VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB REPORT
on

Artificial Intelligence (23CS5PCAIN)

Submitted by

Abhay N Y (1BM24CS400)

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 **Abhay N Y (1BM24CS400),** who is bonafide student of **B.M.S. College of Engineering.** It is in partial fulfillment for the award of **Bachelor of Engineering in Computer Science and Engineering** of the Visvesvaraya Technological University, Belgaum. The Lab report has been approved as it satisfies the academic requirements in respect of an Artificial Intelligence (23CS5PCAIN) work prescribed for the said degree.

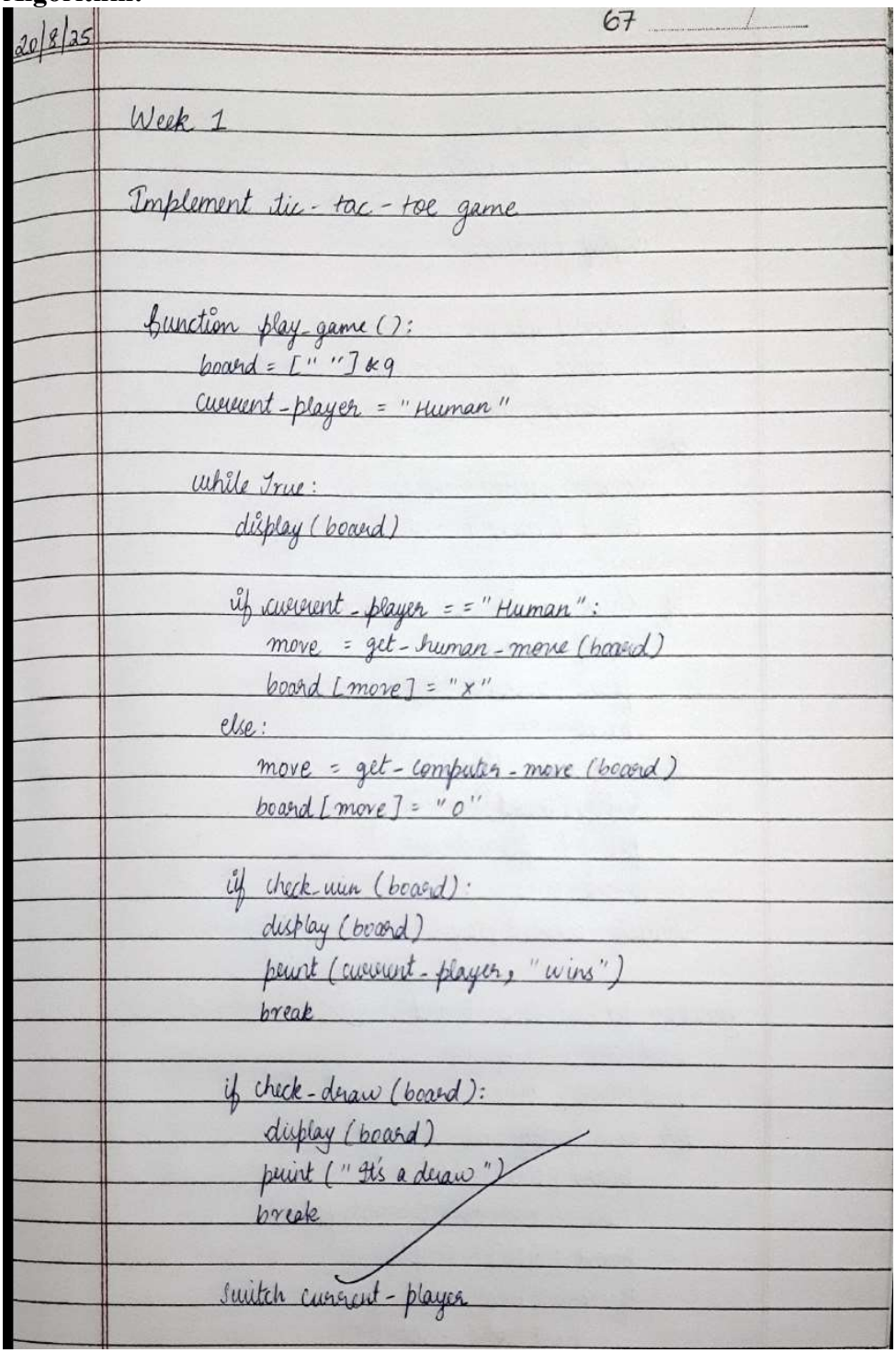
Sandhya A Kulkarni Assistant Professor Department of CSE, BMSCE	Dr. Kavitha Sooda Professor & HOD Department of CSE, BMSCE

