristic == 0:
  print("Solution found:")
print_board(solution)
else:
  print("Local optima reached, not a perfect solution.")
  print("Heuristic value:", heuristic)
  print_board(solution)
```



LAB PROGRAM 6

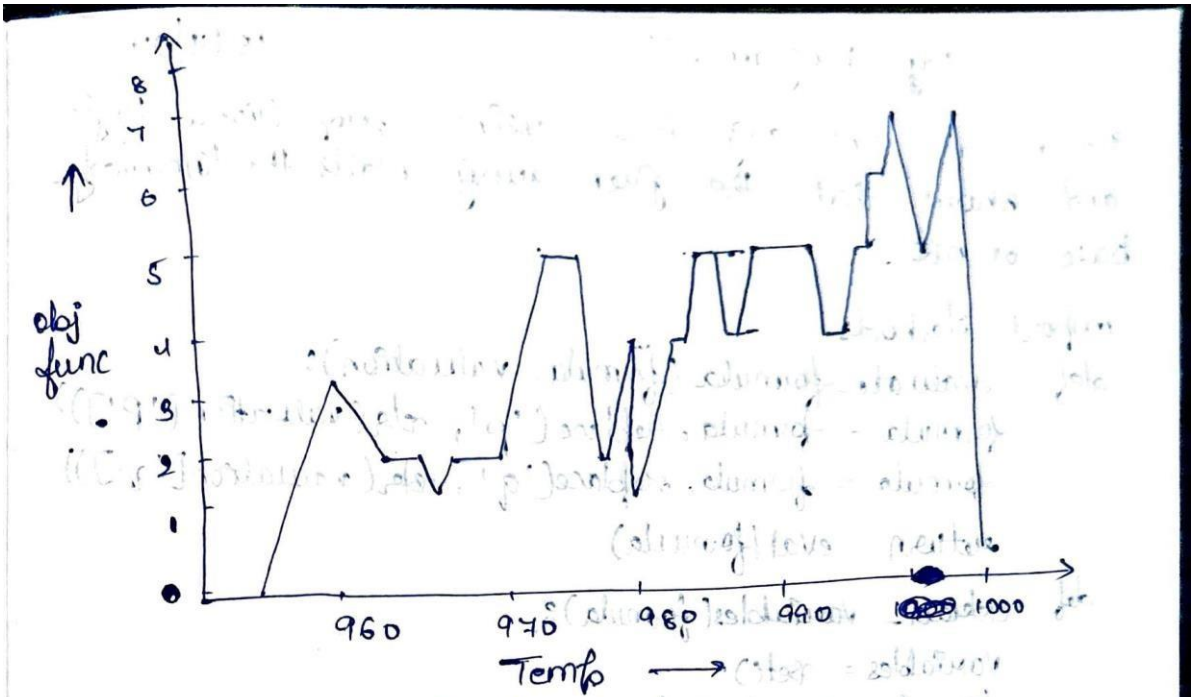
Write a program to implement Simulated Annealing Algorithm

29/10

Lab Program - 6

Implement Simulated Annealing Algorithms:

```
function SIMULATED_ANNEALING (initial_state, schedule,
                              max_iteration) & return a sol state
                                                & None
    current_state ← initial_state
    current_cost ← calculate_cost (current_state)
    for t from 0 to max_iterations - 1 do
        T ← schedule (t)
        if T = 0 then
            returns current state
        if current_cost = 0 then
            return current_state
        neighbours ← GET_NEIGHBOUR (current_state)
        next_state ← randomly select a state from
                     neighbour
        next_cost ← calculate_cost (next_state)
        ΔE ← next_cost - current_cost.
        if ΔE < 0 or random () < e^(-ΔE/T). then
            current_state ← next_state
            current_cost ← next_cost
        print (" Iteration', t,' : current state:",
        current_state", " Cost:", " current_cost "T:"
                                                T)
    print (" max iterations reached without finding
      a solution:")
    return none
```



For 4 Queens Problems:

3	1	2	0

initial state

Iteration 0: Current state : [3,1,2,0] cost = 2 T=10000
Next state : [0,1,2,0], Next Cost : 4
ΔE = 2
Acceptance probability : 0.9989

Iteration 41: Current state : [2,0,2,1] cost = 2, T=959
Next state = [2,0,3,1], Next cost = 0
ΔE = 2
Acceptance prob : 1.008

Solution found at iteration 42 : [2,0,3,1] with cost 0.

Write a program to implement Simulated Annealing.

