VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB REPORT
on

Artificial Intelligence (23CS5PCAIN)

Submitted by

Sarvesh Rastogi (1BM22CS247)

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 "Artificial Intelligence (23CS5PCAIN)" carried out by **Sarvesh Rastogi (1BM22CS247),** 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 Lab report has been approved as it satisfies the academic requirements in respect of an Artificial Intelligence (23CS5PCAIN) work prescribed for the said degree.

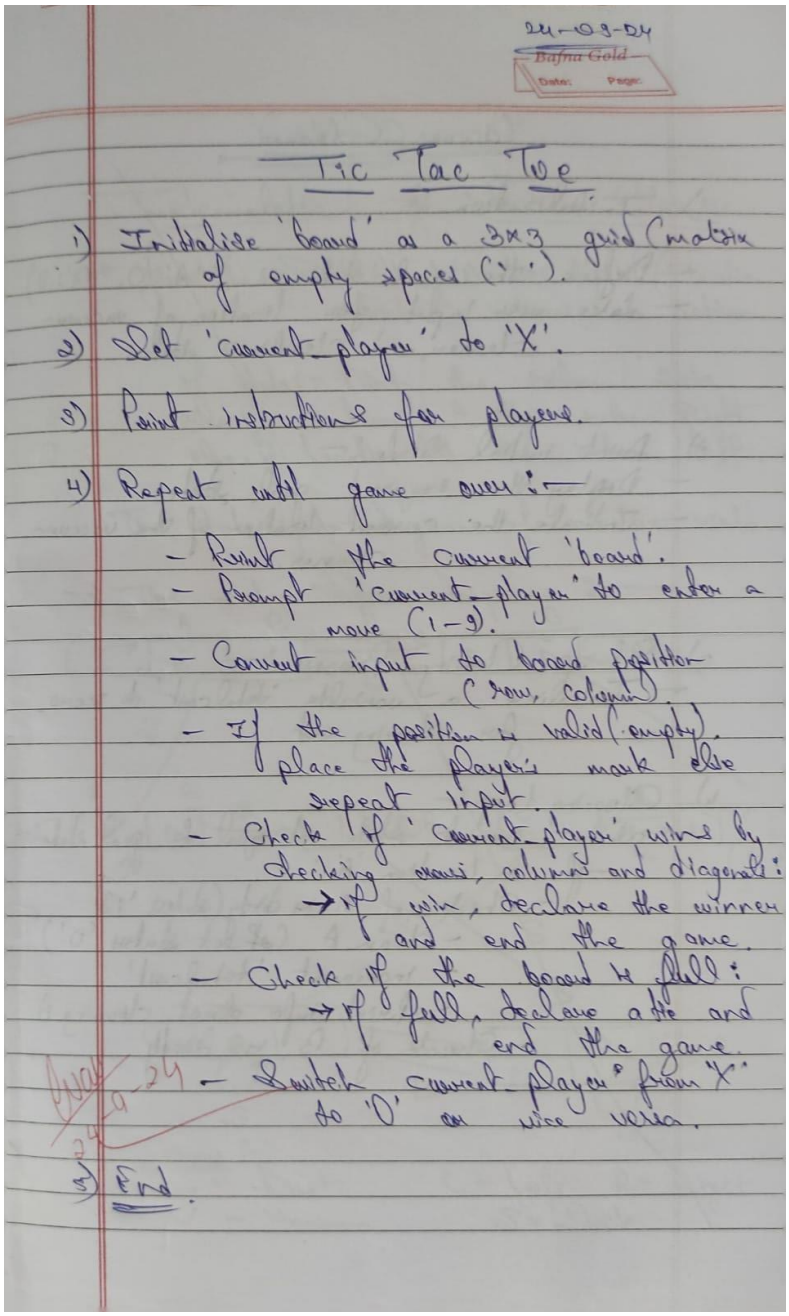
Sonika Sharma D Assistant Professor Department of CSE, BMSCE	Dr. Jyothi S Nayak Professor & HOD Department of CSE, BMSCE

Index

Github Link:
https://github.com/setu-mishra/AI_LAB

Program 1
Implement Tic –Tac –Toe Game

Algorithm:



Code:

```
# Set up the game board as a 2D list
board = [["-", "-", "-"],
        ["-", "-", "-"],
        ["-", "-", "-"]]
```

```
# Define a function to print the game board
def print_board():
    for row in board:
        print(" | ".join(row))

# Define a function to handle a player's turn
def take_turn(player):
    print(player + "'s turn.")
    position = input("Choose a position from 1-9: ")
    while position not in ["1", "2", "3", "4", "5", "6", "7", "8", "9"]:
        position = input("Invalid input. Choose a position from 1-9: ")
    position = int(position) - 1
    row, col = divmod(position, 3)
    while board[row][col] != "-":
        position = int(input("Position already taken. Choose a different position: ")) - 1
        row, col = divmod(position, 3)
    board[row][col] = player
    print_board()

# Define a function to check if the game is over
def check_game_over():
    # Check for a win
    for i in range(3):
        if board[i][0] == board[i][1] == board[i][2] != "-":
            return "win"
        if board[0][i] == board[1][i] == board[2][i] != "-":
            return "win"
    if board[0][0] == board[1][1] == board[2][2] != "-":
        return "win"
    if board[0][2] == board[1][1] == board[2][0] != "-":
        return "win"
    # Check for a tie
    elif all(cell != "-" for row in board for cell in row):
        return "tie"
    # Game is not over
    else:
        return "play"

# Define the main game loop
def play_game():
    print_board()
    current_player = "X"
    game_over = False
    while not game_over:
        take_turn(current_player)
        game_result = check_game_over()
        if game_result == "win":
            print(current_player + " wins!")
```

```
            game_over = True
        elif game_result == "tie":
            print("It's a tie!")
            game_over = True
        else:
            # Switch to the other player
            current_player = "O" if current_player == "X" else "X"

# Start the game
play_game()
```

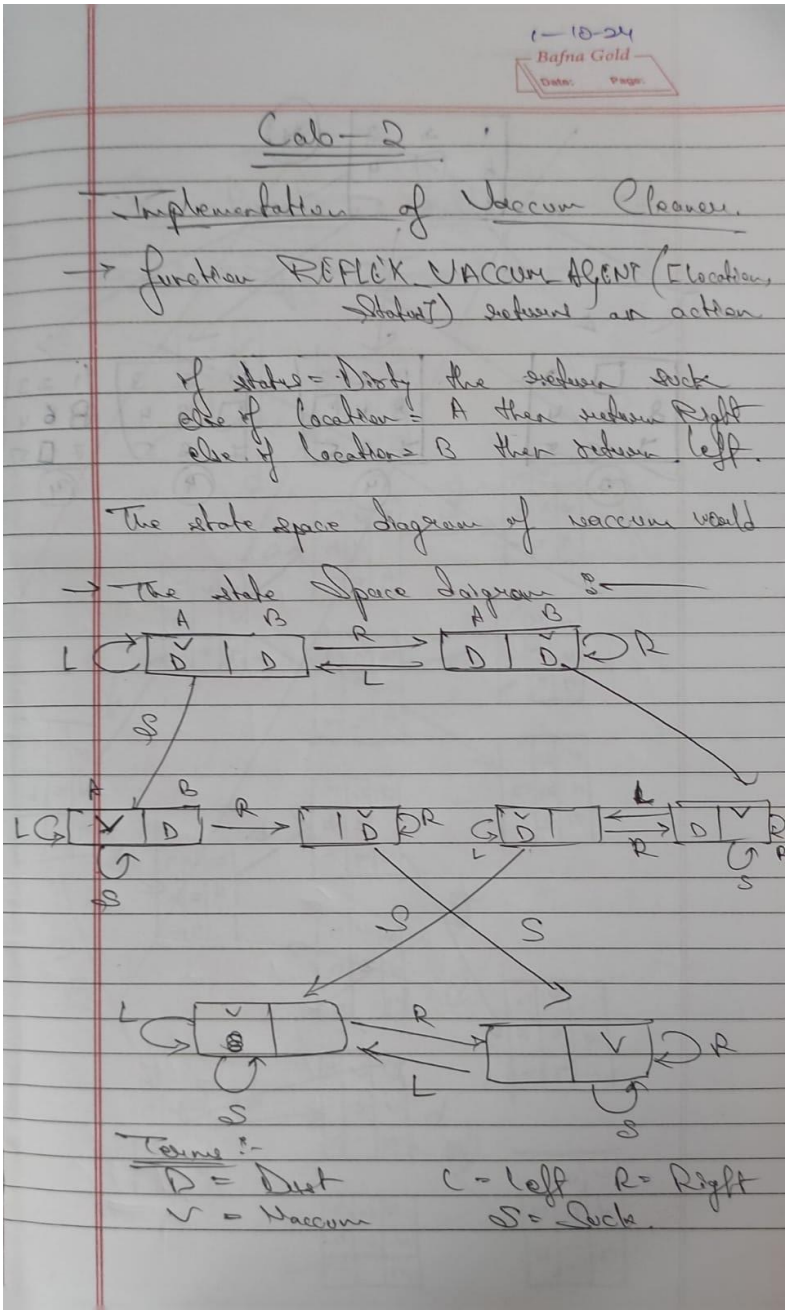
Output:

```
- | - | -
- | - | -
- | - | -
X's turn.
Choose a position from 1-9: 1
X | - | -
- | - | -
- | - | -
O's turn.
Choose a position from 1-9: 2
X | O | -
- | - | -
- | - | -
X's turn.
Choose a position from 1-9: 3
X | O | X
- | - | -
- | - | -
O's turn.
Choose a position from 1-9: 5
X | O | X
- | O | -
- | - | -
X's turn.
Choose a position from 1-9: 4
X | O | X
X | O | -
- | - | -
O's turn.
Choose a position from 1-9: 6
X | O | X
X | O | O
- | - | -
X's turn.
Choose a position from 1-9: 8
X | O | X
```

X | O | O
- | X | -
O's turn.
Choose a position from 1-9: 7
X | O | X
X | O | O
O | X | -
X's turn.
Choose a position from 1-9: 9
X | O | X
X | O | O
O | X | X
It's a tie!

Implement vacuum cleaner agent

Algorithm:



Code:

```
#For two quadrants
def vacuum_cleaner_simulation():

    current_room = input("Enter current room either A or B: ").upper()
    room_A = int(input("Is Room A dirty? (yes:1/no:0): "))
    room_B = int(input("Is Room B dirty? (yes:1/no:0): "))

    cost = 0

    def display_rooms():
        print(f"Room A: {'Clean' if room_A == 0 else 'Dirty'}")
        print(f"Room B: {'Clean' if room_B == 0 else 'Dirty'}")

    print("\nInitial status of rooms:")
    display_rooms()
    print()

    while room_A == 1 or room_B == 1:
        if current_room == 'A' and room_A == 1:
            print("Cleaning Room A...")
            room_A = 0
            cost += 1
        elif current_room == 'B' and room_B == 1:
            print("Cleaning Room B...")
            room_B = 0
            cost += 1
        else:
            current_room = 'B' if current_room == 'A' else 'A'
            print(f"Moving to Room {current_room}...")
        print("Current status:")
        display_rooms()

    print(f"\nBoth rooms are now clean! Total cost: {cost}")

vacuum_cleaner_simulation()

#For four quadrants
def vacuum_cleaner_simulation():
    current_room = input("Enter current room (A, B, C, or D): ").upper()
    room_A = int(input("Is Room A dirty? (yes:1/no:0): "))
    room_B = int(input("Is Room B dirty? (yes:1/no:0): "))
    room_C = int(input("Is Room C dirty? (yes:1/no:0): "))
    room_D = int(input("Is Room D dirty? (yes:1/no:0): "))

    cost = 0
    count=2
    def display_rooms():
```

```
    print(f"Room A: {'Clean' if room_A == 0 else 'Dirty'}")
    print(f"Room B: {'Clean' if room_B == 0 else 'Dirty'}")
    print(f"Room C: {'Clean' if room_C == 0 else 'Dirty'}")
    print(f"Room D: {'Clean' if room_D == 0 else 'Dirty'}")

print("\nInitial status of rooms:")
display_rooms()
print()


while room_A == 1 or room_B == 1 or room_C == 1 or room_D == 1:
    if count==0:
      print("Vacuum is recharging")
      count=2
    else:
      if current_room == 'A' and room_A == 1:
          print("Cleaning Room A...")
          room_A = 0
          cost += 1
          count-=1
      elif current_room == 'B' and room_B == 1:
          print("Cleaning Room B...")
          room_B = 0
          cost += 1
          count-=1
      elif current_room == 'C' and room_C == 1:
          print("Cleaning Room C...")
          room_C = 0
          cost += 1
          count-=1
      elif current_room == 'D' and room_D == 1:
          print("Cleaning Room D...")
          room_D = 0
          cost += 1
          count-=1
      else:
          if current_room == 'A':
              current_room = 'B'
          elif current_room == 'B':
              current_room = 'C'
          elif current_room == 'C':
              current_room = 'D'
          else:
              current_room = 'A'
          print(f"Moving to Room {current_room}...")

print("\nCurrent status:")
display_rooms()
```

```
    print(f"\nAll rooms are now clean! Total cost: {cost}")

vacuum_cleaner_simulation()
```

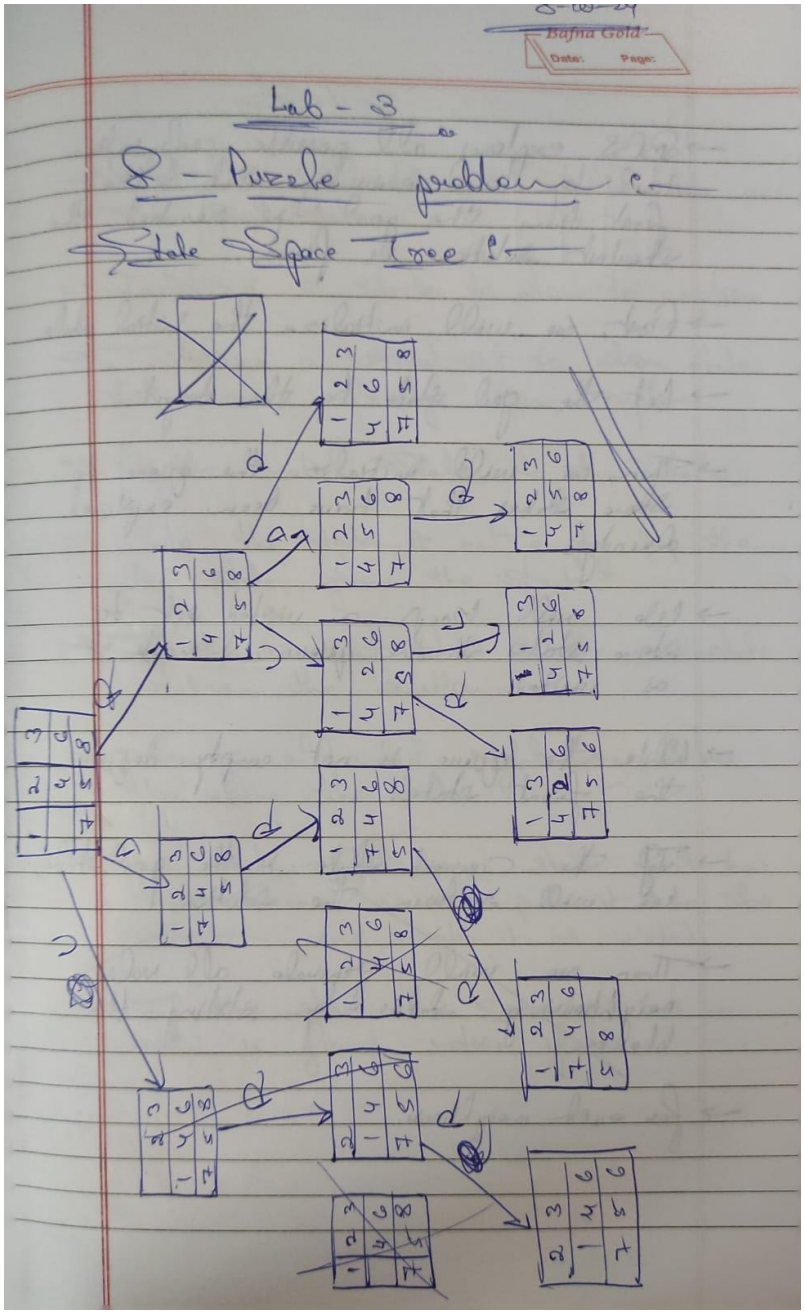
Output:

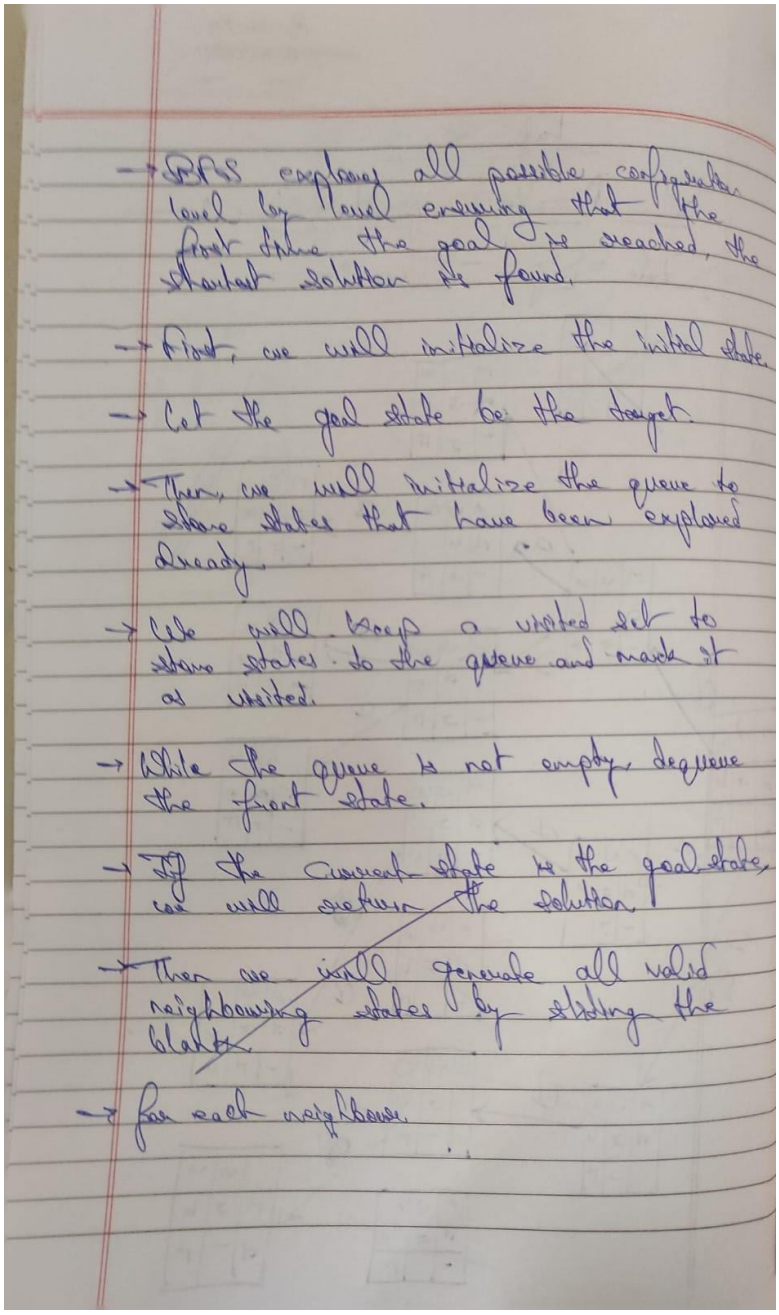
```
Enter current room either A or B: A
Is Room A dirty? (yes:1/no:0): 0
Is Room B dirty? (yes:1/no:0): 1

Initial status of rooms:
Room A: Clean
Room B: Dirty

Moving to Room B...
Current status:
Room A: Clean
Room B: Dirty
Cleaning Room B...
Current status:
Room A: Clean
Room B: Clean

Both rooms are now clean! Total cost: 1
Enter current room (A, B, C, or D): A
Is Room A dirty? (yes:1/no:0): 0
Is Room B dirty? (yes:1/no:0): 0
Is Room C dirty? (yes:1/no:0): 1
Is Room D dirty? (yes:1/no:0): 0

Initial status of rooms:
Room A: Clean
Room B: Clean
Room C: Dirty
Room D: Clean

Moving to Room B...
Moving to Room C...
Cleaning Room C...

Current status:
Room A: Clean
Room B: Clean
Room C: Clean
Room D: Clean

All rooms are now clean! Total cost: 1
```

Program 2

Implement 8 puzzle problems using Depth First Search (DFS)

Algorithm:




→ BFS explores all possible configuration level by level ensuring that the first time the goal is reached, the shortest solution is found.
→ First, we will initialize the initial state.
→ Set the goal state to the target.
→ Then, we will initialize the queue to store states that have been explored already.
→ We will keep a visited set to store states. to the queue and mark it as visited.
→ While the queue is not empty, dequeue the front state.
→ If the current state is the goal state, we will return the solution.
→ Then we will generate all valid neighbouring states by sliding the blank.
→ for each neighbour.

Bafna Gold

Date: Page:

DFS

→ First we will take the initial state and the goal state.

→ Initialize a stack to store to explore

→ Initialize a visited set to store states that have already been explored.

→ Add the initial state to the stack.

→ While the stack is not empty pop the top state from the stack.

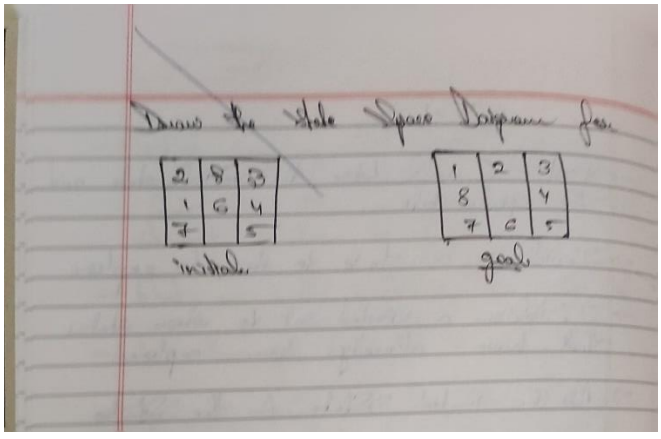
→ If the current state is the goal state return the solution.

→ Generate all valid neighbouring states by sliding the blank.

→ For each neighbouring state if it has not been visited, push it to the stack and mark it as visited.

→ If the stack is exhausted and no soln. is found, return failure.

8/10/24

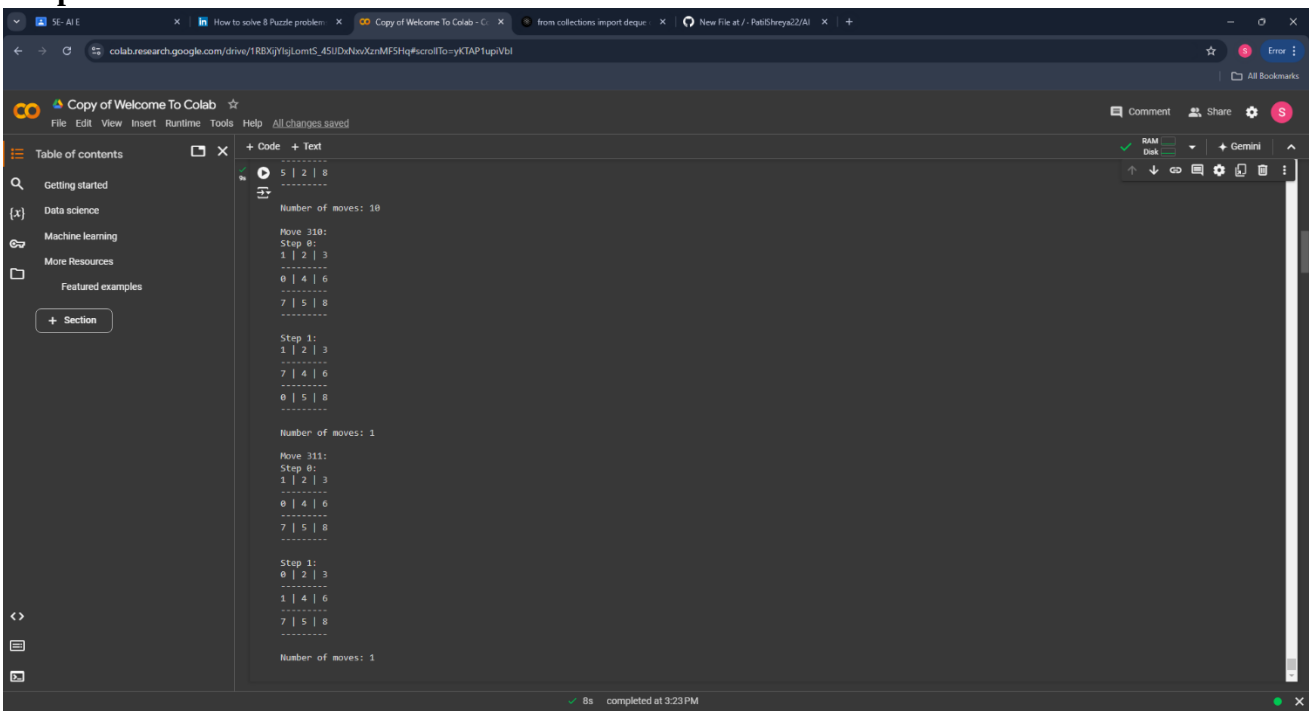


Code:

```
from collections import deque

def dfs(start, max_depth):
    stack = deque([(start, [start], 0)])  # (node, path, level)
    visited = set([start])
    all_moves = []
    while stack:
        node, path, level = stack.pop()
        all_moves.append((path, level))
        if level < max_depth:
            for next_node in get_neighbors(node):
                if next_node not in visited:
                    visited.add(next_node)
                    stack.append((next_node, path + [next_node], level + 1))
    return all_moves

def get_neighbors(node):
    neighbors = []
    for i in range(9):
        if node[i] == 0:
            x, y = i // 3, i % 3
            for dx, dy in [(-1, 0), (1, 0), (0, -1), (0, 1)]:
                nx, ny = x + dx, y + dy
                if 0 <= nx < 3 and 0 <= ny < 3:
                    n = list(node)
                    n[i], n[nx * 3 + ny] = n[nx * 3 + ny], n[i]
                    neighbors.append(tuple(n))
            break
    return neighbors

def print_board(board):
    board = [board[i:i+3] for i in range(0, 9, 3)]
    for row in board:
        print(" | ".join(str(x) for x in row))
        print("---------")

def main():
    start = tuple(int(x) for x in input("Enter the initial state (space-separated): ").split())
    max_depth = 10  # maximum depth to search
    all_moves = dfs(start, max_depth)
    if all_moves:
        print("All possible moves:")
        for i, (path, level) in enumerate(all_moves):
            print(f"Move {i+1}:")
            for j, node in enumerate(path):
                print(f"Step {j}:")
                print_board(node)
```

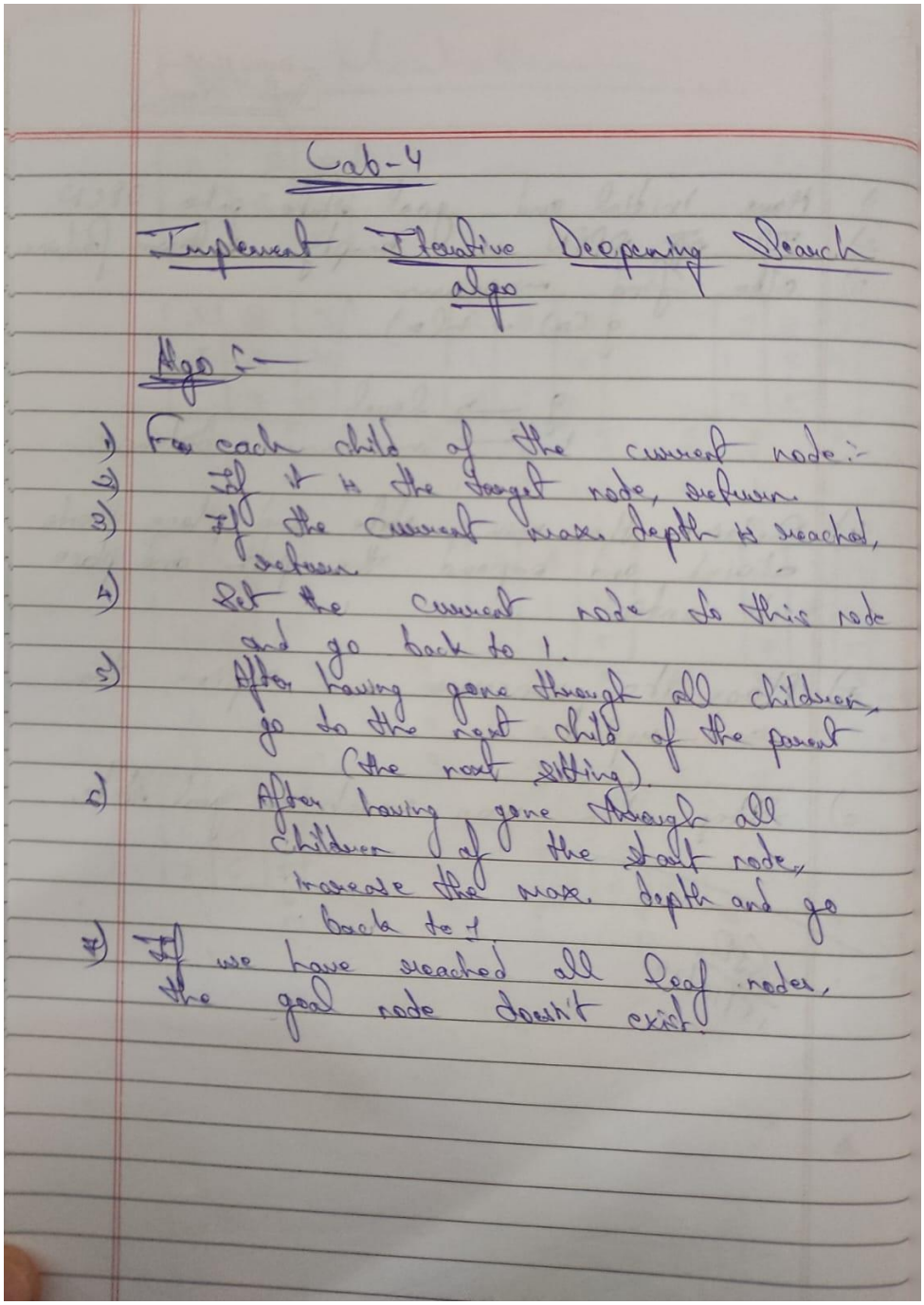
```
            print()
        print(f"Number of moves: {level}")
        print()
    else:
        print("No solution found.")

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

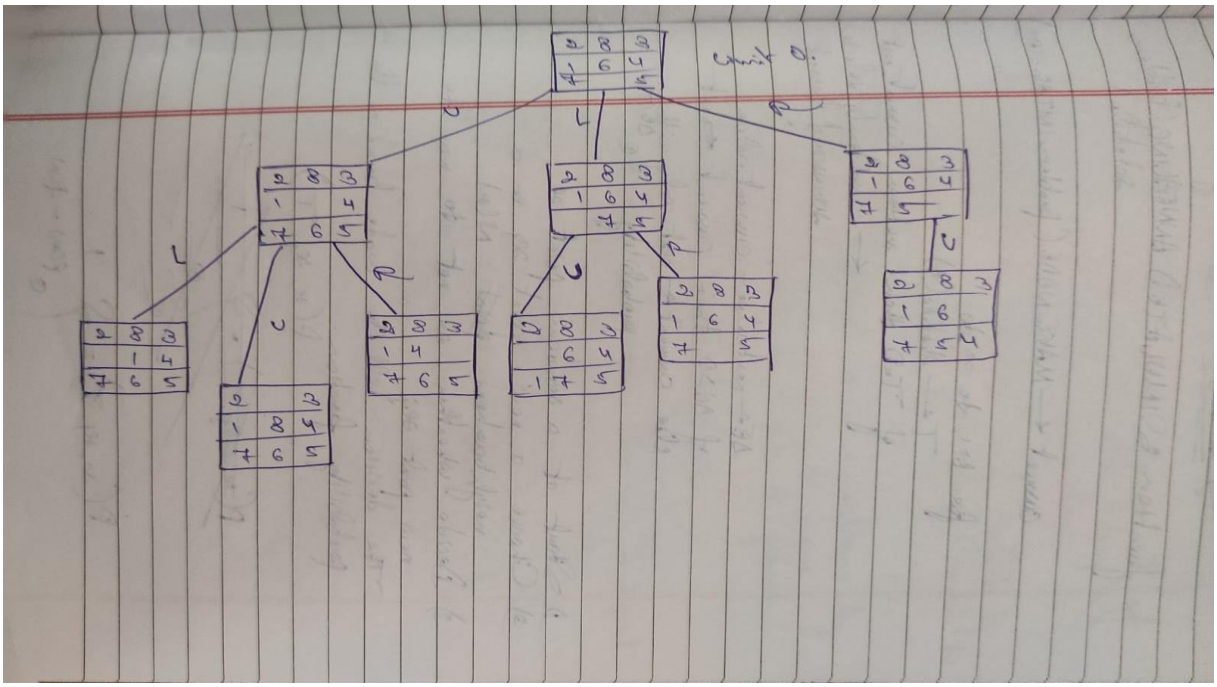
Output:



Implement Iterative deepening search algorithm

Algorithm:



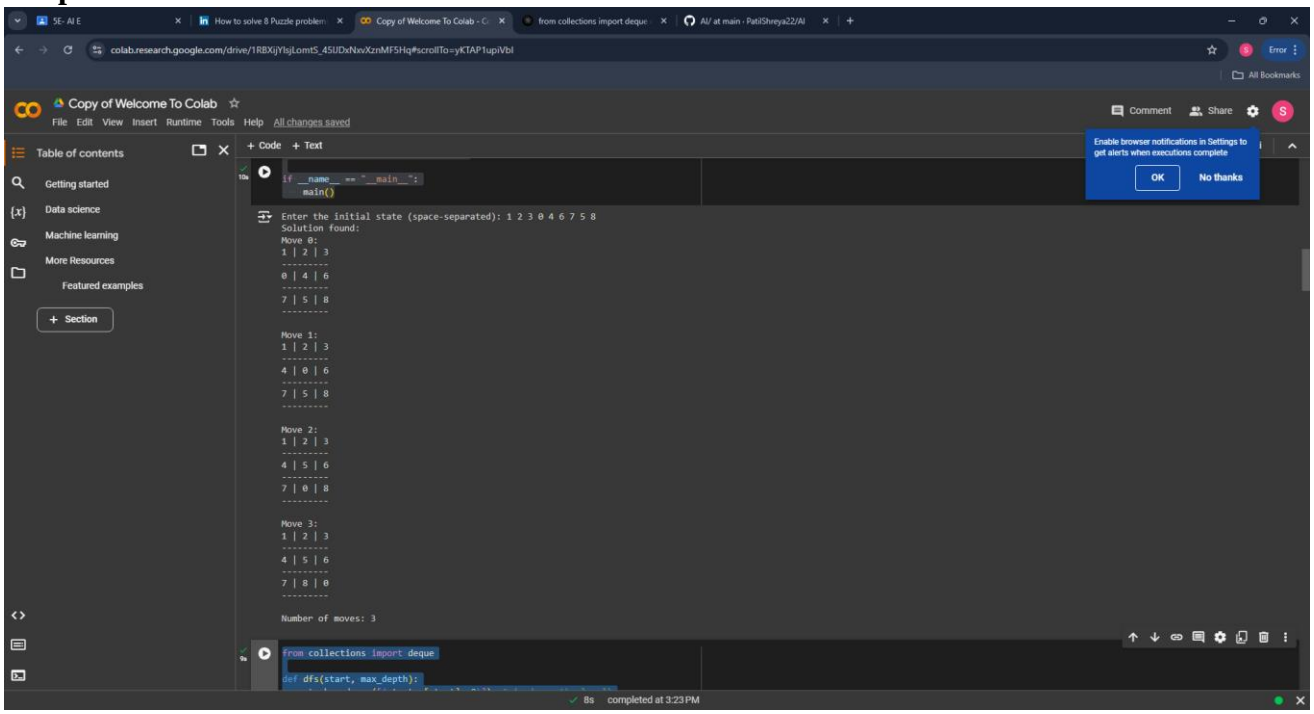


Code:

```
from collections import deque

def bfs(start, goal):
    queue = deque([(start, [start], 0)])  # (node, path, level)
    visited = set([start])
    while queue:
        node, path, level = queue.popleft()
        if node == goal:
            return path, level
        for next_node in get_neighbors(node):
            if next_node not in visited:
                visited.add(next_node)
                queue.append((next_node, path + [next_node], level + 1))
    return None, None

def get_neighbors(node):
    neighbors = []
    for i in range(9):
        if node[i] == 0:
            x, y = i // 3, i % 3
            for dx, dy in [(-1, 0), (1, 0), (0, -1), (0, 1)]:
                nx, ny = x + dx, y + dy
                if 0 <= nx < 3 and 0 <= ny < 3:
                    n = list(node)
                    n[i], n[nx * 3 + ny] = n[nx * 3 + ny], n[i]
                    neighbors.append(tuple(n))
```

```
            break
    return neighbors

def print_board(board):
    board = [board[i:i+3] for i in range(0, 9, 3)]
    for row in board:
        print(" | ".join(str(x) for x in row))
        print("---------")

def main():
    start = tuple(int(x) for x in input("Enter the initial state (space-separated): ").split())
    goal = (1, 2, 3, 4, 5, 6, 7, 8, 0)
    path, level = bfs(start, goal)
    if path:
        print("Solution found:")
        for i, node in enumerate(path):
            print(f"Move {i}:")
            print_board(node)
            print()
        print(f"Number of moves: {level}")
    else:
        print("No solution found.")

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

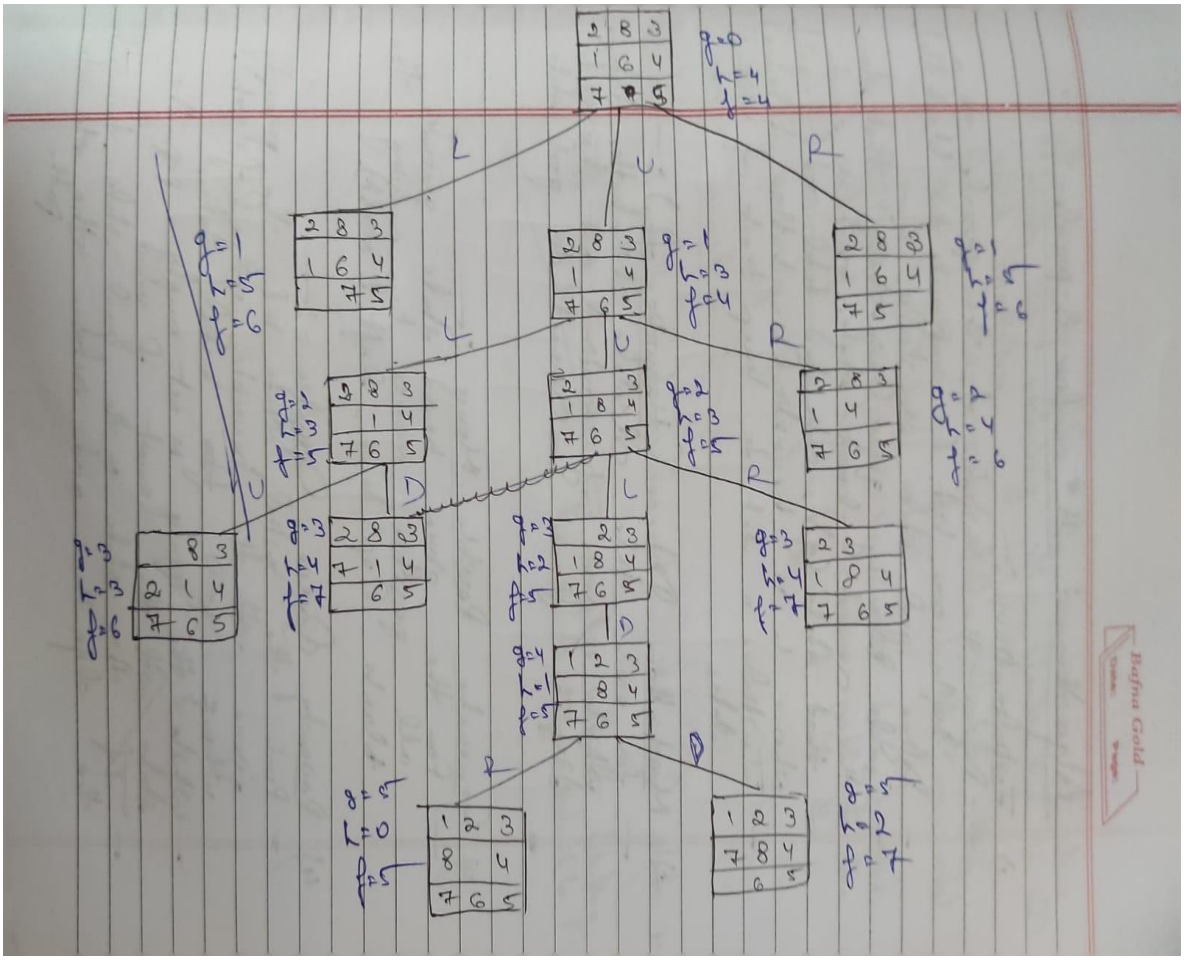
Output:



Program 3

Implement A* search algorithm

Algorithm:



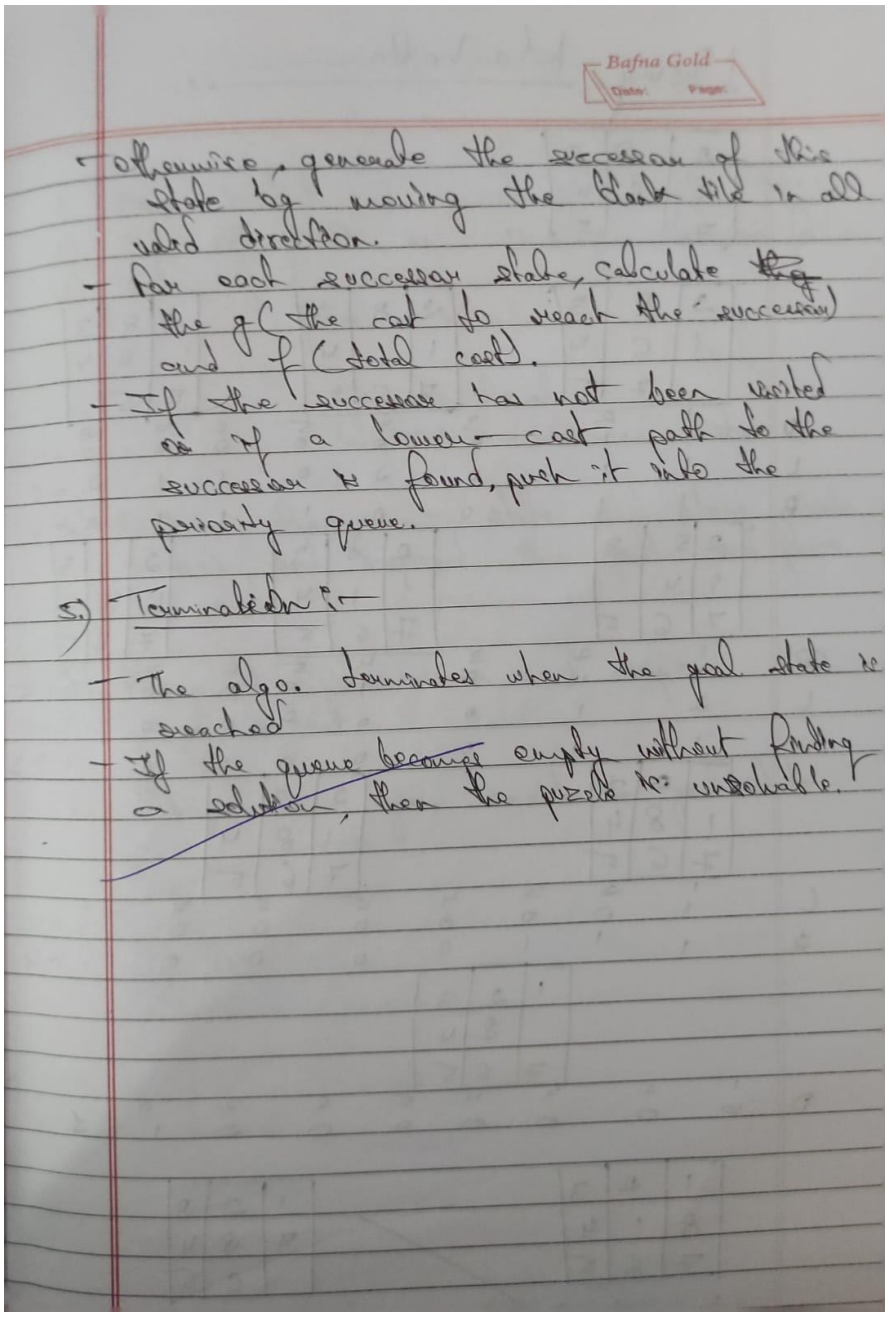
Algorithm : A* for 8-puzzle.

1) Initialization :-
- Define the initial of the puzzle as a tuple or list. (0 represents the blank tile).
- Define goal state, similarly.
- Create priority queue (min-heap) that stores the current board state, cost to reach that state (g), estimated total cost ($f = g + h$) where h is heuristic (Manhattan distance).
- Initialize a set to keep track of visited states.

2) Manhattan Distance (Heuristic fn.) :-
- For each tile calculate Manhattan distance which is sum of vertical and horizontal distance from current to goal state.

3) Generate Possible Moves :-
- For the given board state generate all possible moves by moving the blank (up, down, left, right).