Index

Github Link:
https://github.com/abhay-ny/AI_LAB

Program 1
Implement Tic –Tac –Toe Game
Implement vacuum cleaner agent

Algorithm:



20/8/25 67

Week 1

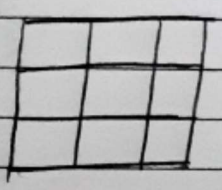
Implement tic - tac - toe game

```
function play_game():
    board = [" "]*9
    current_player = "Human"

    while True:
        display(board)

        if current_player == "Human":
            move = get_human_move(board)
            board[move] = "X"
        else:
            move = get_computer_move(board)
            board[move] = "O"

        if check_win(board):
            display(board)
            print(current_player, "wins")
            break

        if check_draw(board):
            display(board)
            print("It's a draw")
            break

        Switch current_player
```

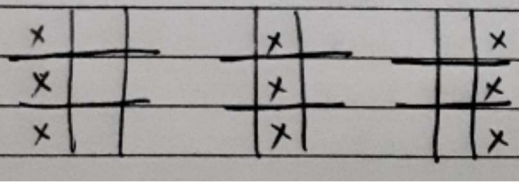
```
function play game():
    board = 3x3 empty grid
    while True:
        display(board)

        If current player is Human:
            move = get Human Move(board)
            board[move] = 'x'
        else:
            move = get Best Move(board)
            board[move] = 'o'

        If check win(board, current player):
            display(board)
            print(current player, "wins")
            break
        If check draw(board):
            display(board)
            print("Its a draw")
            break
        switch current player

function get Best Move(board):
    best score = -infinity
    best move = none
    for each empty cell in board:
        board[cell] = 'o'
        score = minimax(board, false)
        board[cell] = empty
        If score > best score:
            best score = score
            best move = cell
    return best move
```

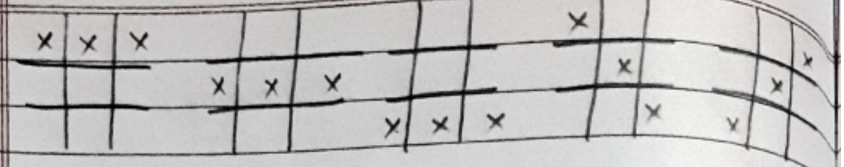
```
function minmax (board, isMaximizing):
    if checkWin (board, 'O'): return +1
    if checkWin (board, 'X'): return -1
    if checkDraw (board): return 0

    if ismaxing:
        bestScore = -infinity
        for each empty cell in board:
            board[cell] = 'O'
            score = minimax (board, false)
            board [cell] = empty
            bestScore = max (bestScore, Score)
        return bestScore
    else:
        bestScore = + infinity
        for each empty cell in board:
            board[cell] = 'X'
            Score = minmax (board, true)
            board [cell] = empty
            bestscore = min (bestScore, score)
        return bestScore.
```

① Initialization

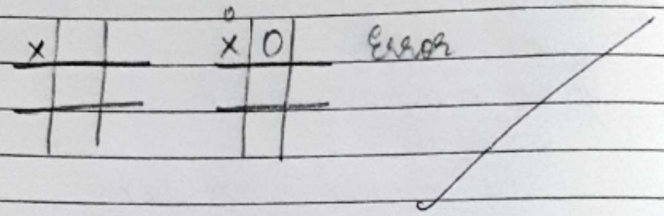


② Winning Conditions

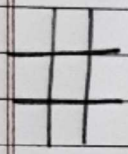




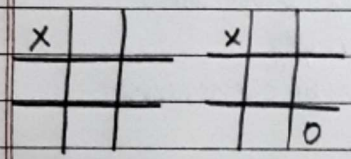
3. If the player tries to put it in the same place it returns error



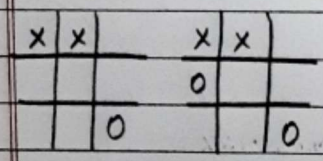
Output:



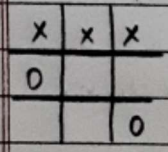
enter position (1-9): 1



enter position (1-9): 2



enter position (1-9): 3



Human wins!