```
import numpy as np
from scipy.optimize import dual_annealing

def queens_max(position):
    # This function calculates the number of pairs of queens that are not attacking
    each other position = np.round(position).astype(int)  # Round and convert to
    integers for queen positions n = len(position)
    queen_not_attacking = 0

    for i in range(n -
        1):
        no_attack_on_
        j = 0 for j in
        range(i + 1, n):
            # Check if queens are on the same row or on the same diagonal
            if position[i] != position[j] and abs(position[i] -
                position[j]) != (j - i): no_attack_on_j += 1
        if no_attack_on_j == n -
            1 - i:
            queen_not_attacking
            += 1
    if queen_not_attacking ==
        n - 1:
        queen_not_attacking
        += 1
    return -queen_not_attacking  # Negative because we want to maximize this value

# Bounds for each queen's position (0 to 7 for an 8x8
chessboard) bounds = [(0, 7) for _ in range(8)]

# Use dual_annealing for simulated annealing
optimization result =
dual_annealing(queens_max, bounds)

# Display the results
best_position = np.round(result.x).astype(int)
best_objective = -result.fun  # Flip sign to get the number of non-attacking queens

print('The best position found is:', best_position)
print('The number of queens that are not attacking each other is:', best_objective)
```

OUTPUT:

```
The best position found is: [6 3 1 7 5 0 2 4]
The number of queens that are not attacking each other is: 8
```

LAB PROGRAM 7

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

Lab Program - 7 12/11/24

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

```
import itertools
def evaluate_formula(formula, valuation):
    formula = formula.replace('p', str(valuation['p']))
    formula = formula.replace('q', str(valuation['q']))
    return eval(formula)
def extract_variables(formula):
    variables = set()
    for char in formula:
        if char.isalpha():
            variables.add(char)
    return list(variables)
def generate_truth_table(KB, query):
    variables = extract_variable(KB) + extract_variables(query)
    variables = list(set(variables))
    entails_query = True
    for assignment in itertools.product([False, True],
                repeat=len(variables)):
        valuation = dict(zip(variables, assignment))
        KB_truth = evaluate_formula(KB, valuation)
        query_truth = evaluate_formula(query, valuation)
        if KB_truth and not query_truth:
            entails_query = False
KB = input("enter the knowledge base
query = input("enter the query:)
generate_truth_table(KB, query)
```

O/P

Enter the knowledge base (e.g., 'pand (p|=q)): p|=q
Enter the query (e.g., 'q') = q
Truth table:

P	q	KB	Query
F	F	F	F
F	T	T	T
T	F	T	F
T	T	F	T

Code:

```
from itertools import product

# Evaluate a logical formula using the assignment of truth values
def evaluate_formula(formula, assignment):
    return eval(formula, {}, assignment)

# Generate all possible truth assignments for the given variables
def generate_all_assignments(variables):
    return [dict(zip(variables, values)) for values in product([True, False],
repeat=len(variables))]

# Create knowledge base and query from user input
def create_knowledge_base():
    print("Please enter the meanings of the following propositions:")
    p = input("Enter the meaning of proposition p : ")
    q = input("Enter the meaning of proposition q : ")
    r = input("Enter the meaning of proposition r : ")

    print(f"\nYou defined the following propositions:")
    print(f"p: {p}")
    print(f"q: {q}")
    print(f"r: {r}")

    print("\nNow, define the knowledge base (KB) and query (Q) using these propositions.")
    print("You can use 'p', 'q', 'r', 'not', 'or', 'and', and parentheses in your formulas.")

    KB = input("\nEnter the knowledge base (KB) formula : ")
    Q = input("\nEnter the query (Q) formula: ")

    return p, q, r, KB, Q

# Check if the knowledge base (KB) entails the query (Q) by evaluating the truth table
def truth_table_entailment(KB, Q, variables):
    all_assignments = generate_all_assignments(variables)

    print(f"\n{'p':<8}{'q':<8}{'r':<8}{'KB':<8}{'Q':<8}{'Entails'}")

    for assignment in all_assignments:
        KB_value = evaluate_formula(KB, assignment)
        Q_value = evaluate_formula(Q, assignment)
        entails = "Yes" if KB_value == True and Q_value == True else "No"

print(f"{assignment['p']:<8}{assignment['q']:<8}{assignment['r']:<8}{KB_value:<8}{Q_valu
e:<8}{entails}")

        if KB_value == True and Q_value == False:
            return False
    return True
```

```
# Main execution
if __name__== "__main__":
    p, q, r, KB, Q = create_knowledge_base()
    variables = ['p', 'q', 'r']

    # Check if the KB entails the query Q
    result = truth_table_entailment(KB, Q, variables)

    if result:
        print("\nKB entails Q.")
    else:
        print("\nKB does not entail Q.")
```

OUTPUT

```
Please enter the meanings of the following propositions:
Enter the meaning of proposition p : "It is raining"
Enter the meaning of proposition q : "The ground is wet"
Enter the meaning of proposition r : "The sun is shining"

You defined the following propositions:
p: "It is raining"
q: "The ground is wet"
r: "The sun is shining"

Now, define the knowledge base (KB) and query (Q) using these propositions.
You can use 'p', 'q', 'r', 'not', 'or', 'and', and parentheses in your formulas.