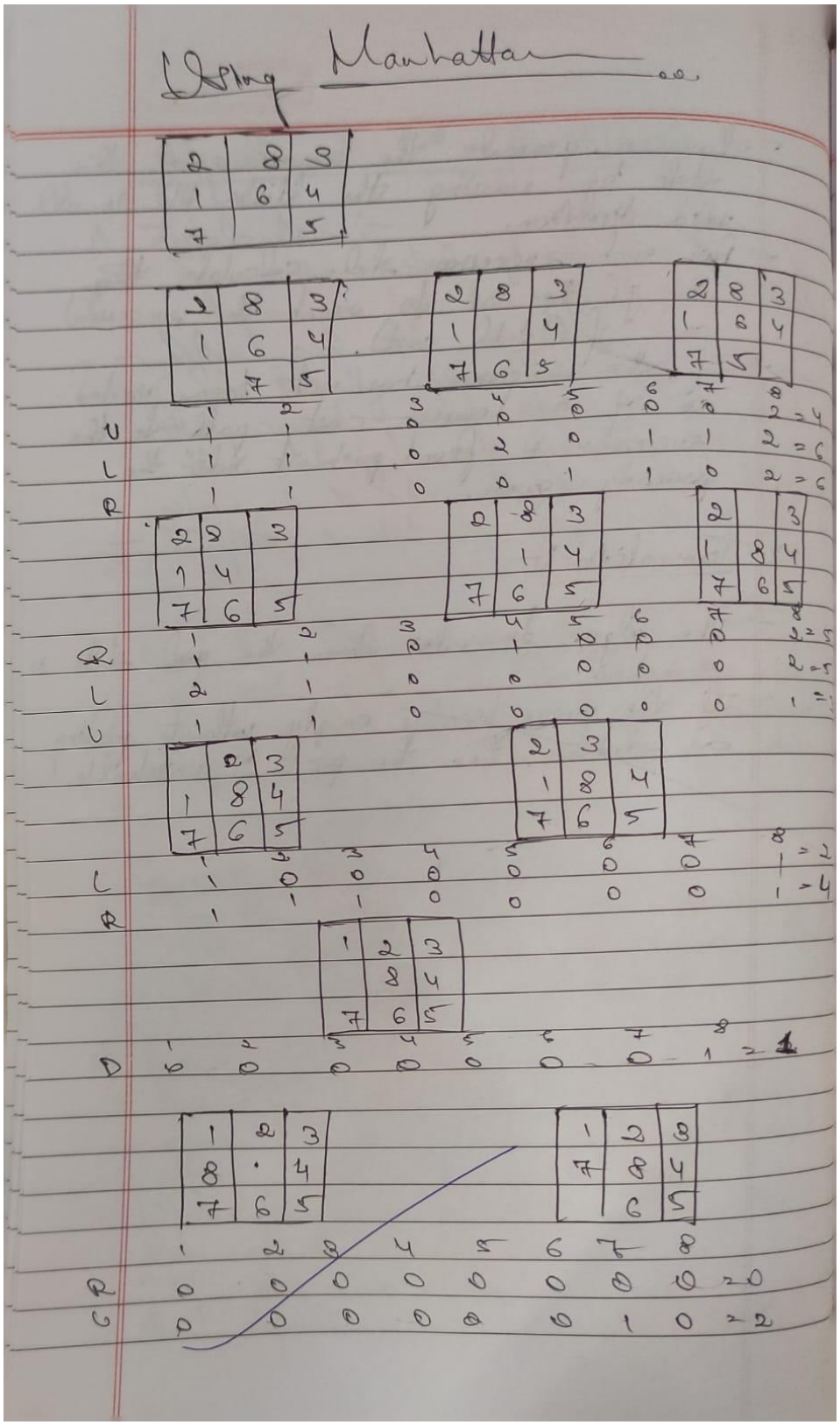
4) Priority Queue Operations :-
- Push the initial state into the priority queue with the heuristic value ($f = g + h$), where $g = 0$ for initial state.
- While the queue is not empty, pop the state with the lowest f value.
- If this state is the goal state, terminate the algorithm and return the path.


Bafna Gold
Date: Page:

- Otherwise, generate the successor of this state by moving the blank tile in one valid direction.
- For each successor state, calculate the g (the cost to reach the successor) and f (total cost).
- If the successor has not been visited or if a lower-cost path to the successor is found, push it into the priority queue.

5) Termination :-

- The algo. terminates when the goal state is reached.
- If the queue becomes empty without finding a solution, then the puzzle is unsolvable.


Using Manhattan

1) Have initial and goal state onto OPEN.
2) If OPEN is empty return failure
3) else find minimum
$g(x) + h(x)$

$g \longrightarrow$ level
$h \longrightarrow$ minimum distance

4) Retrive the min state and place it onto closed and expand the path and place it onto

5) Place its successor and explore them and repeat.

6) Stop when you reach its goal state.

SRD
15/10/24

Code:

```
import heapq

def misplaced_tile(state, goal_state):
    misplaced = 0
    for i in range(3):
        for j in range(3):
            if state[i][j] != 0 and state[i][j] != goal_state[i][j]:
                misplaced += 1
    return misplaced

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

def generate_neighbors(state):
    neighbors = []
    x, y = find_blank(state)
    directions = [(0, 1), (0, -1), (1, 0), (-1, 0)]

    for dx, dy in directions:
        nx, ny = x + dx, y + dy
        if 0 <= nx < 3 and 0 <= ny < 3:
            new_state = [list(row) for row in state]
            new_state[x][y], new_state[nx][ny] = new_state[nx][ny], new_state[x][y]
            neighbors.append(tuple(tuple(row) for row in new_state))

    return neighbors

def reconstruct_path(came_from, current):

    path = [current]
    while current in came_from:
        current = came_from[current]
        path.append(current)
    path.reverse()
    return path

def a_star(start, goal):
    open_list = []
    heapq.heappush(open_list, (0 + misplaced_tile(start, goal), 0, start))

    g_score = {start: 0}
    came_from = {}

    visited = set()
```

```
    while open_list:
        _, g, current = heapq.heappop(open_list)

        if current == goal:
            path = reconstruct_path(came_from, current)
            return path, g

        visited.add(current)

        for neighbor in generate_neighbors(current):
            if neighbor in visited:
                continue
            tentative_g = g_score[current] + 1

            if tentative_g < g_score.get(neighbor, float('inf')):
                came_from[neighbor] = current
                g_score[neighbor] = tentative_g
                f_score = tentative_g + misplaced_tile(neighbor, goal)  # f(n) = g(n) + h(n)

                heapq.heappush(open_list, (f_score, tentative_g, neighbor))

    return None, None

def print_state(state):

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

def get_state_from_user(prompt):

    state = []
    for i in range(3):
        row = input(f"{prompt} row {i+1} (space-separated): ")
        state.append(tuple(map(int, row.split())))
    return tuple(state)

if __name__ == "__main__":
    print("Enter the initial state:")
    start_state = get_state_from_user("Initial state")
    print("\nEnter the goal state:")
    goal_state = get_state_from_user("Goal state")

    print("\nInitial State:")
    print_state(start_state)

    print("\nGoal State:")
```

```
    print_state(goal_state)

    solution, cost = a_star(start_state, goal_state)

    if solution:
        print(f"\nSolution found with cost: {cost}")
        print("Steps:")
        for step in solution:
            print_state(step)
    else:
        print("\nNo solution found.")
```

Output:

```
Enter the initial state:
Initial state row 1 (space-separated): 2 8 3
Initial state row 2 (space-separated): 1 6 4
Initial state row 3 (space-separated): 7 0 5

Enter the goal state:
Goal state row 1 (space-separated): 1 2 3
Goal state row 2 (space-separated): 8 0 4
Goal state row 3 (space-separated): 7 6 5

Initial State:
(2, 8, 3)
(1, 6, 4)
(7, 0, 5)

Goal State:
(1, 2, 3)
(8, 0, 4)
(7, 6, 5)

Solution found with cost: 5
Steps:
(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)

```
import heapq

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

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

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

def generate_neighbors(state):
    neighbors = []
    x, y = find_blank(state)
    directions = [(0, 1), (0, -1), (1, 0), (-1, 0)]

    for dx, dy in directions:
        nx, ny = x + dx, y + dy
        if 0 <= nx < 3 and 0 <= ny < 3:
            new_state = [list(row) for row in state]
            new_state[x][y], new_state[nx][ny] = new_state[nx][ny], new_state[x][y]
            neighbors.append(tuple(tuple(row) for row in new_state))
```

```
    return neighbors

def reconstruct_path(came_from, current):
    path = [current]
    while current in came_from:
        current = came_from[current]
        path.append(current)
    path.reverse()
    return path

def a_star(start, goal):
    open_list = []
    heapq.heappush(open_list, (0 + manhattan_distance(start, goal), 0, start))

    g_score = {start: 0}
    came_from = {}

    visited = set()

    while open_list:
        _, g, current = heapq.heappop(open_list)

        if current == goal:
            path = reconstruct_path(came_from, current)
            return path, g

        visited.add(current)

        for neighbor in generate_neighbors(current):
            if neighbor in visited:
                continue
            tentative_g = g_score[current] + 1

            if tentative_g < g_score.get(neighbor, float('inf')):
                came_from[neighbor] = current
                g_score[neighbor] = tentative_g
                f_score = tentative_g + manhattan_distance(neighbor, goal)

                heapq.heappush(open_list, (f_score, tentative_g, neighbor))

    return None, None

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

def get_state_from_user(prompt):
```

```
    state = []
    for i in range(3):
        row = input(f"{prompt} row {i+1} (space-separated): ")
        state.append(tuple(map(int, row.split())))
    return tuple(state)

if __name__ == "__main__":
    print("Enter the initial state:")
    start_state = get_state_from_user("Initial state")
    print("\nEnter the goal state:")
    goal_state = get_state_from_user("Goal state")

    print("\nInitial State:")
    print_state(start_state)

    print("\nGoal State:")
    print_state(goal_state)

    solution, cost = a_star(start_state, goal_state)

    if solution:
        print(f"\nSolution found with cost: {cost}")
        print("Steps:")
        for step in solution:
            print_state(step)
    else:
        print("\nNo solution found.")
```

Output:

```
Enter the initial state:
Initial state row 1 (space-separated): 2 8 3
Initial state row 2 (space-separated): 1 6 4
Initial state row 3 (space-separated): 7 0 5

Enter the goal state:
Goal state row 1 (space-separated): 1 2 3
Goal state row 2 (space-separated): 8 0 4
Goal state row 3 (space-separated): 7 6 5

Initial State:
(2, 8, 3)
(1, 6, 4)
(7, 0, 5)


Goal State:
(1, 2, 3)
(8, 0, 4)
```

(7, 6, 5)

Solution found with cost: 5
Steps:
(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)

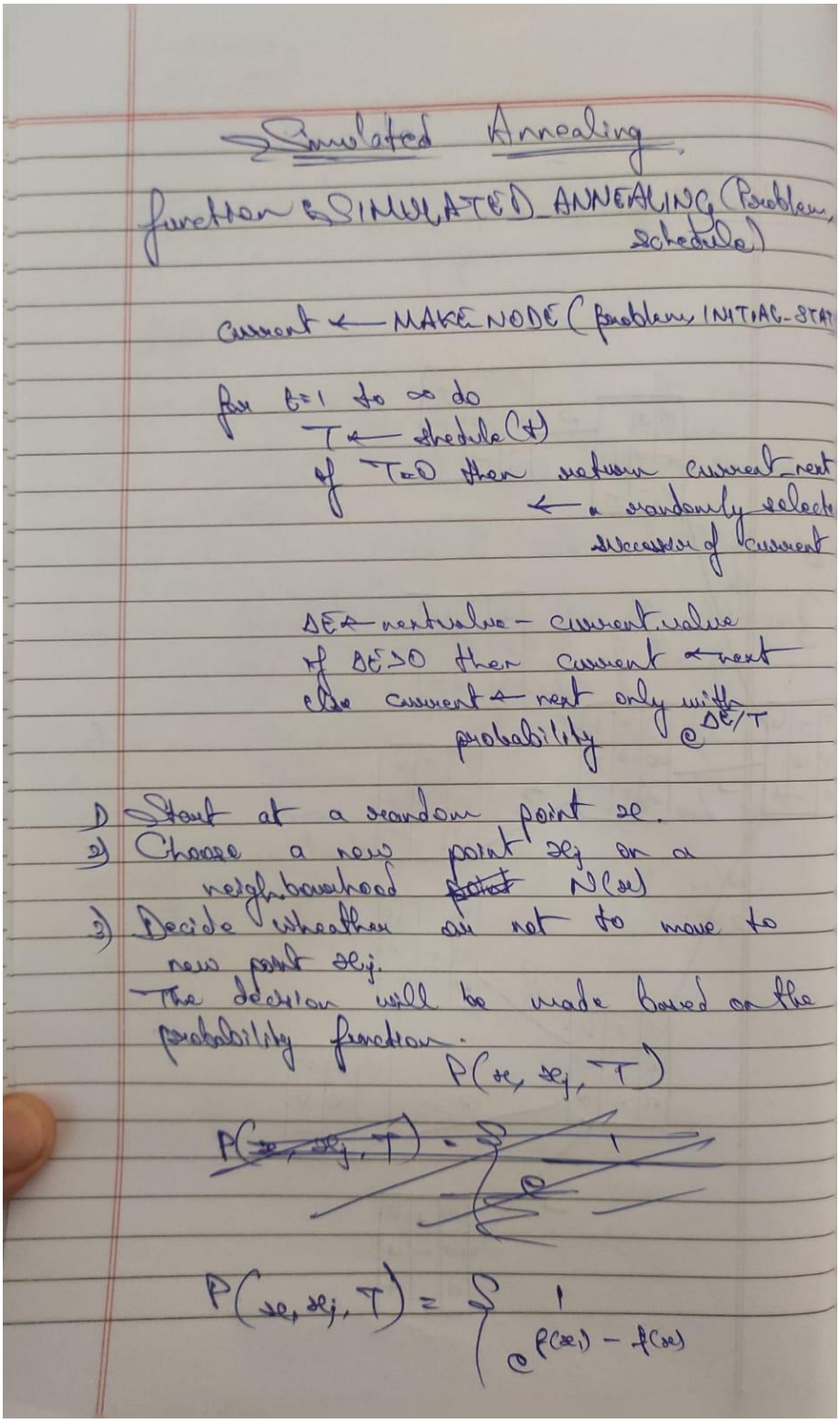
(1, 2, 3)
(0, 8, 4)
(7, 6, 5)

(1, 2, 3)
(8, 0, 4)
(7, 6, 5)

Program 4

Implement Hill Climbing search algorithm to solve N-Queens problem

Algorithm:



Bafna Gold
Date: Page:

For 8 queens problem:

1) Initialize: Start with arrangement of queens, set initial temp & cooling parameters.

2) Evaluate fitness: Calculate no. of conflicts by swapping 2 queens. Calculate the fitness.

3) Generate Neighbour: Create a neighbour by swapping 2 queens. Calculate its fitness.

4) Accept or reject neighbour.

5) Gradually reduce the temperature & repeat until the temp is very low.

6) Return the best arrangement found.

Output: Best state found [4 7 5 0 6 3 7 2].
Number of conflicts = 2

Code:

```
from random import randint
N = int(input("Enter the number of queens:"))

def configureRandomly(board, state):
	for i in range(N):
		state[i] = randint(0, 100000) % N;
		board[state[i]][i] = 1;

def printBoard(board):
	for i in range(N):
		print(*board[i])

def printState( state):
	print(*state)

def compareStates(state1, state2):
	for i in range(N):
		if (state1[i] != state2[i]):
			return False;

	return True;

def fill(board, value):
	for i in range(N):
		for j in range(N):
			board[i][j] = value;

def calculateObjective( board, state):
	attacking = 0;
	for i in range(N):
		row = state[i]
		col = i - 1;
		while (col >= 0 and board[row][col] != 1) :
			col -= 1
		if (col >= 0 and board[row][col] == 1) :
			attacking += 1;

		row = state[i]
		col = i + 1;
		while (col < N and board[row][col] != 1):
			col += 1;
```

```
            if (col < N and board[row][col] == 1) :
                attacking += 1;
            row = state[i] - 1
            col = i - 1;
            while (col >= 0 and row >= 0 and board[row][col] != 1) :
                col-= 1;
                row-= 1;

            if (col >= 0 and row >= 0 and board[row][col] == 1) :
                attacking+= 1;

            row = state[i] + 1
            col = i + 1;
            while (col < N and row < N and board[row][col] != 1) :
                col+= 1;
                row+= 1;

            if (col < N and row < N and board[row][col] == 1) :
                attacking += 1;

            row = state[i] + 1
            col = i - 1;
            while (col >= 0 and row < N and board[row][col] != 1) :
                col -= 1;
                row += 1;

            if (col >= 0 and row < N and board[row][col] == 1) :
                attacking += 1;

            row = state[i] - 1
            col = i + 1;
            while (col < N and row >= 0 and board[row][col] != 1) :
                col += 1;
                row -= 1;

            if (col < N and row >= 0 and board[row][col] == 1) :
                attacking += 1;
        return int(attacking / 2);

def generateBoard( board, state):
        fill(board, 0);
```

```
        for i in range(N):
                board[state[i]][i] = 1;

def copyState( state1, state2):

        for i in range(N):
                state1[i] = state2[i];

def getNeighbour(board, state):

        opBoard = [[0 for _ in range(N)] for _ in range(N)]
        opState = [0 for _ in range(N)]

        copyState(opState, state);
        generateBoard(opBoard, opState);

        opObjective = calculateObjective(opBoard, opState);

        NeighbourBoard = [[0 for _ in range(N)] for _ in range(N)]

        NeighbourState = [0 for _ in range(N)]
        copyState(NeighbourState, state);
        generateBoard(NeighbourBoard, NeighbourState);

        for i in range(N):
                for j in range(N):
                        if (j != state[i]) :
                                NeighbourState[i] = j;
                                NeighbourBoard[NeighbourState[i]][i] = 1;
                                NeighbourBoard[state[i]][i] = 0;
                                temp = calculateObjective( NeighbourBoard, NeighbourState);