Code:

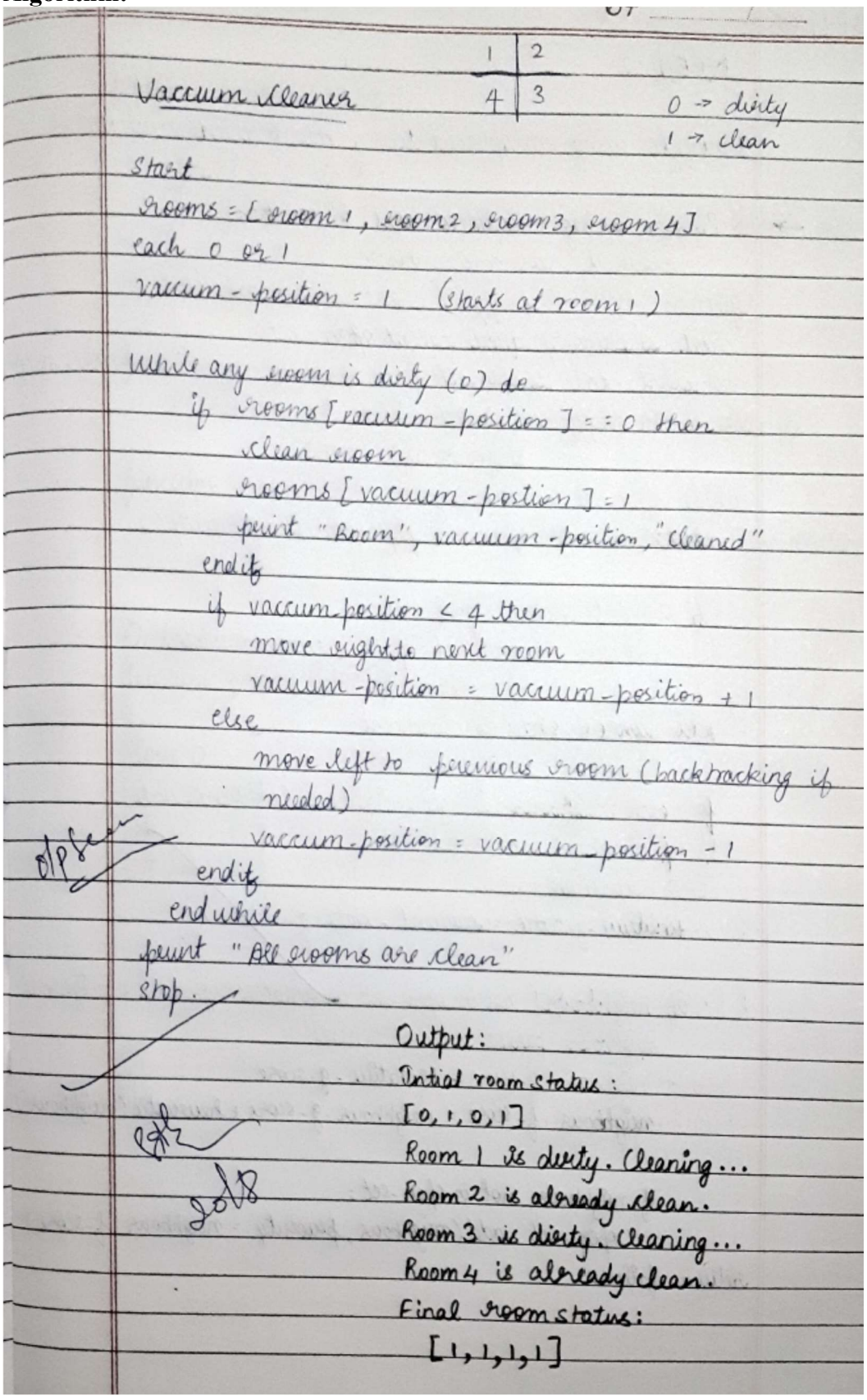
```
# tic tac toe game by Abhay N Y
import random
# Display board
def display(board):
    print("\n")
    for i in range(3):
        print(board[3*i] + " | " + board[3*i+1] + " | " + board[3*i+2])
        if i < 2:
            print("--+---+--")
    print("\n")
# Check for winner
def check_win(board, player):
    win_combos = [(0,1,2), (3,4,5), (6,7,8),
                  (0,3,6), (1,4,7), (2,5,8),
                  (0,4,8), (2,4,6)]
    for a,b,c in win_combos:
        if board[a] == board[b] == board[c] == player:
            return True
    return False
# Check draw
def check_draw(board):
    return " " not in board
# Human move
def human_move(board):
    while True:
        try:
            move = int(input("Enter position (1-9): ")) - 1
            if board[move] == " ":
                return move
            else:
                print("Spot taken, try again.")
```

```
        except (ValueError, IndexError):
            print("Invalid input, choose 1-9.")
# Computer move (random)
def computer_move(board):
    empty = [i for i in range(9) if board[i] == " "]
    return random.choice(empty)
# Main game
def play():
    board = [" "] * 9
    current_player = "Human"
    while True:
        display(board)
        if current_player == "Human":
            move = human_move(board)
            board[move] = "X"
        else:
            move = computer_move(board)
            board[move] = "O"
        if check_win(board, "X"):
            display(board)
            print("Abhay N Y wins!")
            break
        elif check_win(board, "O"):
            display(board)
            print("Computer wins!")
            break
        elif check_draw(board):
            display(board)
            print("It's a draw!")
            break
        current_player = "Computer" if current_player == "Human" else "Human"
play()
```