Enter the knowledge base (KB) formula : p and q

Enter the query (Q) formula: q

p       q       r       KB      Q       Entails
1       1       1       1       1       Yes
1       1       0       1       1       Yes
1       0       1       0       0       No
1       0       0       0       0       No
0       1       1       0       1       No
0       1       0       0       1       No
0       0       1       0       0       No
0       0       0       0       0       No

KB entails Q.
```

LAB PROGRAM 8

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

CODE:

```
from typing import List, Set, Dict, Union

def unify(literal1: str, literal2: str) -> Union[Dict[str, str], None]:
"""
Unify two literals and return a substitution dictionary, or None if they cannot be unified.
"""
if literal1 == literal2:
return {}
if literal1.startswith("~") and literal2.startswith("~"):
return None
if literal1.startswith("~"):
neg, pos = literal1, literal2
else:
neg, pos = literal2, literal1

if neg[1:] == pos:
return {}
return None

def apply_substitution(clause: Set[str], substitution: Dict[str, str]) -> Set[str]:
"""
Apply a substitution to a clause.
"""
new_clause = set()
for literal in clause:
for var, value in substitution.items():
literal = literal.replace(var, value)
new_clause.add(literal)
return new_clause

def resolve(clause1: Set[str], clause2: Set[str]) -> Union[Set[str], None]:
"""
Resolves two clauses. Returns the resolvent clause or None if resolution is not possible.
"""
for lit1 in clause1:
for lit2 in clause2:
substitution = unify(lit1, lit2)
if substitution is not None:
# Create a new clause with unified literals removed
new_clause = (clause1 - {lit1}) | (clause2 - {lit2})
return apply_substitution(new_clause, substitution)
return None

def resolution(knowledge_base: List[Set[str]], query: Set[str]) -> bool:
```

```
"""
Implements the resolution algorithm.
"""
    # Negate the query and add it to the knowledge base
    negated_query = {f"~{literal}" if not literal.startswith("~") else literal[1:] for literal
    in query} clauses = knowledge_base + [negated_query]

    new_clause

    s = set()

    while True:
    pairs = [(clauses[i], clauses[j]) for i in range(len(clauses)) for j in range(i + 1,
    len(clauses))] for clause1, clause2 in pairs:
    resolvent = resolve(clause1,
    clause2) if resolvent is not
    None:
    if not resolvent:  # Empty clause
    found return True
    new_clauses.add(frozenset(resolv
    ent))

    # If no new clauses are generated, resolution
    has failed if all(frozenset(c) in new_clauses
    for c in clauses):
    return False

    # Add new clauses to the set of clauses
    clauses.extend(map(set, new_clauses))

    if __name__== "__main__":
    print("Enter knowledge base (clauses) as sets of literals (comma-
    separated).") print("Example: Likes(John, Food), Food(Apple). Enter
    'done' when finished.")

    knowledge
    _base = []
    while True:
    clause = input("Clause:
    ").strip() if
    clause.lower() ==
    "done":
    break
    knowledge_base.append(set(lit.strip() for lit in clause.split(',')))

    print("Enter query as a set of literals (comma-
    separated).") query = set(lit.strip() for lit in
    input("Query: ").strip().split(','))

    result = resolution(knowledge_base, query)
    print("Result:", "Entailed (True)" if result else "Not Entailed (False)")
```

OUTPUT:

```
Enter knowledge base (clauses) as sets of literals (comma-separated).
Example: Likes(John, Food), Food(Apple). Enter 'done' when finished.
Clause: Likes(John, Food), Food(Apple)
Clause: ~Likes(John, Apple)
Clause: done
Enter query as a set of literals (comma-separated).
Query: Likes(John, Apple)
Result: Entailed (True)
```

Resolution in FOL and generation of proof tree by resolution

1. It's a crime for an American to sell weapons to hostile nations.

$\forall p \forall q \forall r (American(p) \wedge Weapon(q) \wedge Sells(p,q,r) \wedge Hostile(r) \Rightarrow criminal(p))$

$\neg American(p) \vee \neg Weapon(q) \vee \neg Sells(p,q,r) \vee Hostile(r) \vee criminal(p)$

2. Country A has some missiles

$\exists x (Owns(A,x) \wedge Missile(x))$

$Owns(A,T) \wedge Missile(T)$

3. All missiles were sold to country A by Robert.

$\forall x (Missile(x) \wedge Owns(A,x) \Rightarrow Sells(Robert,x,A))$

$\neg Missile(x) \vee \neg Owns(A,x) \vee Sells(Robert,x,A)$

4. Missiles are weapons

$\forall x (Missile(x) \Rightarrow Weapon(x))$

$\neg Missile(x) \vee Weapon(x)$

5. Enemy of america is hostile

$\forall x (Enemy(x, America) \Rightarrow Hostile(x))$

$\neg Enemy(x, America) \vee Hostile(x)$

6. Robert is an America $\therefore American(Robert)$

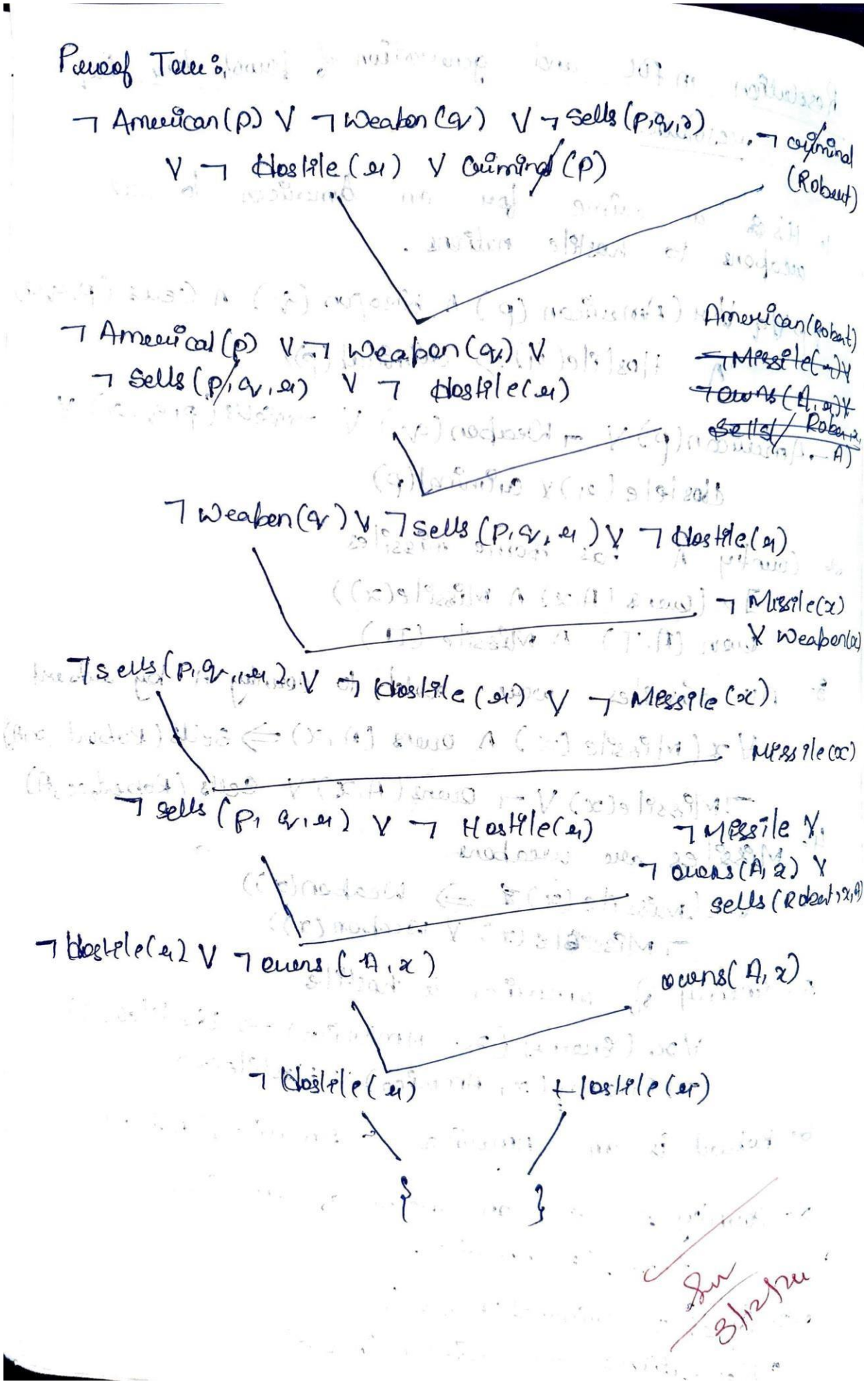
7. Country A is an enemy of America.

$Enemy(A, America)$.

$\rightsquigarrow$ Goal : $criminal(Robert)$

- Resolution : $\neg criminal(Robert)$

Proof Tree:



8w
3/12/24

LAB PROGRAM 9

Implement unification in first order logic.

```
#Implement unification in first order logic.

class
   UnificationError(Exce
   ption): pass

def unify(expr1, expr2,
   substitutions=None): if
   substitutions is None:
      substitutions = {}

   # If both expressions are identical, return current
   substitutions if expr1 == expr2:
      return substitutions

   # If the first expression is a
   variable if
   is_variable(expr1):
      return unify_variable(expr1, expr2, substitutions)

   # If the second expression is a
   variable if is_variable(expr2):
      return unify_variable(expr2, expr1, substitutions)

   # If both expressions are compound
   expressions if is_compound(expr1) and
   is_compound(expr2):
      if expr1[0] != expr2[0] or len(expr1[1:]) !=
         len(expr2[1:]): raise
         UnificationError("Expressions do not match.")
      return unify_lists(expr1[1:], expr2[1:], unify(expr1[0], expr2[0], substitutions))