                                if (temp <= opObjective) :
                                        opObjective = temp;
                                        copyState(opState, NeighbourState);
                                        generateBoard(opBoard, opState);

                                NeighbourBoard[NeighbourState[i]][i] = 0;
                                NeighbourState[i] = state[i];
                                NeighbourBoard[state[i]][i] = 1;
```

```
        copyState(state, opState);
        fill(board, 0);
        generateBoard(board, state);

def hillClimbing(board, state):


        neighbourBoard = [[0 for _ in range(N)] for _ in range(N)]
        neighbourState = [0 for _ in range(N)]

        copyState(neighbourState, state);
        generateBoard(neighbourBoard, neighbourState);

        while True:

                # Copying the neighbour board and
                # state to the current board and
                # state, since a neighbour
                # becomes current after the jump.

                copyState(state, neighbourState);
                generateBoard(board, state);

                # Getting the optimal neighbour

                getNeighbour(neighbourBoard, neighbourState);

                if (compareStates(state, neighbourState)) :

                        printBoard(board);
                        break;

                elif (calculateObjective(board, state) == calculateObjective(
neighbourBoard,neighbourState)):

                        # Random neighbour
                        neighbourState[randint(0, 100000) % N] = randint(0, 100000) % N;
                        generateBoard(neighbourBoard, neighbourState);

# Driver code
state = [0] * N
board = [[0 for _ in range(N)] for _ in range(N)]
```

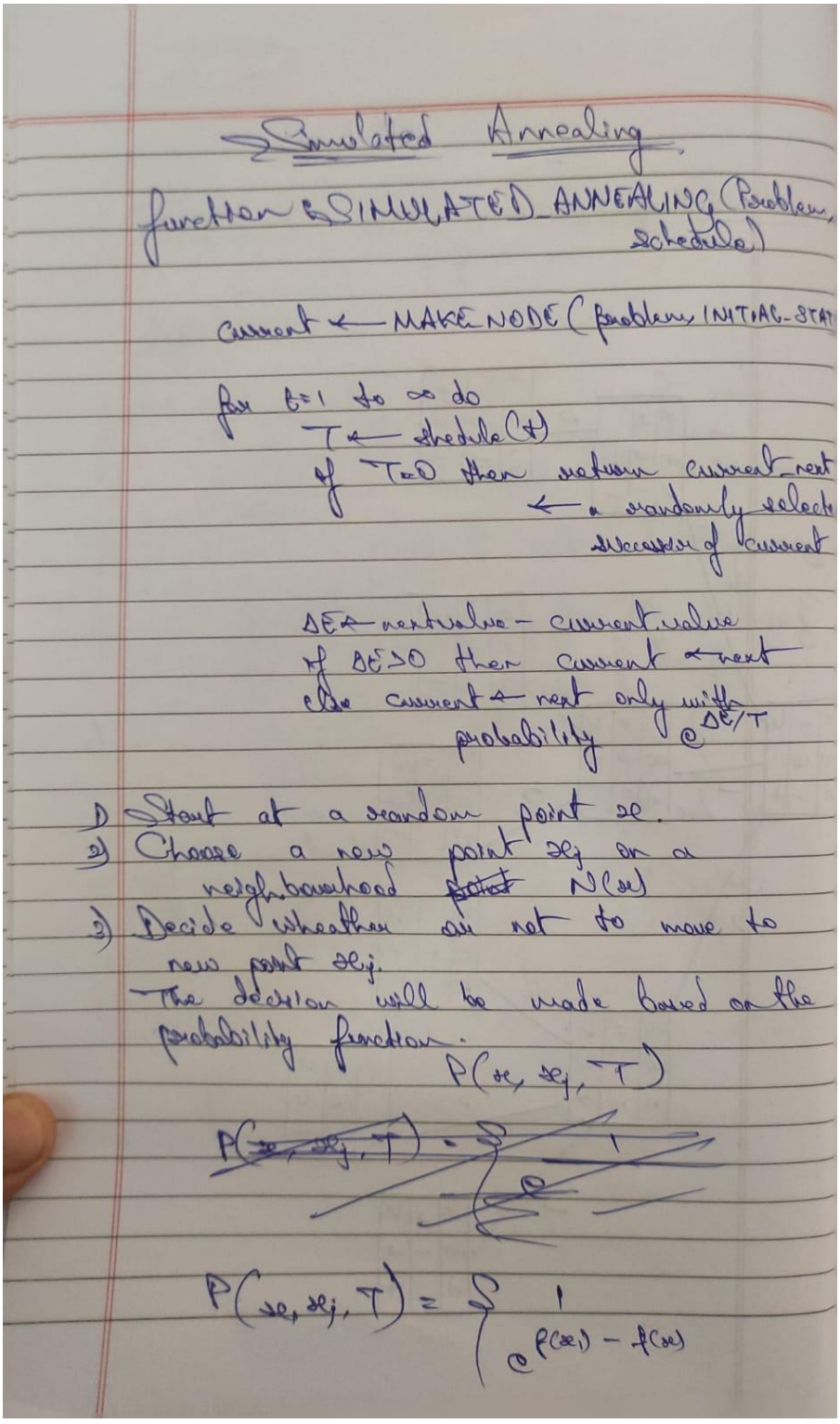
```
configureRandomly(board, state);
hillClimbing(board, state);
```

Output:

```
Enter the number of queens:8
0 0 0 0 1 0 0 0
0 1 0 0 0 0 0 0
0 0 0 0 0 0 0 1
1 0 0 0 0 0 0 0
0 0 0 1 0 0 0 0
0 0 0 0 0 0 1 0
0 0 1 0 0 0 0 0
0 0 0 0 0 1 0 0
```

Program 5
Simulated Annealing to Solve 8-Queens problem

Algorithm:



Bafna Gold
Date: Page:

8 queens problem :

1) Initialize : Start with arrangement of queens, set initial temp & cooling parameters.

2) Evaluate fitness : Calculate no. of conflicts by swapping 2 queens. Calculate the fitness.

3) Generate Neighbour : Create a neighbour by swapping 2 queens. Calculate its fitness.

4) Accept or reject neighbour.

5) Gradually reduce the temperature & repeat until the temp is very low.

6) Return the best arrangement found.

Output : Best state found [4 7 5 0 6 3 7 2].
Number of conflicts = 2

Code:

```
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)
```

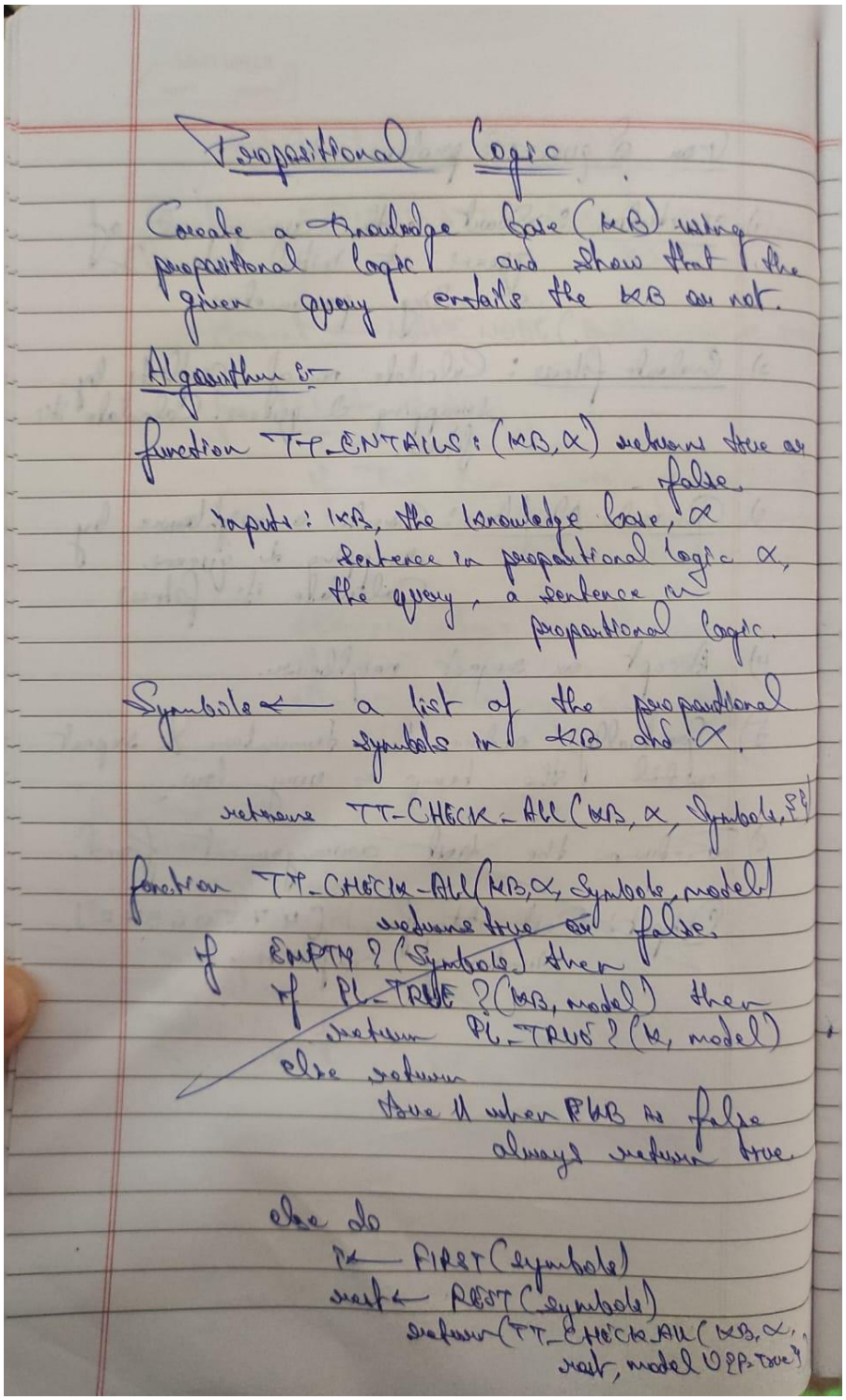
Output:

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

Program 6

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

Algorithm:



Propositional Logic

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

Algorithm:-

function TT_ENTAILS : (KB, α) returns true or false.

Inputs : KB, the knowledge base, α sentence in propositional logic α, the query, a sentence in propositional logic.

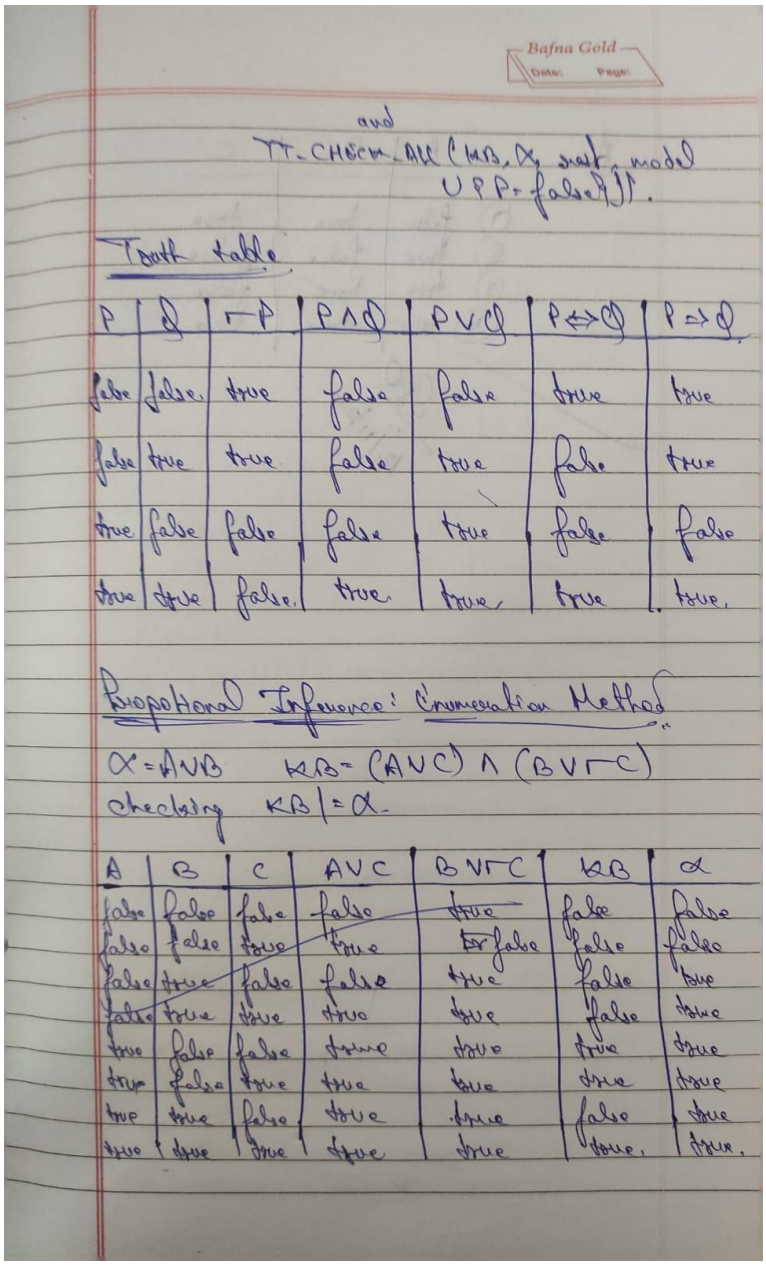
Symbols ← a list of the propositional symbols in KB and α.

return TT-CHECK-ALL(KB, α, Symbols, {})

function TT_CHECK-ALL(KB, α, Symbols, model)
returns true or false.

if EMPTY?(Symbols) then
if PL-TRUE?(KB, model) then
return PL-TRUE?(α, model)
else return
true if when PKB is false
always return true.

else do
P ← FIRST(Symbols)
rest ← REST(Symbols)
return(TT-CHECK-ALL(KB, α,
rest, model ∪ {P=true})



and

TT-CHECK-ALL(KB, α, sent, model)
UPP = false ∀ P.

Truth table

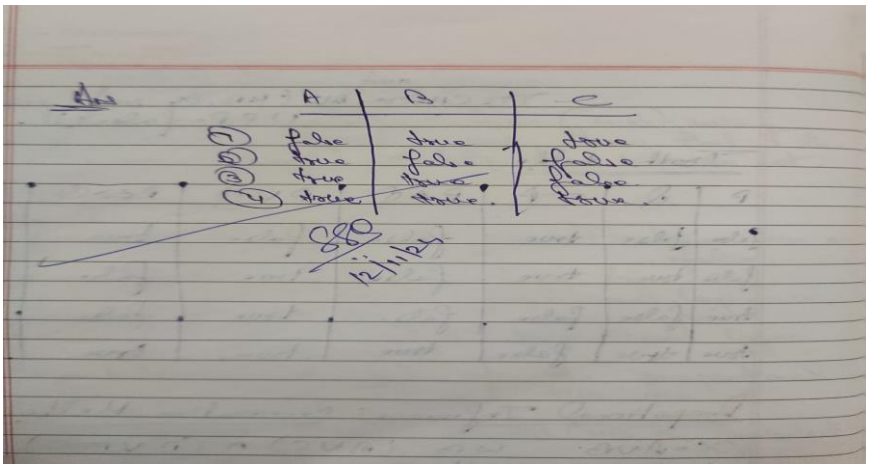
P	Q	¬P	P∧Q	P∨Q	P⇔Q	P⇒Q
false	false	true	false	false	true	true
false	true	true	false	true	false	true
true	false	false	false	true	false	false
true	true	false	true	true	true	true

Propositional Inference: Enumeration Method

α = A∨B KB = (A∨C) ∧ (B∨¬C)

checking KB |= α

A	B	C	A∨C	B∨¬C	KB	α
false	false	false	false	true	false	false
false	false	true	true	false	false	false
false	true	false	false	true	false	true
false	true	true	true	true	false	true
true	false	false	true	true	true	true
true	false	true	true	true	true	true
true	true	false	true	true	false	true
true	true	true	true	true	true	true



	A	B	C
①	false	true	true
②	true	false	false
③	true	true	false
④	true	true	true

12/11/24

Code:

```
import itertools

def evaluate_formula(formula, valuation):
    """
    Evaluate the propositional formula under the given truth assignment (valuation).
    The formula is a string of logical operators like 'AND', 'OR', 'NOT', and can contain variables 'A', 'B', 'C'.
    """
    # Create a local environment (dictionary) for variable assignments
    env = {var: valuation[i] for i, var in enumerate(['A', 'B', 'C'])}

    # Replace logical operators with Python equivalents
    formula = formula.replace('AND', 'and').replace('OR', 'or').replace('NOT', 'not')

    # Replace variables in the formula with their corresponding truth values
    for var in env:
        formula = formula.replace(var, str(env[var]))

    # Evaluate the formula and return the result (True or False)
    try:
        return eval(formula)
    except Exception as e:
        raise ValueError(f"Error in evaluating formula: {e}")

def truth_table(variables):
    """
    Generate all possible truth assignments for the given variables.
    """
    return list(itertools.product([False, True], repeat=len(variables)))

def entails(KB, alpha):
    """
    Decide if KB entails alpha using a truth-table enumeration algorithm.
    KB is a propositional formula (string), and alpha is another propositional formula (string).
    """
    # Generate all possible truth assignments for A, B, and C
    assignments = truth_table(['A', 'B', 'C'])

    print(f"{'A':<10}{'B':<10}{'C':<10}{'KB':<15}{'alpha':<15}{'KB entails alpha?'}")  # Header for the truth table
    print("-" * 70)  # Separator for readability

    for assignment in assignments:
        # Evaluate KB and alpha under the current assignment
        KB_value = evaluate_formula(KB, assignment)
        alpha_value = evaluate_formula(alpha, assignment)

        # Print the current truth assignment and the results for KB and alpha
```

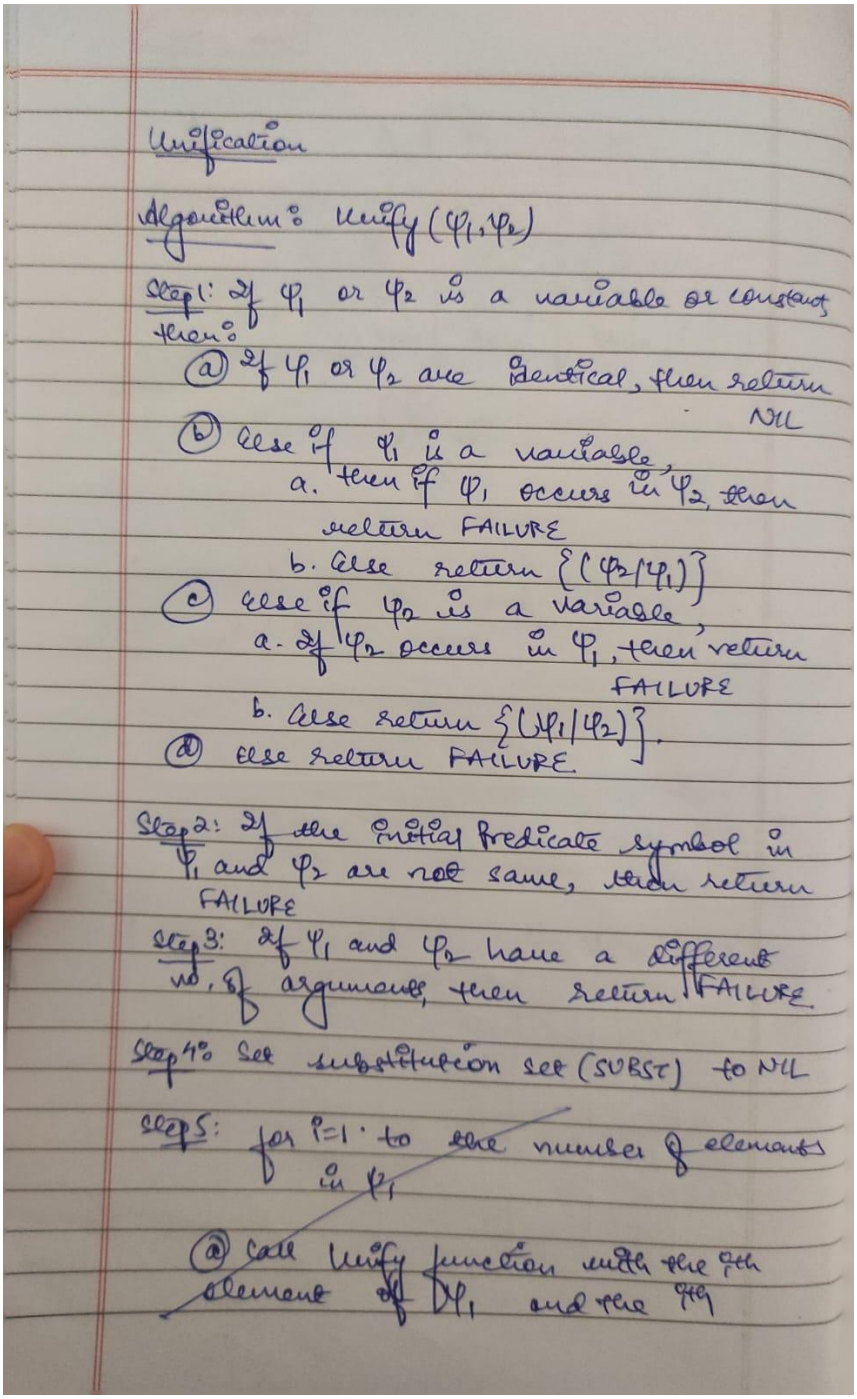
```
print(f"{str(assignment[0]):<10}{str(assignment[1]):<10}{str(assignment[2]):<10}{str(KB_value):<15}{str(alpha_value):<15}{'Yes' if KB_value and alpha_value else 'No'}")

        # If KB is true and alpha is false, then KB does not entail alpha
        if KB_value and not alpha_value:
            return False

    # If no counterexample was found, then KB entails alpha
    return True

# Define the formulas for KB and alpha
alpha = 'A OR B'
KB = '(A OR C) AND (B OR NOT C)'

# Check if KB entails alpha
result = entails(KB, alpha)

# Print the final result of entailment
print(f"\nDoes KB entail alpha? {result}")
```

Output:

```
A       B       C       KB          alpha       KB entails alpha?
------------------------------------------------------------------------
False   False   False   False       False       No
False   False   True    False       False       No
False   True    False   False       True        No
False   True    True    True        True        Yes
True    False   False   True        True        Yes
True    False   True    False       True        No
True    True    False   True        True        Yes
True    True    True    True        True        Yes

Does KB entail alpha? True
```

Program 7
Implement unification in first order logic

Algorithm:



Unification

Algorithm: Unify(ψ_1, ψ_2)

Step 1: If ψ_1 or ψ_2 is a variable or constant then:

(a) If ψ_1 or ψ_2 are identical, then return NIL

(b) Else if ψ_1 is a variable,
  a. then if ψ_1 occurs in ψ_2, then return FAILURE
  b. Else return $\{(\psi_2/\psi_1)\}$

(c) Else if ψ_2 is a variable,
  a. If ψ_2 occurs in ψ_1, then return FAILURE
  b. Else return $\{(\psi_1/\psi_2)\}$.

(d) Else return FAILURE.

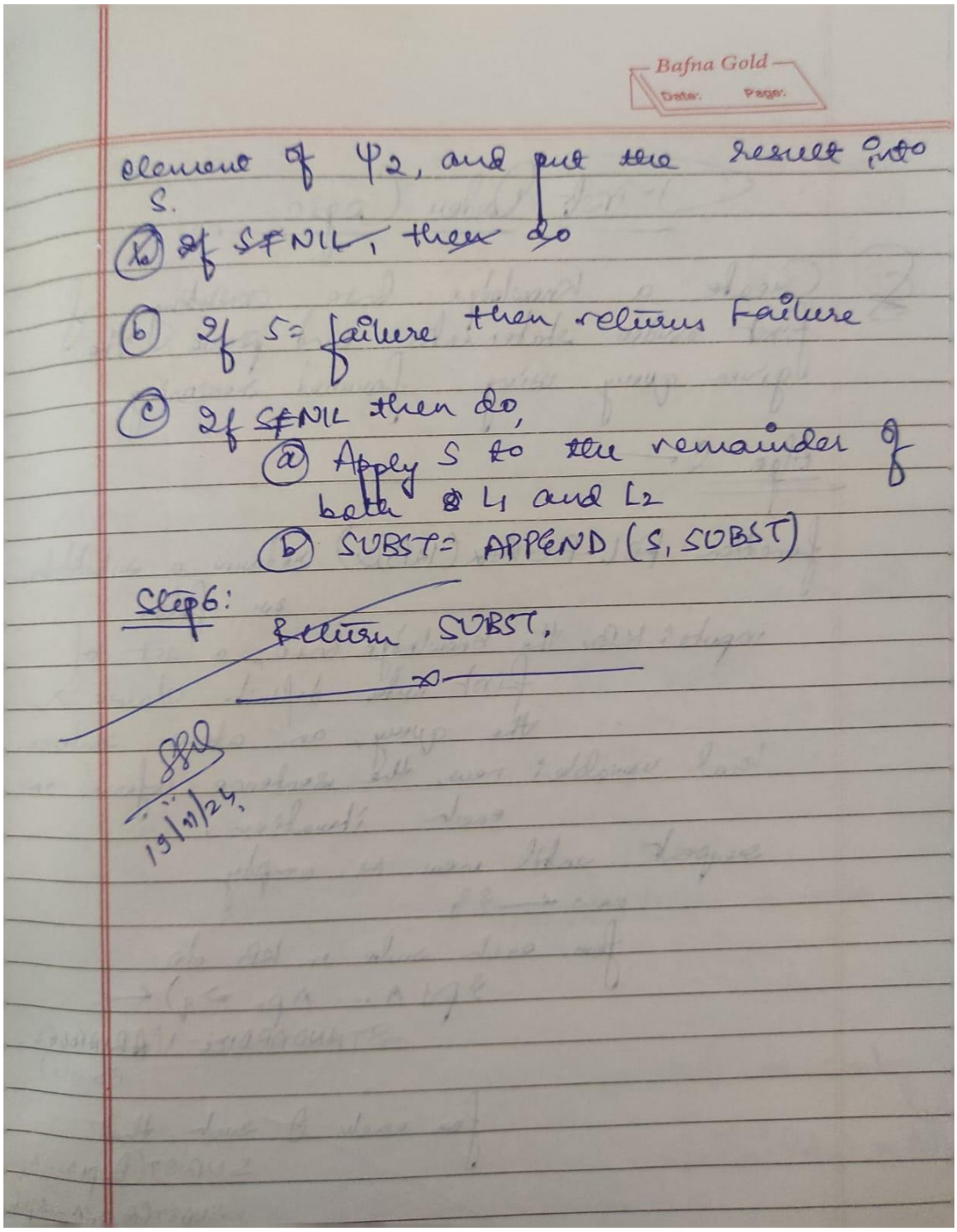
Step 2: If the initial predicate symbol in ψ_1 and ψ_2 are not same, then return FAILURE

Step 3: If ψ_1 and ψ_2 have a different no. of arguments, then return FAILURE.

Step 4: Set substitution set (SUBST) to NIL

Step 5: for i=1 to the number of elements in ψ_1

(a) Call Unify function with the ith element of ψ_1 and the ith


Bafna Gold
Date: Page:
elements of Ψ2, and put the result into S.
(a) If S≠NIL, then do
(b) If S= failure then returns Failure
(c) If S≠NIL then do,
(a) Apply S to the remainder of both L1 and L2
(b) SUBST= APPEND (S, SUBST)
Step6:
Return SUBST.
19/11/24.

Code:

```
class Term:
    def __init__(self, symbol, args=None):
        self.symbol = symbol
        self.args = args if args else []

    def __str__(self):
        if not self.args:
            return str(self.symbol)
        return f"{self.symbol}({','.join(str(arg) for arg in self.args)})"

    def is_variable(self):
        return isinstance(self.symbol, str) and self.symbol.isupper() and not self.args

def occurs_check(var, term, substitution):
    """Check if variable occurs in term"""
    if term.is_variable():
        if term.symbol in substitution:
            return occurs_check(var, substitution[term.symbol], substitution)
        return var.symbol == term.symbol
    return any(occurs_check(var, arg, substitution) for arg in term.args)

def substitute(term, substitution):
    """Apply substitution to term"""
    if term.is_variable() and term.symbol in substitution:
        return substitute(substitution[term.symbol], substitution)
    if not term.args:
        return term
    return Term(term.symbol, [substitute(arg, substitution) for arg in term.args])

def unify(term1, term2, substitution=None, iteration=1):
    """Unify two terms with detailed iteration steps"""
    if substitution is None:
        substitution = {}

    print(f"\nIteration {iteration}:")
    print(f"Attempting to unify: {term1} and {term2}")
    print(f"Current substitution: {', '.join(f'{k}->{v}' for k,v in substitution.items()) or 'empty'}")

    term1 = substitute(term1, substitution)
    term2 = substitute(term2, substitution)

    if term1.symbol == term2.symbol and not term1.args and not term2.args:
        print("Terms are identical - no substitution needed")
        return substitution

    if term1.is_variable():
        if occurs_check(term1, term2, substitution):
```

```
            print(f"Occurs check failed: {term1.symbol} occurs in {term2}")
            return None
        substitution[term1.symbol] = term2
        print(f"Added substitution: {term1.symbol} -> {term2}")
        return substitution

    if term2.is_variable():
        if occurs_check(term2, term1, substitution):
            print(f"Occurs check failed: {term2.symbol} occurs in {term1}")
            return None
        substitution[term2.symbol] = term1
        print(f"Added substitution: {term2.symbol} -> {term1}")
        return substitution

    if term1.symbol != term2.symbol or len(term1.args) != len(term2.args):
        print(f"Unification failed: Different predicates or argument lengths")
        return None

    for arg1, arg2 in zip(term1.args, term2.args):
        result = unify(arg1, arg2, substitution, iteration + 1)
        if result is None:
            return None
        substitution = result

    return substitution

def parse_term(s):
    """Parse terms like P(X,f(Y)) or X"""
    s = s.strip()
    if '(' not in s:
        return Term(s)

    pred = s[:s.index('(')]
    args_str = s[s.index('(')+1:s.rindex(')')]

    args = []
    current = ''
    depth = 0
    for c in args_str:
        if c == '(' or c == '[':
            depth += 1
        elif c == ')' or c == ']':
            depth -= 1
        elif c == ',' and depth == 0:
            args.append(parse_term(current.strip()))
            current = ''
            continue
        current += c
```

```
    if current:
        args.append(parse_term(current.strip()))

    return Term(pred, args)

def print_examples():
    print("\nExample format:")
    print("1. Simple terms: P(X,Y)")
    print("2. Nested terms: P(f(X),g(Y))")
    print("3. Mixed terms: Knows(John,X)")
    print("4. Complex nested terms: P(f(g(X)),h(Y,Z))")
    print("\nNote: Use capital letters for variables (X,Y,Z) and lowercase for constants and predicates.")

def validate_input(expr):
    """Basic validation for input expressions"""
    if not expr:
        return False

    # Check balanced parentheses
    count = 0
    for char in expr:
        if char == '(':
            count += 1
        elif char == ')':
            count -= 1
        if count < 0:
            return False
    return count == 0

def main():
    while True:
        print("\n=== First Order Predicate Logic Unification ===")
        print("1. Start Unification")
        print("2. Show Examples")
        print("3. Exit")

        choice = input("\nEnter your choice (1-3): ")

        if choice == '1':
            print("\nEnter two expressions to unify.")
            print_examples()

            while True:
                expr1 = input("\nEnter first expression (or 'back' to return): ")
                if expr1.lower() == 'back':
                    break

                if not validate_input(expr1):
```

```
            print("Invalid expression! Please check the format and try again.")
            continue

        expr2 = input("Enter second expression: ")
        if not validate_input(expr2):
            print("Invalid expression! Please check the format and try again.")
            continue

        try:
            term1 = parse_term(expr1)
            term2 = parse_term(expr2)

            print("\nUnification Process:")
            result = unify(term1, term2)

            print("\nFinal Result:")
            if result is None:
                print("Unification failed!")
            else:
                print("Unification successful!")
                print("Final substitutions:", ', '.join(f'{k}->{v}' for k,v in result.items()))

            retry = input("\nTry another unification? (y/n): ")
            if retry.lower() != 'y':
                break

        except Exception as e:
            print(f"Error processing expressions: {str(e)}")
            print("Please check your input format and try again.")

    elif choice == '2':
        print("\n=== Example Expressions ===")
        print("1. P(X,h(Y)) and P(a,f(Z))")
        print("2. P(f(a),g(Y)) and P(X,X)")
        print("3. Knows(John,X) and Knows(X,Elisabeth)")
        print("\nPress Enter to continue...")
        input()

    elif choice == '3':
        print("\nThank you for using the Unification Program!")
        break

    else:
        print("\nInvalid choice! Please enter 1, 2, or 3.")