Output:

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

Enter position (1-9):  1

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



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

Enter position (1-9):  3

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



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

Enter position (1-9):  2

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

Abhay N Y wins!
```

Algorithm:



	1	2
Vaccum cleaner	4	3

0 → dirty
1 → clean

Start
rooms = [room 1, room 2, room 3, room 4]
each 0 or 1
vaccum - position = 1 (starts at room 1)

While any room is dirty (0) do
if rooms [vacuum - position] = = 0 then
clean room
rooms [vacuum - postion] = 1
print "Room", vacuum - position, "cleaned"
endif
if vaccum_position < 4 then
move right to next room
vacuum - position = vacuum - position + 1
else
move left to previous room (backtracking if needed)
vaccum_position = vacuum_position - 1
endif
endwhile
print "All rooms are clean"
stop.

Output:
Intial room status:
[0, 1, 0, 1]
Room 1 is dirty. Cleaning...
Room 2 is already clean.
Room 3 is dirty. Cleaning...
Room 4 is already clean.
Final room status:
[1, 1, 1, 1]

Code:

```
# Vacuum Cleaner for 4 Rooms by Abhay N Y
def vacuum_cleaner(rooms):
    print("Initial Room Status:")
    print(rooms)
    # Traverse through each room
    for i in range(len(rooms)):
        if rooms[i] == 0:
            print(f"Room {i+1} is Dirty. Cleaning...")
            rooms[i] = 1
        else:
            print(f"Room {i+1} is already Clean.")
    print("\nFinal Room Status of Abhay's Room:")
    print(rooms)
# 0 = Dirty, 1 = Clean
# Example: [0, 1, 0, 1] means Room1=Dirty, Room2=Clean, Room3=Dirty, Room4=Clean
rooms = [0, 1, 0, 1]
vacuum_cleaner(rooms)
```

Output:

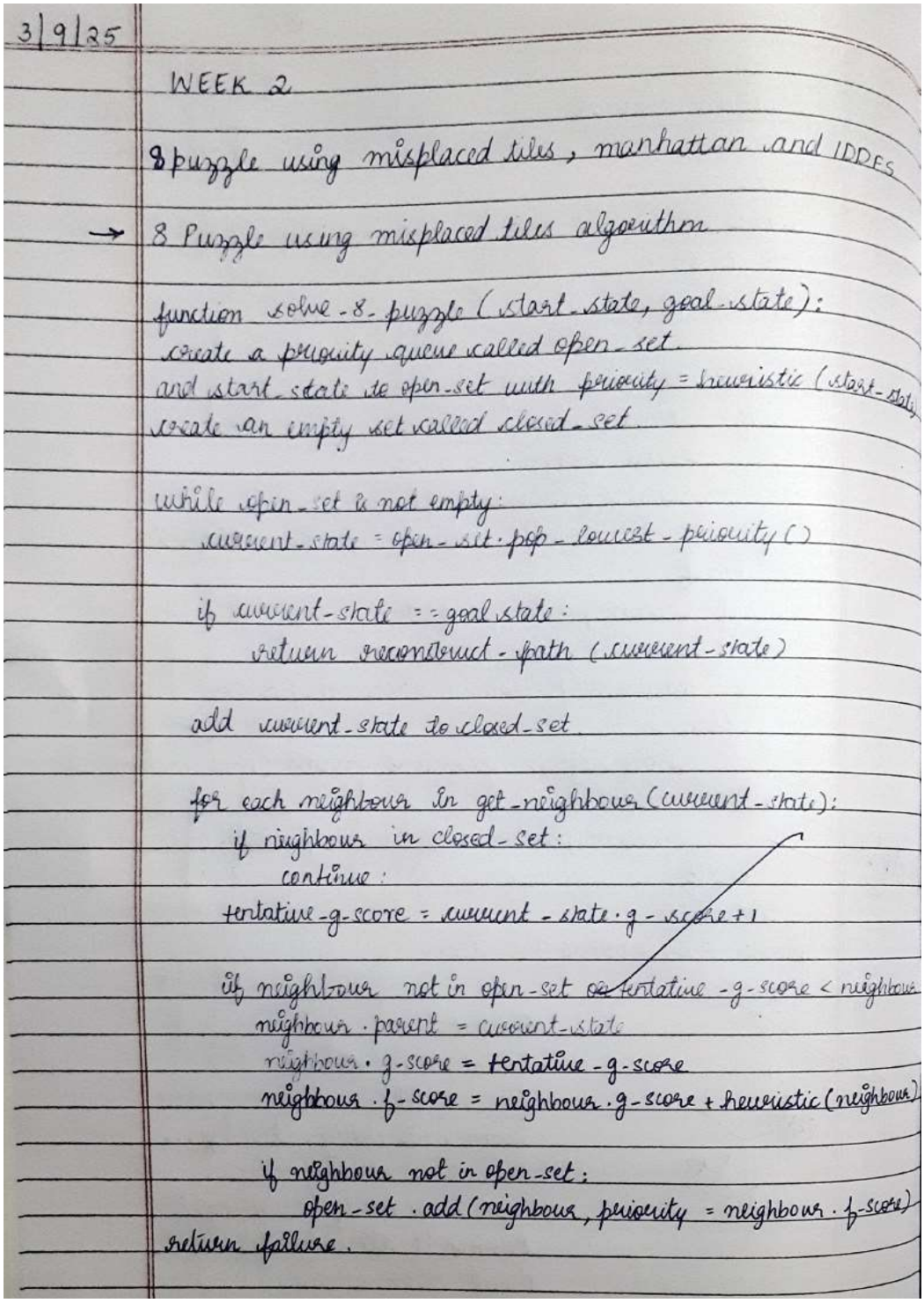
```
Initial Room Status:
[0, 1, 0, 1]
Room 1 is Dirty. Cleaning...
Room 2 is already Clean.
Room 3 is Dirty. Cleaning...
Room 4 is already Clean.

Final Room Status of Abhay's Room:
[1, 1, 1, 1]
```

Program 2

Implement 8 puzzle problems using Depth First Search (DFS).

Implement Iterative deepening search algorithm.

Algorithm:



3/9/25

WEEK 2

8puzzle using misplaced tiles, manhattan and IDDFS

→ 8 Puzzle using misplaced tiles algorithm

```
function solve-8-puzzle (start_state, goal_state):
create a priority queue called open_set.
and start_state to open_set with priority = heuristic (start_state
create an empty set called closed_set.

while open_set is not empty:
    current_state = open_set.pop_lowest_priority()

    if current_state == goal state:
        return reconstruct_path (current_state)

    add current_state to closed_set.