   # If expressions are not compatible
   raise UnificationError(f"Cannot unify {expr1} and {expr2}.")

def unify_variable(var, expr,
   substitutions): if var in
   substitutions:
      return unify(substitutions[var], expr,
   substitutions) elif occurs_check(var, expr,
   substitutions):
      raise UnificationError(f"Occurs check failed: {var} in
   {expr}.") else:
      substitutions[var]
      = expr return
      substitutions
```

```
def unify_lists(list1, list2,
    substitutions): for expr1,
    expr2 in zip(list1, list2):
        substitutions = unify(expr1, expr2,
    substitutions) return substitutions

def is_variable(term):
    return isinstance(term, str) and term[0].islower()
def is_compound(term):
    return isinstance(term, (list, tuple)) and len(term) > 0

def occurs_check(var, expr,
    substitutions): if var == expr:
        return True
    elif is_compound(expr):
        return any(occurs_check(var, sub, substitutions) for
    sub in expr) elif expr in substitutions:
        return occurs_check(var, substitutions[expr],
    substitutions) return False

# Function to parse input into a usable expression
format def parse_expression(expr_str):
    # Try to evaluate the expression as a
    tuple or list try:
        expr = eval(expr_str)
        if isinstance(expr, (tuple, list)) and
            len(expr) > 0: return expr
        else:
            raise ValueError("Expression must be a non-empty tuple or
    list.") except Exception as e:
        raise ValueError(f"Invalid expression format: {e}")

# Example usage: allow user input for the expressions
try:
    expr1_str = input("Enter the first expression (e.g., ('f', 'x', ('g', 'y'))): ")
    expr2_str = input("Enter the second expression (e.g., ('f', 'a', ('g', 'b'))): ")
```

OUTPUT:

```
Enter the first expression (e.g., ('f', 'x', ('g', 'y'))): ('P', 'x', 'a', 'b')
Enter the second expression (e.g., ('f', 'a', ('g', 'b'))): ('P', 'y', 'z', 'b')
Unified substitutions: {'x': 'y', 'a': 'z'}
```

AIM: Implement unification in first order logic.

Pseudo code

```
FUNCTION unify(expr1, expr2, substitutions={}):
    IF expr1 == expr2:
        RETURN substitutions
    IF is_variable(expr1):
        RETURN unify_variable(expr1, expr2, substitutions)
    IF is_variable(expr2):
        RETURN unify_variable(expr2, expr1, substitutions)
    IF is_compound(expr1) AND is_compound(expr2):
        IF expr1[0] != expr2[0] OR: len(expr1[1:]) != len(expr2[1:]):
            RAISE "UnificationError: expressions do not match."
        FOR each pair (arg1, arg2) IN zip(expr1[1:], expr2[1:]):
            SUBSTITUTIONS = unify(arg1, arg2, substitutions)
        RETURN substitutions
    RAISE "Unification Error: Cannot unify expr1 and expr2"

FUNCTION unify_variable(var, expr, substitutions):
    IF var IN substitutions:
        RETURN unify(substitutions[var], expr, substitutions)
    IF occurs_check(var, expr, substitutions):
        RAISE "UnificationError: Occurs Check failed."
    substitutions[var] = expr
    RETURN substitutions

FUNCTION occurs_check(var, expr, substitutions):
    IF var == expr:
        RETURN true
    IF is_compound(expr):
        FOR sub IN expr:
            IF occurs_check(var, sub, substitutions):
                RETURN true
    IF expr IN substitutions:
        RETURN occurs_check(var, substitutions[expr], substitutions)
    RETURN false
```

Questions

1. Unify $P(x,a,b)$ and $P(y,z,b)$:

   Step 1: Compare the predicates, both are P so they are compatible.

   Step 2: Match arguments

   Unifier: $\{x=y, z=a\}$

2. Unify $\forall x\, P(x, f(y))$ and $P(z, f(a))$

   Substitution $= \Sigma$. First argument: second argument: $f(y)$ and $f(a)$ the function f are same so, $y=a$.

   Unifier:

   $\{x=z, y=a\}$

LAB PROGRAM 10

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

```
import re

# Helper functions to apply the transformations step by step.

# 1. Eliminate biconditionals and implications
def eliminate_biconditionals_implications(expr):
    # Eliminate biconditionals (⇔)
    expr = re.sub(r'([A-Za-z0-9()]+) ⇔ ([A-Za-z0-9()]+)', r'(\1 ⇒ \2) ∧ (\2 ⇒ \1)', expr)
    # Eliminate implications (⇒)
    expr = re.sub(r'([A-Za-z0-9()]+) ⇒ ([A-Za-z0-9()]+)', r'¬\1 ∨ \2', expr)
    return expr