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

Output:

```
=== First Order Predicate Logic Unification ===
1. Start Unification
2. Show Examples
3. Exit

Enter your choice (1-3): 1

Enter two expressions to unify.

Example format:
1. Simple terms: P(X,Y)
2. Nested terms: P(f(X),g(Y))
3. Mixed terms: Knows(John,X)
4. Complex nested terms: P(f(g(X)),h(Y,Z))

Note: Use capital letters for variables (X,Y,Z) and lowercase for constants and predicates.

Enter first expression (or 'back' to return): p(X,f(Y))
Enter second expression: p(a,f(g(x)))

Unification Process:

Iteration 1:
Attempting to unify: p(X,f(Y)) and p(a,f(g(x)))
Current substitution: empty

Iteration 2:
Attempting to unify: X and a
Current substitution: empty
Added substitution: X -> a

Iteration 2:
Attempting to unify: f(Y) and f(g(x))
Current substitution: X->a

Iteration 3:
Attempting to unify: Y and g(x)
Current substitution: X->a
Added substitution: Y -> g(x)

Final Result:
Unification successful!
Final substitutions: X->a, Y->g(x)

Try another unification? (y/n): y

Enter first expression (or 'back' to return): q(a,g(X,a),f(Y))
```

Enter second expression: q(a,g(f(h),a),X)

Unification Process:

Iteration 1:
Attempting to unify: q(a,g(X,a),f(Y)) and q(a,g(f(h),a),X)
Current substitution: empty

Iteration 2:
Attempting to unify: a and a
Current substitution: empty
Terms are identical - no substitution needed

Iteration 2:
Attempting to unify: g(X,a) and g(f(h),a)
Current substitution: empty

Iteration 3:
Attempting to unify: X and f(h)
Current substitution: empty
Added substitution: X -> f(h)

Iteration 3:
Attempting to unify: a and a
Current substitution: X->f(h)
Terms are identical - no substitution needed

Iteration 2:
Attempting to unify: f(Y) and X
Current substitution: X->f(h)

Iteration 3:
Attempting to unify: Y and h
Current substitution: X->f(h)
Added substitution: Y -> h

Final Result:
Unification successful!
Final substitutions: X->f(h), Y->h

Try another unification? (y/n): y

Enter first expression (or 'back' to return): p(b,X,f(g(Z)))
Enter second expression: p(Z,f(Y),f(Y))

Unification Process:

Iteration 1:

Attempting to unify: p(b,X,f(g(Z))) and p(Z,f(Y),f(Y))
Current substitution: empty

Iteration 2:
Attempting to unify: b and Z
Current substitution: empty
Added substitution: Z -> b

Iteration 2:
Attempting to unify: X and f(Y)
Current substitution: Z->b
Added substitution: X -> f(Y)

Iteration 2:
Attempting to unify: f(g(Z)) and f(Y)
Current substitution: Z->b, X->f(Y)

Iteration 3:
Attempting to unify: g(b) and Y
Current substitution: Z->b, X->f(Y)
Added substitution: Y -> g(b)

Final Result:
Unification successful!
Final substitutions: Z->b, X->f(Y), Y->g(b)

Try another unification? (y/n): y

Enter first expression (or 'back' to return): p(f(a),g(Y))
Enter second expression: p(X,X)

Unification Process:

Iteration 1:
Attempting to unify: p(f(a),g(Y)) and p(X,X)
Current substitution: empty

Iteration 2:
Attempting to unify: f(a) and X
Current substitution: empty
Added substitution: X -> f(a)

Iteration 2:
Attempting to unify: g(Y) and X
Current substitution: X->f(a)
Unification failed: Different predicates or argument lengths

Final Result:

Unification failed!

Try another unification? (y/n): n

=== First Order Predicate Logic Unification ===
1. Start Unification
2. Show Examples
3. Exit

Enter your choice (1-3): 2

=== Example Expressions ===
1. P(X,h(Y)) and P(a,f(Z))
2. P(f(a),g(Y)) and P(X,X)
3. Knows(John,X) and Knows(X,Elisabeth)

Press Enter to continue...

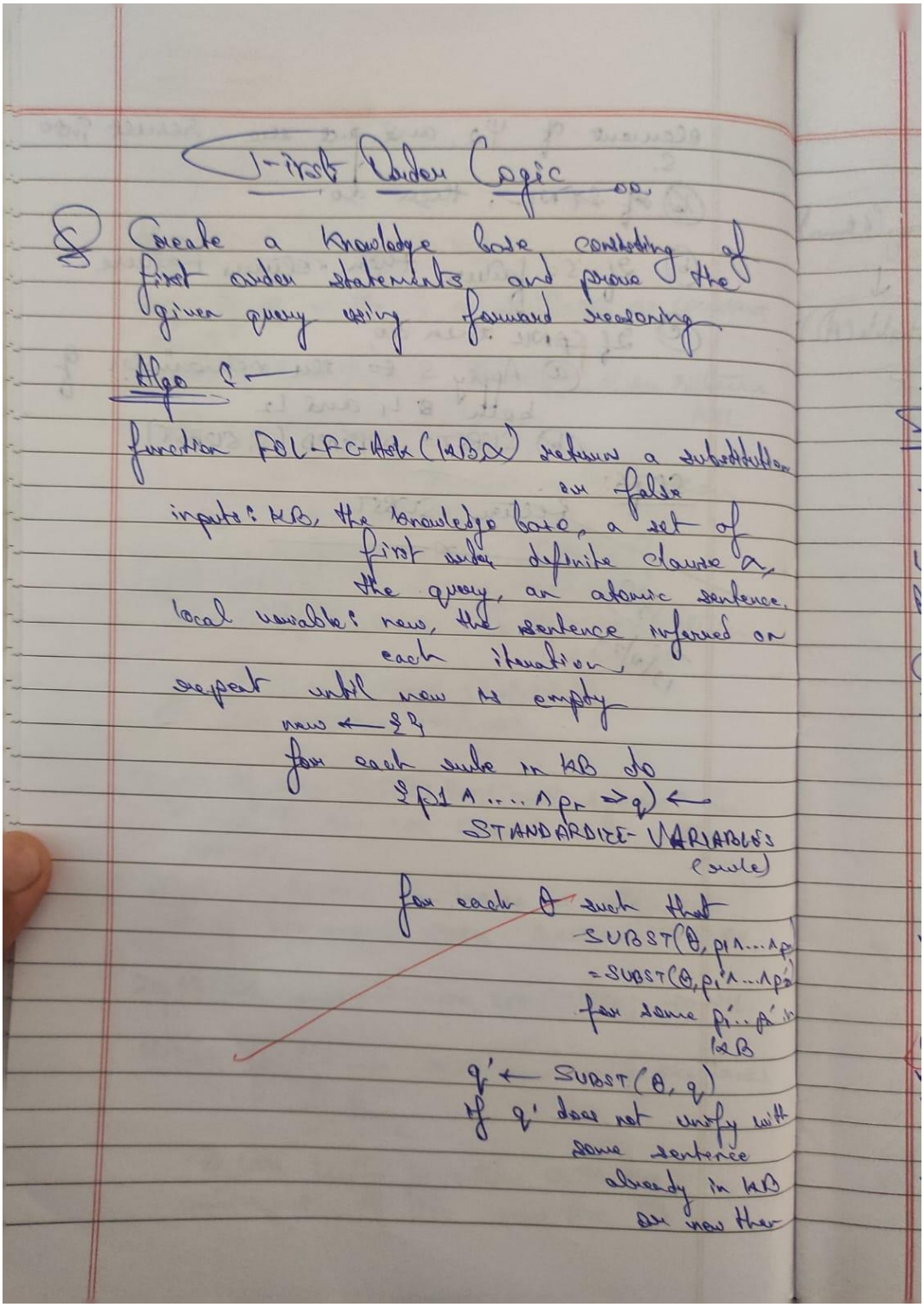
=== First Order Predicate Logic Unification ===
1. Start Unification
2. Show Examples
3. Exit

Enter your choice (1-3): 3

Program 8

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

Algorithm:



add q' to nbus
θ ← unify(q', α)
if θ is not fail
then subst θ

add new to KB
return false.

Representation in FOL

It is a crime for an American to sell weapon to hostile nations

Let's say p, q and r are variable

American(p) ∧ Weapon(q) ∧ Sells(p, q, r) ∧ Hostile(r) ⇒ Criminal(p)

Country A has some missiles

∃ x Owns(A, x) ∧ Missile(x)

Existential instantiation introducing a new constant T1 :

Owns(A, T1)

Missile(T1)

All of the missiles were sold to country A by Robert

∀ x Missiles(x) ∧ Owns(A, x) ⇒ Sells(Robert, x, A)

Missiles are weapons

Missiles(x) ⇒ Weapon(x)

Enemy of America is known as hostile

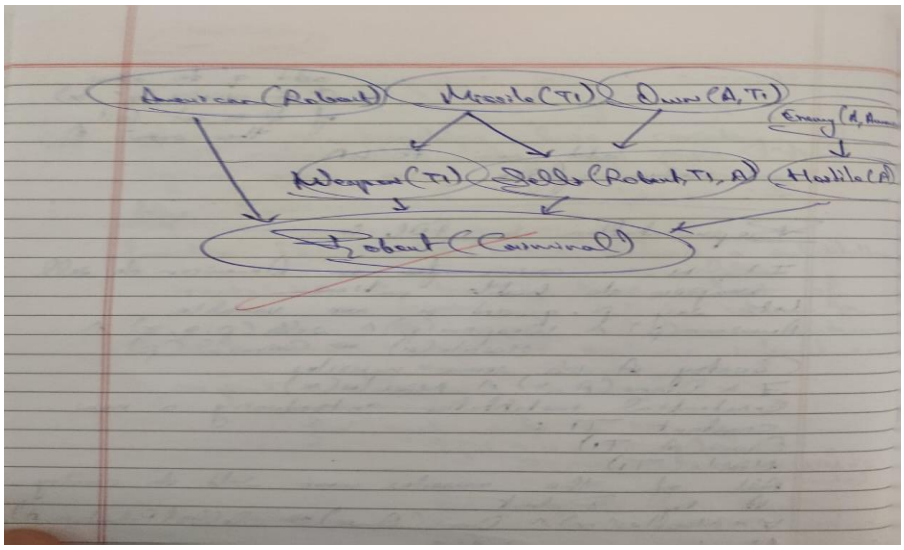
∀ x Enemy(x, America) ⇒ Hostile(x)

Robert is an American

American(Robert)

The country A, an enemy of American

Enemy(A, America)



Code:

```
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, query):
        """
        Applies forward reasoning to infer new facts based on rules and initial facts.

        Parameters:
            query (str): The fact to verify if it can be inferred.

        Returns:
            bool: True if the query can be inferred, False otherwise.
        """
        applied_rules = True

        while applied_rules:
            applied_rules = False
            for condition, result in self.rules:
                # Check if all conditions are met in the current facts
                if condition.issubset(self.facts) and result not in self.facts:
                    self.facts.add(result)  # Add the inferred result
                    applied_rules = True
                    print(f"Applied rule: {condition} -> {result}")
                    # If the query is inferred, return True immediately
                    if query in self.facts:
                        return True

        # Return whether the query can be inferred from the facts
        return query in self.facts


# Define the Knowledge Base (KB) with rules as (condition, result)
rules = [
    ({"American(Robert)", "Missile(m1)", "Owns(CountryA, m1)"}, "Sells(Robert, m1, CountryA)"),  # Sells(Robert, m1, CountryA) based on facts
    ({"Sells(Robert, m1, CountryA)", "American(Robert)", "Hostile(CountryA)"}, "Criminal(Robert)"),  # Criminal inference
```

```
]

# Define initial facts
facts = {
    "American(Robert)",
    "Hostile(CountryA)",
    "Missile(m1)",
    "Owns(CountryA, m1)",
}

# Query
query = "Criminal(Robert)"

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

# Final output
print("\nFinal facts:")
print(reasoner.facts)
print(f"\nQuery '{query}' inferred: {result}")
```

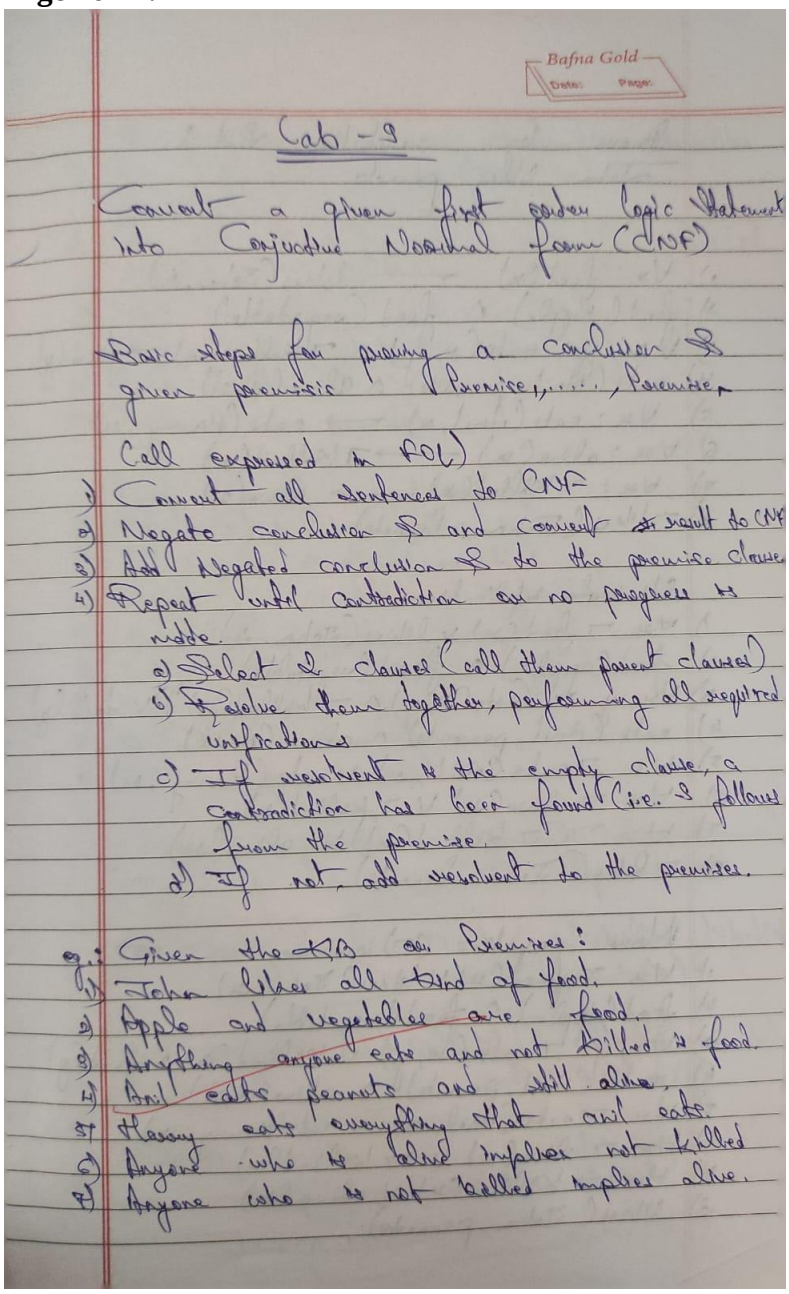
Output:

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

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

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

Program 9

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

Algorithm:



Bafna Gold
Date: Page:

Rename variables or standardize variables

1) $\forall x \neg food(x) \vee (likes(John, x)$
2) $food(apple) \wedge food(vegetable)$
3) $\forall y, z \neg eats(y, z) \vee killed(y) \vee food(z)$
4) $eats(Anil, peanuts) \wedge alive(Anil)$
5) $\forall w \neg eats(Anil, w) \vee eats(Harry, w)$
6) $\forall g \neg alive(g) \vee \neg killed(g)$
7) $\forall k killed(k) \vee alive(k)$
8) $\neg likes(John, peanuts)$

Drop universal quantifiers

1) $\neg food(x) \vee likes(John, x)$
2) $food(apple) \wedge food(vegetable)$
3) $\neg eats(y, z) \vee killed(y) \vee food(z)$
4) $eats(Anil, peanuts) \wedge alive(Anil)$
5) $\neg eats(Anil, w) \vee eats(Harry, w)$
6) $\neg alive(g) \vee \neg killed(g)$
7) $killed(k) \vee alive(k)$
8) $\neg likes(John, peanuts)$

$\neg likes(John, peanuts)$ — $\neg food(x) \vee likes(John, x)$ {peanuts/x}

~~$\neg food(peanuts)$~~ — $\neg eats(y, z) \vee killed(y) \vee food(z)$ {peanuts/z}

$\neg eats(y, peanuts) \vee killed(y)$ — $eats(Anil, peanuts)$ {Anil/y}

$killed(Anil)$ — $\neg alive(g) \vee \neg killed(g)$ {Anil/g}

$\neg alive(Anil)$ — $alive(Anil)$

{} hence proved

Prove by resolution that:
John likes peanuts.