    for each neighbour in get_neighbour (current_state):
        if neighbour in closed_set:
            continue:
        tentative_g_score = current_state.g_score+1

        if neighbour not in open_set or tentative_g_score < neighbour
            neighbour.parent = current_state
            neighbour.g_score = tentative_g_score
            neighbour.f_score = neighbour.g_score + heuristic(neighbour)

            if neighbour not in open_set:
                open_set.add(neighbour, priority = neighbour.f_score)
return failure.
```

Ex:

Calculates how far each tile is from its goal position in terms of moves & sums these distances (manhattan)

counts the number of tiles that are not in their correct position store 67

```
function heuristic (state):
    count = 0
    for i in 0 to 8:
        if state[i] != goal_state[i] and state[i] != blank_tile:
            count += 1
    return count

function get_neighbour (state):
    // return all valid states by moving the blanks title up,
                down, left or right
function reconstruct_path (state):
    // backtrack through parent pointers to return the solution
        path.
```

Output:

Solution found! Path of states:

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

Code:

#8 puzzle game using misplaced algorithm by Abhay NY

```
import heapq
# Define the goal state
GOAL_STATE = (1, 2, 3, 4, 5, 6, 7, 8, 0)
N = 3  # The size of the grid (3x3)
# Helper function to convert a 2D grid to a 1D tuple
def grid_to_tuple(grid):
    return tuple(tile for row in grid for tile in row)
# Helper function to convert a 1D tuple to a 2D grid
def tuple_to_grid(tup):
    return [list(tup[i:i + N]) for i in range(0, N * N, N)]
# A node in the search tree
class PuzzleNode:
    def __init__(self, state, parent=None, move=None, cost=0):
        self.state = state
        self.parent = parent
        self.move = move
        self.cost = cost
        self.misplaced_tiles = self.calculate_misplaced_tiles()
    # The __lt__ method is used for comparison in the heapq (priority queue)
    def __lt__(self, other):
        return (self.cost + self.misplaced_tiles) < (other.cost + other.misplaced_tiles)
    # Heuristic function: count misplaced tiles
    def calculate_misplaced_tiles(self):
        count = 0
        for i in range(N * N):
            if self.state[i] != 0 and self.state[i] != GOAL_STATE[i]:
                count += 1
        return count
    # Check if the current state is the goal state
    def is_goal_state(self):
```