# 2. Move negations inward
def move_negations_inward(expr):
    # Move negation inside ∀ and ∃
    expr = re.sub(r'¬\((∀x [A-Za-z0-9()]+)\)', r'∃x ¬\1', expr)
    expr = re.sub(r'¬\((∃x [A-Za-z0-9()]+)\)', r'∀x ¬\1', expr)
    # De Morgan's laws
    expr = re.sub(r'¬\(([^()]+) ∨ ([^()]+)\)', r'¬\1 ∧ ¬\2', expr)
    expr = re.sub(r'¬\(([^()]+) ∧ ([^()]+)\)', r'¬\1 ∨ ¬\2', expr)
    # Double negation elimination
    expr = re.sub(r'¬¬([A-Za-z0-9()]+)', r'\1', expr)
    return expr

# 3. Skolemization: Replace existential quantifiers with Skolem constants/functions
def skolemize(expr):
    # Replace ∃x with Skolem constant (G1)
    expr = re.sub(r'∃([A-Za-z0-9]+)', r'G1', expr)
    # Existential quantifier -> Skolem function
    expr = re.sub(r'∃([A-Za-z0-9]+)\(([A-Za-z0-9, ()]+)\)', r'F1(\2)', expr)
    return expr

# 4. Drop universal quantifiers
def drop_universal_quantifiers(expr):
    # Drop ∀ quantifiers
    expr = re.sub(r'∀([A-Za-z0-9]+)', '', expr)
    return expr

# 5. Distribute AND over OR to get CNF
def distribute_and_over_or(expr):
    # Distribute AND over OR
    expr = re.sub(r'(\([A-Za-z0-9()]+ ∧ [A-Za-z0-9()]+\)) ∨ ([A-Za-z0-9()]+)', r'(\1 ∨ \2)', expr)
    expr = re.sub(r'([A-Za-z0-9()]+ ∧ [A-Za-z0-9()]+) ∨ ([A-Za-z0-9()]+)', r'(\1 ∨ \2)', expr)
    return expr
```

```
# Convert to CNF using the above steps
def convert_to_cnf(expr):
    # Step 1: Eliminate biconditionals and implications
    expr = eliminate_biconditionals_implications(expr)
    # Step 2: Move negations inward
    expr = move_negations_inward(expr)
    # Step 3: Skolemize the expression
    expr = skolemize(expr)
    # Step 4: Drop universal quantifiers
    expr = drop_universal_quantifiers(expr)
    # Step 5: Distribute AND over OR to get CNF
    expr = distribute_and_over_or(expr)
    return expr

# Example FOL expressions (from the problem statement)
fol_expressions = [
    "Mary is the mother of John: Mother(Mary, John)",
    "John and Mary are both students: Student(John) ∧ Student(Mary)",
    "If it is raining, then the ground is wet: Raining ⇒ Wet(Ground)",
    "There is a person who knows every other person: ∃x ∀y (x ≠ y ⇒ Knows(x, y))",
    "Nobody is taller than themselves: ∀x ¬Taller(x, x)",
    "All students in the class passed the exam: ∀x (Student(x) ⇒ Passed(x, Exam))",
    "Mary has a pet dog: ∃x (Pet(x) ∧ Dog(x) ∧ Has(Mary, x))",
    "If Alice is a teacher, then Alice teaches mathematics: Teacher(Alice) ⇒ Teaches(Alice, Mathematics)",
    "Everyone loves someone: ∀x ∃y Loves(x, y)",
    "No one is both a teacher and a student: ∀x ¬(Teacher(x) ∧ Student(x))",
    "Every man respects his parent: ∀x (Man(x) ⇒ Respects(x, Parent(x)))",
    "Not all students like both Mathematics and Science: ¬∀x (Student(x) ⇒ (Likes(x, Mathematics) ∧
Likes(x, Science)))"
]

# Convert and print CNF for each expression
for expr in fol_expressions:
    print(f"Original FOL Expression: {expr}")
    expression = expr.split(":")[1].strip()  # Remove the description part and keep the formula
    cnf = convert_to_cnf(expression)
    print(f"CNF: {cnf}\n")
```

OUTPUT:

```
Original FOL Expression: Mary is the mother of John: Mother(Mary, John)
CNF: Mother(Mary, John)

Original FOL Expression: John and Mary are both students: Student(John) ∧ Student(Mary)
CNF: Student(John) ∧ Student(Mary)

Original FOL Expression: If it is raining, then the ground is wet: Raining ⇒ Wet(Ground)
CNF: ¬Raining ∨ Wet(Ground)

Original FOL Expression: There is a person who knows every other person: ∃x ∀y (x ≠ y ⇒ Knows(x, y))
CNF: G1  (x ≠ ¬y ∨ Knows(x, y))

Original FOL Expression: Nobody is taller than themselves: ∀x ¬Taller(x, x)
CNF:  ¬Taller(x, x)

Original FOL Expression: All students in the class passed the exam: ∀x (Student(x) ⇒ Passed(x, Exam))
CNF:  ¬(Student(x) ∨ Passed(x, Exam))

Original FOL Expression: Mary has a pet dog: ∃x (Pet(x) ∧ Dog(x) ∧ Has(Mary, x))
CNF: G1 (Pet(x) ∧ Dog(x) ∧ Has(Mary, x))