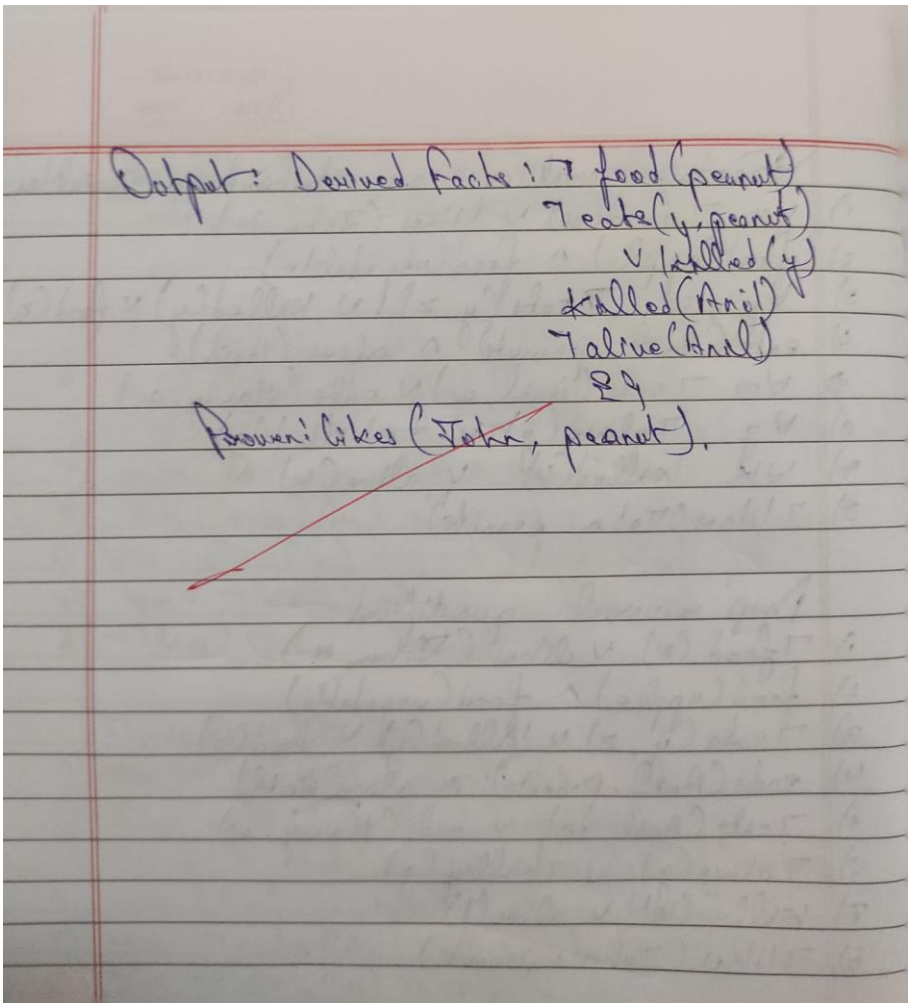
1) $\forall x: food(x) \rightarrow likes(John, x)$
2) $food(Apple) \wedge food(vegetable)$
3) $\forall x, y: eats(x, y) \wedge \neg killed(x) \rightarrow food(y)$
4) $eats(Anil, peanuts) \wedge alive(Anil)$
5) $\forall x: eats(Anil, x) \rightarrow eats(Harry, x)$
6) $\forall x: alive(x) \rightarrow \neg killed(x)$
7) $\forall x: \neg killed(x) \rightarrow alive(x)$
8) $\neg likes(John, Peanuts)$

Eliminate implications

1) $\forall x \neg food(x) \vee likes(John, x)$
2) $food(apple) \wedge food(vegetable)$
3) $\forall x, y \neg (eats(x, y) \wedge \neg killed(x)) \vee food(y)$
4) $eats(Anil, peanuts) \wedge alive(Anil)$
5) $\forall x \neg eats(Anil, x) \vee eats(Harry, x)$
6) $\forall x \neg alive(x) \vee \neg killed(x)$
7) $\forall x \neg(\neg killed(x)) \vee alive(x)$
8) $\neg likes(John, Peanuts)$

Move negation ($\neg$) inwards

1) $\forall x \neg food(x) \vee likes(John, x)$
2) $food(apple) \wedge food(vegetable)$
3) $\forall x, y \neg eats(x, y) \vee killed(x) \vee food(y)$
4) $eats(Anil, peanuts) \wedge alive(Anil)$
5) $\forall x \neg eats(Anil, x) \vee eats(Harry, x)$
6) $\forall x \neg alive(x) \vee \neg killed(x)$
7) $\forall x \neg killed(x) \vee alive(x)$
8) $likes(John, peanuts)$.



Code:

```
# Knowledge Base (KB)
facts = {
    "Eats(Anil, Peanuts)": True,
    "not Killed(Anil)": True,
    "Food(Apple)": True,
    "Food(Vegetables)": True,
}

rules = [
    # Rule: Food(X) :- Eats(Y, X) and not Killed(Y)
    {"conditions": ["Eats(Y, X)", "not Killed(Y)"], "conclusion": "Food(X)"},
    # Rule: Likes(John, X) :- Food(X)
    {"conditions": ["Food(X)"], "conclusion": "Likes(John, X)"},
]

# Query
query = "Likes(John, Peanuts)"
```

```
# Helper function to substitute variables in a rule
def substitute(rule_part, substitutions):
    for var, value in substitutions.items():
        rule_part = rule_part.replace(var, value)
    return rule_part

# Function to resolve the query
def resolve_query(facts, rules, query):
    working_facts = facts.copy()
    while True:
        new_facts_added = False
        for rule in rules:
            conditions = rule["conditions"]
            conclusion = rule["conclusion"]

            # Try all substitutions for variables (X, Y) in the rules
            for entity in ["Apple", "Vegetables", "Peanuts", "Anil", "John"]:
                substitutions = {"X": "Peanuts", "Y": "Anil"}  # Fixed for this problem
                resolved_conditions = [substitute(cond, substitutions) for cond in conditions]
                resolved_conclusion = substitute(conclusion, substitutions)

                # Check if all conditions are true
                if all(working_facts.get(cond, False) for cond in resolved_conditions):
                    if resolved_conclusion not in working_facts:
                        working_facts[resolved_conclusion] = True
                        new_facts_added = True
                        print(f"Derived Fact: {resolved_conclusion}")

        if not new_facts_added:
            break

    # Check if the query is resolved
    return working_facts.get(query, False)

# Run the resolution process
if resolve_query(facts, rules, query):
    print(f"Proven: {query}")
else:
    print(f"Not Proven: {query}")
```

Output:

Derived Fact: Food(Peanuts)

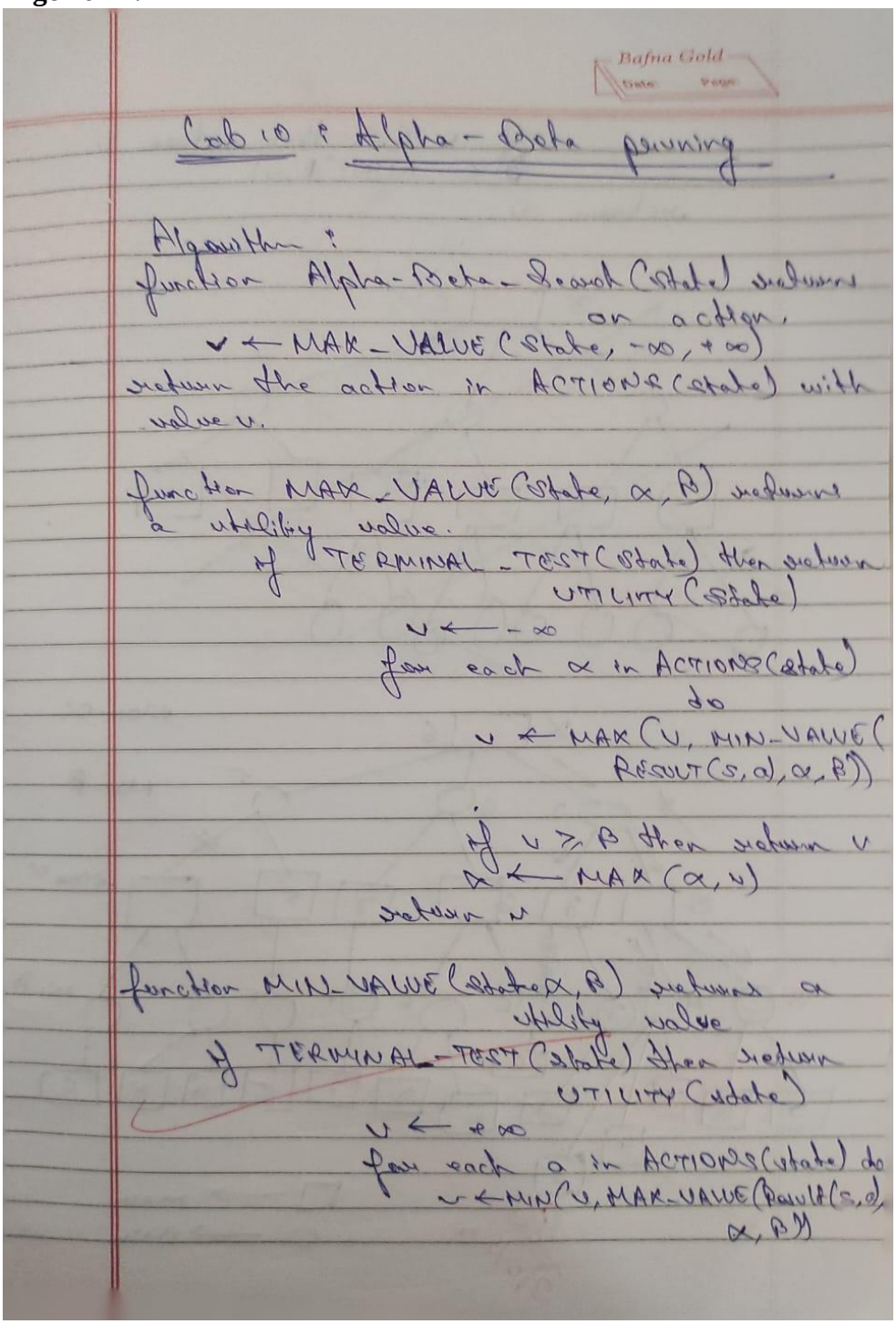
Derived Fact: Likes(John, Peanuts)

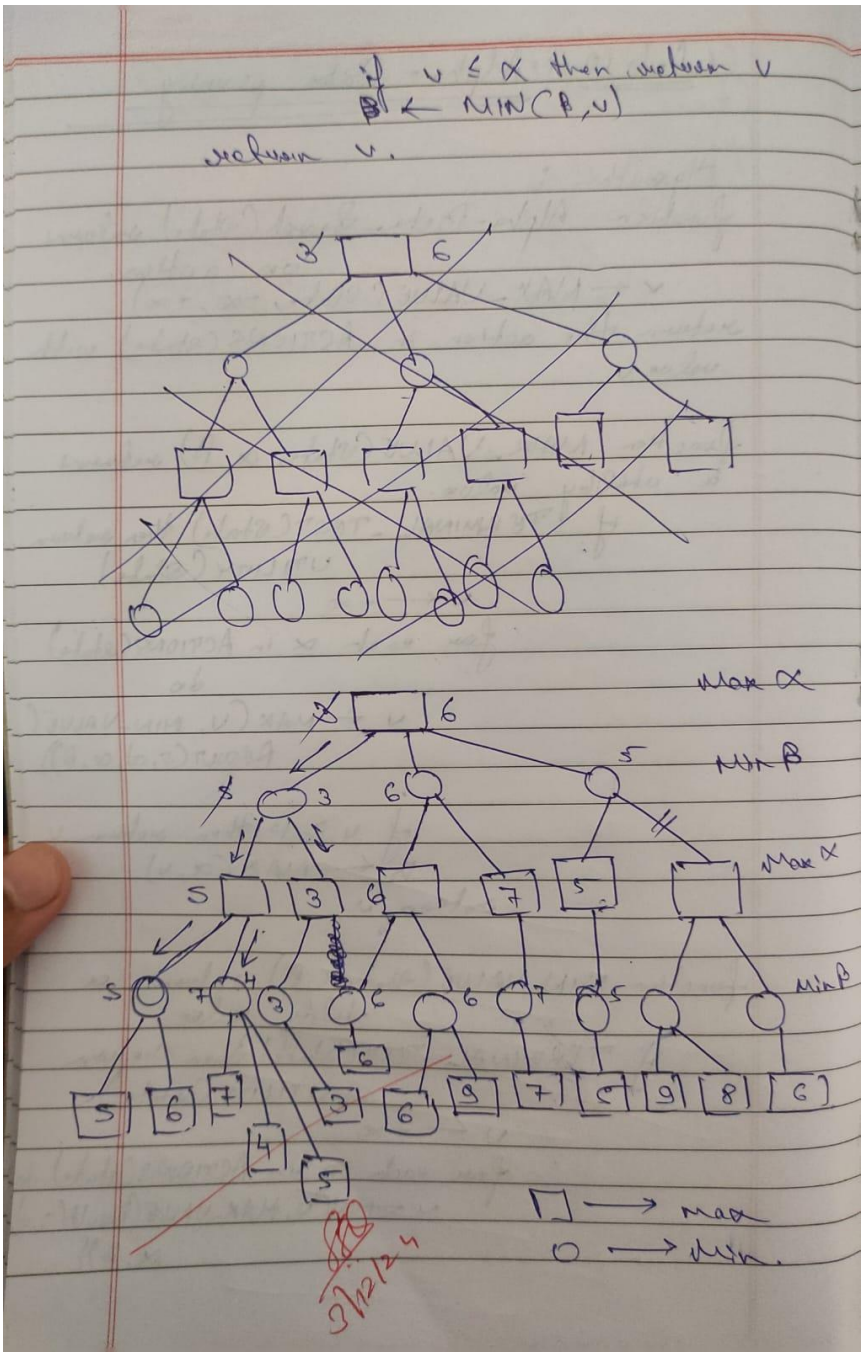
Proven: Likes(John, Peanuts)

Program 10

Implement Alpha-Beta Pruning.

Algorithm:





Code:

```
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