```
        return self.state == GOAL_STATE
    # Generate all possible next states (neighbors)
    def get_neighbors(self):
        neighbors = []
        state_grid = tuple_to_grid(self.state)
        blank_row, blank_col = self.find_blank(state_grid)
        # Possible moves: Up, Down, Left, Right
        moves = [(-1, 0), (1, 0), (0, -1), (0, 1)]
        for dr, dc in moves:
            new_row, new_col = blank_row + dr, blank_col + dc
            # Check if the move is within the grid boundaries
            if 0 <= new_row < N and 0 <= new_col < N:
                new_grid = [row[:] for row in state_grid]
                # Swap the blank tile with the adjacent tile
                new_grid[blank_row][blank_col], new_grid[new_row][new_col] = \
                    new_grid[new_row][new_col], new_grid[blank_row][blank_col]
                new_state = grid_to_tuple(new_grid)
                neighbors.append(PuzzleNode(new_state, self, (dr, dc), self.cost + 1))
        return neighbors
    # Find the position of the blank tile (0)
    def find_blank(self, grid):
        for r in range(N):
            for c in range(N):
                if grid[r][c] == 0:
                    return r, c
        return None, None
def solve_8_puzzle(initial_state):
    """
    Solves the 8-puzzle using A* search with the misplaced tiles heuristic.
    Args:
        initial_state (tuple): The starting configuration of the puzzle.
```

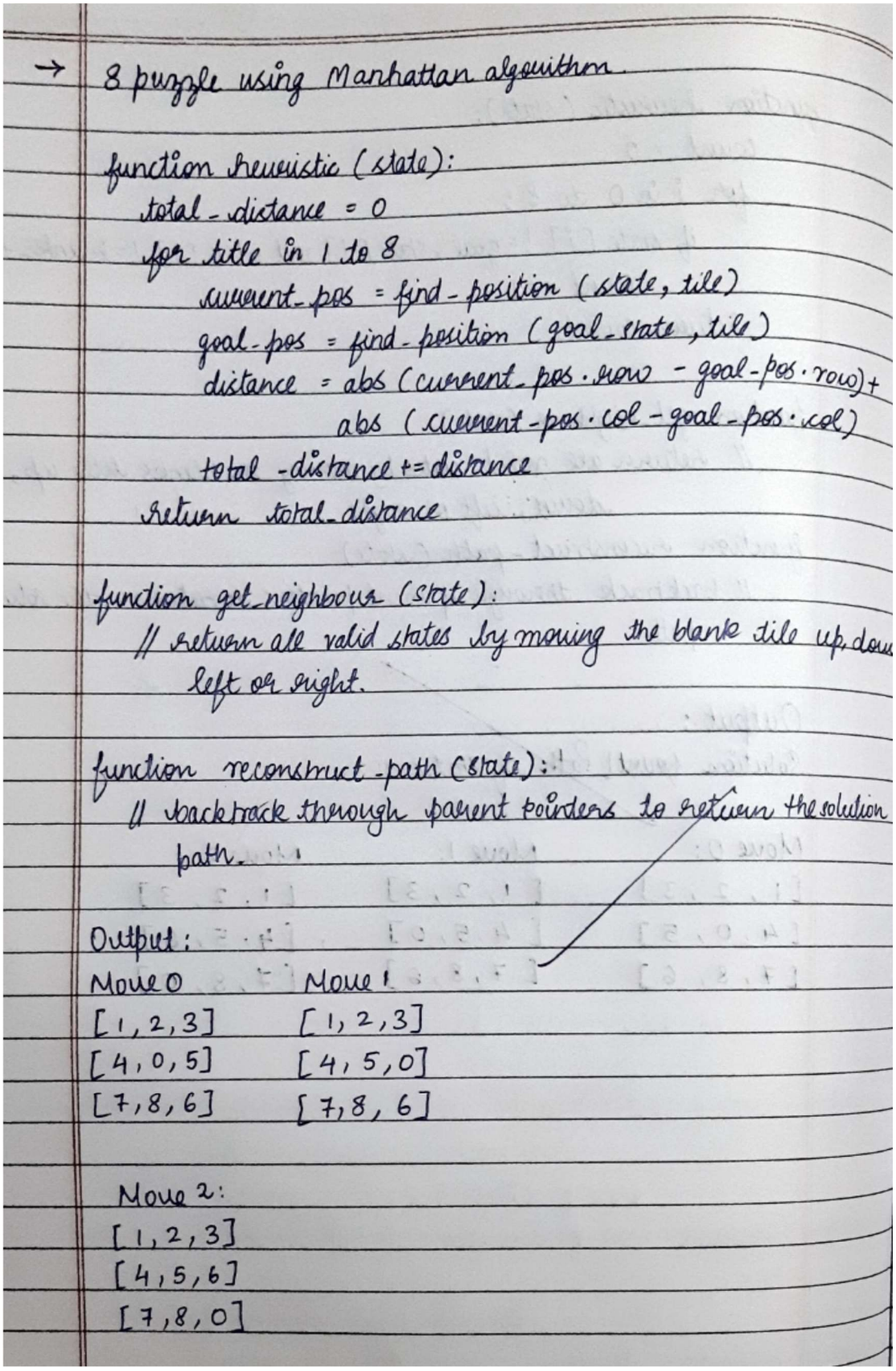
```
    Returns:
        A list of states representing the solution path, or None if no solution exists.
    """
    open_list = []
    root_node = PuzzleNode(initial_state)
    heapq.heappush(open_list, (root_node.cost + root_node.misplaced_tiles, root_node))
    # Keep track of visited states to avoid cycles
    closed_list = set()
    closed_list.add(initial_state)
    while open_list:
        # Get the node with the lowest f_score
        f_score, current_node = heapq.heappop(open_list)
        # Check if the goal is reached
        if current_node.is_goal_state():
            path = []
            while current_node:
                path.append(current_node.state)
                current_node = current_node.parent
            return path[::-1] # Return the path in correct order
        # Expand the current node
        for neighbor in current_node.get_neighbors():
            if neighbor.state not in closed_list:
                closed_list.add(neighbor.state)
                heapq.heappush(open_list, (neighbor.cost + neighbor.misplaced_tiles, neighbor))
    return None  # No solution found
# Example Usage:
if __name__ == "__main__":
    # A solvable initial state.
    initial_puzzle = (1, 2, 3, 4, 0, 5, 7, 8, 6)
    solution_path = solve_8_puzzle(initial_puzzle)
    if solution_path:
        print("Solution found! Path of states:")
```