Original FOL Expression: If Alice is a teacher, then Alice teaches mathematics: Teacher(Alice) ⇒ Teaches(Alice, Mathematics)
CNF: ¬Teacher(Alice) ∨ Teaches(Alice, Mathematics)

Original FOL Expression: Everyone loves someone: ∀x ∃y Loves(x, y)
CNF:  G1 Loves(x, y)

Original FOL Expression: No one is both a teacher and a student: ∀x ¬(Teacher(x) ∧ Student(x))
CNF:  ¬(Teacher(x) ∧ Student(x))

Original FOL Expression: Every man respects his parent: ∀x (Man(x) ⇒ Respects(x, Parent(x)))
CNF:  ¬(Man(x) ∨ Respects(x, Parent(x)))

Original FOL Expression: Not all students like both Mathematics and Science: ¬∀x (Student(x) ⇒ (Likes(x, Mathematics) ∧ Likes(x, Science)))
CNF: ¬ ¬(Student(x) ∨ (Likes(x, Mathematics) ∧ Likes(x, Science)))
```

First Order Logic

1. Translate FOL and vice versa

① "John is a human".
Human(John)

② "Every human is mortal".
$\forall x \, Human(x) \Rightarrow Mortal(x)$

③ "John loves Mary"
Loves(John, Mary)

④ "There is someone who loves mary."
$\exists x \, Loves(x, Mary)$

⑤ "All dogs are animals".
$\forall x \, Dogs(x) \Rightarrow Animals(x)$

⑥ "Some dogs are brown".
$\exists x \, dog(x) \wedge brown(x)$

2. Translation of formal statements into English

(a) $\exists x [H(x) \wedge \forall y (\neg M(x,y)) \wedge U(x)]$

$M(x,y)$ means 'x is married to y.
$H(x)$ is a man
$U(x)$ means x is unhappy
$\forall y (\neg M(x,y))$:
x is not married to any y.
$U(x)$: is unhappy

$\Rightarrow$ "There exist a man who is not married to anyone and is unhappy."

(b) $\exists z [P(z,x) \wedge S(z,y) \wedge W(y)]$

$P(z,x)$: z is a parent of x.
$S(z,y)$: z and y are siblings.
$W(y)$: y is a woman

"There exist a person who is a parent of x, is a a sibling of y and y is a woman".

Pseudocode

```
function translated_to_fol(sent):
    sent = lowercase and trim(sent)
    if sent contains " is a human":
        return translate_is_a_human(sent)
    else if sent contains " is mortal":
        return translate_is_mortal(sent)
    else if sentence contains "loves":
        return translate_loves(sentence)
    // other specific sentence pattern
    else
        return "Translation not available for this
                sentence structure".

function translate_is_a_human(sentence):
    if sentence matches the pattern "subject is a human":
        subject = extracted subject
        return "H(subject)"
    else
        return "Invalid sentence structure"

function translate_is_mortal(sentence):
    // similar to translate_is_a_human

function main():
    // print instruction on how to use the translator
    while user input is not "exit":
        get sentence from the user
        fol_translation = translate_to_fol(sentence)
        print the fol translation
    // start the program
main()
```

LAB PROGRAM 11

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

3/12/24

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

Facts:

1. American (Robert)
2. Enemy (A, America)
3. Owns (A, T1)
4. Missile (T1)
5. Sells Weapon (Robert, T1)

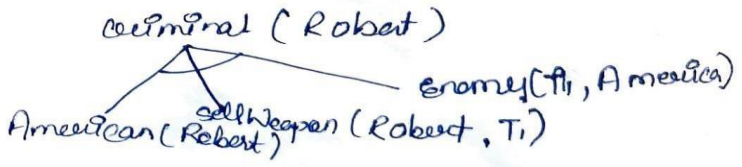
Rule:

American (Robert) ∧ Sells Weapon (Robert, T1) ∧ Enemy (A, America) ⇒ Criminal (Robert)

Representation in FOL

1. American (p) ∧ weapon (q) ∧ Sells (p, q, r) ∧ Hostile (r) ⇒ Criminal (p)
2. ∃x Owns (A, x) ∧ Missile (x)
3. Owns (A, T1)
   Missile (T1)
4. ∀x Missile (x) ∧ Owns (A, x) ⇒ Sells (Robert, x, A)
5. Missile (x) ⇒ Weapon (x)
6. ∀x Enemy (x, America) ⇒ Hostile (x)

Proof tree



CODE:

```
#ForwardReasoning
class
ForwardReasoning:
    def _init_(self, rules,
        facts): """
        Initializes the ForwardReasoning system.

        Parameters:
            rules (list): List of rules as tuples (condition, result),
                        where 'condition' is a set of
            facts. facts (set): Set of initial known
            facts.
        """
        self.rules = rules  # List of rules (condition -> result)
        self.facts = set(facts)  # Known facts

    def
        infer(self):
        """
        Applies forward reasoning to infer new facts based on rules and initial facts.

        Returns:
            set: Final set of facts after
        reasoning. """
        applied_rules = True

        while applied_rules:
            applied_rules = False
            for condition, result in self.rules:
                if condition.issubset(self.facts) and result not in self.facts:
                    self.facts.add(result)
                    applied_rules = True
                    print(f"Applied rule: {condition} -> {result}")
        return self.facts

# Define rules as (condition, result) where condition is a
set rules = [
        "American(Robert)"
    , "Hostile(CountryA)",
    "Missile(m1)",
    "Owns(CountryA,
    m1)",
    "Owns(CountryA, m) ^ Missile(m) => Sells(Robert, m,
```

```
    CountryA)", "American(x) ^ Hostile(y) ^ Sells(x, z, y) =>
    Criminal(x)"

]

# Define initial facts
facts = {"A", "D"}

# Initialize and run forward reasoning
reasoner = ForwardReasoning(rules,
facts) final_facts = reasoner.infer()

print("\nFinal
facts:")
print(final_facts)
```

OUTPUT:

```
Applied rule: {'American(Robert)'} -> Sells(Robert, m1, CountryA)
Applied rule: {'Sells(Robert, m1, CountryA)', 'Hostile(CountryA)', 'American(Robert)'} -> Criminal(Robert)

Final facts:
{'Hostile(CountryA)', 'Sells(Robert, m1, CountryA)', 'Criminal(Robert)', 'Missile(m1)', 'American(Robert)', 'Owns(CountryA, m1)'}

Query 'Criminal(Robert)' inferred: True
```

LAB PROGRAM 12

Implement Alpha-Beta Pruning.

CODE:

```
#Alpha-Beta
Pruning import
math

def minimax(depth, index, maximizing_player, values, alpha,
   beta): # Base case: when we've reached the leaf nodes
   if depth == 0:
      return values[index]

   if maximizing_player:
      max_eval = float('-inf')
      for i in range(2):  # 2 children per node
         eval = minimax(depth - 1, index * 2 + i, False, values, alpha,
         beta) max_eval = max(max_eval, eval)
         alpha = max(alpha, eval)
         if beta <= alpha:  # Beta
            cutoff break
      return
   max_eval else:
      min_eval = float('inf')
      for i in range(2):  # 2 children per node
         eval = minimax(depth - 1, index * 2 + i, True, values, alpha,
         beta) min_eval = min(min_eval, eval)
         beta = min(beta, eval)
         if beta <= alpha:  # Alpha
            cutoff break
      return min_eval

# Accept values from the user
leaf_values = list(map(int, input("Enter the leaf node values separated by spaces: ").split()))

# Check if the number of values is a power
of 2 if math.log2(len(leaf_values)) % 1 !=
0:
   print("Error: The number of leaf nodes must be a power of 2 (e.g., 2, 4, 8,
16).") else:
   # Calculate depth of the tree
   tree_depth = int(math.log2(len(leaf_values)))

   # Run Minimax with Alpha-Beta Pruning
```

```
    optimal_value = minimax(depth=tree_depth, index=0, maximizing_player=True,
values=leaf_values, alpha=float('-inf'), beta=float('inf'))

    print("Optimal value calculated using

Minimax:",optimal_value)
```

OUTPUT:

```
Enter the leaf node values separated by spaces: -1 8 -3 -1 2 1 -3 4
Optimal value calculated using Minimax: 2
```

17/12/24

AIM: Alpha Beta Pruning (cutoff) Search Algorithm

```
FUNCTION alpha-beta-pruning(node, depth, alpha,
    beta, maximising-player, path):
IF depth is 0 OR node is a terminal node THEN
    RETURN node's value, path

IF maximising-player THEN
    max-eval = negative infinity
    optimal-path = null
    FOR each child of node DO
        child-value, child-path = alpha-beta-
        pruning(child, depth-1, alpha, beta, FALSE,
            path + child's name)
        IF child-value > max-eval THEN
            max-eval = child-value
            optimal-path = child-path
        alpha = maximum(alpha, max-eval)
        if beta <= alpha THEN
            BREAK
    RETURN max-eval, optimal-path
ELSE
    min-eval = positive infinity
    optimal-path = null
    FOR each child of node DO
        child-value, child-path = alpha-beta-p
        pruning(child, depth-1, alpha, beta,
            TRUE, path + child's name)
        IF child-value < min-eval THEN
            min-eval = child-value
            optimal-path = child-path
        beta = min(beta, min-eval)
        IF beta <= alpha THEN
            Break
    RETURN min-eval, optimal-path
```

```
maximising_player = TRUE
intial_alpha = negative INFINITY
intial_beta = positive infinity
depth = 3
optimal_value, optimal_path = alpha_beta_pruning(root_node, depth, intial_alpha, intial_beta,
                                                 maximising_player)
print optimal_value
print optimal_path
```

Output

The optimal value is : 10

The optimal path is : A → B → D → H

Proof Tree