```
    for i, state in enumerate(solution_path):
        print(f"Move {i}:")
        grid = tuple_to_grid(state)
        for row in grid:
            print(row)
        print()
else:
    print("No solution exists for this puzzle.")
```

Output:

```
Solution found! Path of states:
Move 0:
[1, 2, 3]
[4, 0, 5]
[7, 8, 6]

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

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

Algorithm:



→ 8 puzzle using Manhattan algorithm

```
function heuristic (state):
    total_distance = 0
    for title in 1 to 8
        current_pos = find_position (state, tile)
        goal_pos = find_position (goal_state, tile)
        distance = abs (current_pos.row - goal_pos.row) +
                   abs (current_pos.col - goal_pos.col)
        total_distance += distance
    return total_distance

function get_neighbour (state):
    // return all valid states by moving the blank tile up, down
       left or right.

function reconstruct_path (state):
    // backtrack through parent pointers to return the solution
       path.
```

Output:

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

Move 2:

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

Code:

#8 puzzle game using manhatten algorithm by Abhay NY

```
import heapq
# Define the goal state
GOAL_STATE = (1, 2, 3, 4, 5, 6, 7, 8, 0)
N = 3  # The size of the grid (3x3)
# Helper function to convert a 2D grid to a 1D tuple
def grid_to_tuple(grid):
    return tuple(tile for row in grid for tile in row)
# Helper function to convert a 1D tuple to a 2D grid
def tuple_to_grid(tup):
    return [list(tup[i:i + N]) for i in range(0, N * N, N)]
# Pre-calculate goal positions for quick lookup
GOAL_POSITIONS = {
    tile: (i // N, i % N) for i, tile in enumerate(GOAL_STATE)
}
# A node in the search tree
class PuzzleNode:
    def __init__(self, state, parent=None, cost=0):
        self.state = state
        self.parent = parent
        self.cost = cost
        self.manhattan_distance = self.calculate_manhattan_distance()
    # The __lt__ method is used for comparison in the heapq (priority queue)
    def __lt__(self, other):
        return (self.cost + self.manhattan_distance) < (other.cost + other.manhattan_distance)
    # Heuristic function: calculate Manhattan distance
    def calculate_manhattan_distance(self):
        distance = 0
        for i in range(N * N):
            tile_value = self.state[i]
```

```
            if tile_value != 0:
                current_row, current_col = i // N, i % N
                goal_row, goal_col = GOAL_POSITIONS[tile_value]
                distance += abs(current_row - goal_row) + abs(current_col - goal_col)
        return distance
    # Check if the current state is the goal state
    def is_goal_state(self):
        return self.state == GOAL_STATE
    # Generate all possible next states (neighbors)
    def get_neighbors(self):
        neighbors = []
        state_grid = tuple_to_grid(self.state)
        blank_row, blank_col = self.find_blank(state_grid)
        moves = [(-1, 0), (1, 0), (0, -1), (0, 1)] # Up, Down, Left, Right
        for dr, dc in moves:
            new_row, new_col = blank_row + dr, blank_col + dc
            if 0 <= new_row < N and 0 <= new_col < N:
                new_grid = [row[:] for row in state_grid]
                # Swap the blank tile with the adjacent tile
                new_grid[blank_row][blank_col], new_grid[new_row][new_col] = \
                    new_grid[new_row][new_col], new_grid[blank_row][blank_col]
                new_state = grid_to_tuple(new_grid)
                neighbors.append(PuzzleNode(new_state, self, self.cost + 1))
        return neighbors
    # Find the position of the blank tile (0)
    def find_blank(self, grid):
        for r in range(N):
            for c in range(N):
                if grid[r][c] == 0:
                    return r, c
        return None, None
def solve_8_puzzle(initial_state):
```

```
    """
    Solves the 8-puzzle using A* search with the Manhattan distance heuristic.
    Args:
        initial_state (tuple): The starting configuration of the puzzle.
    Returns:
        A list of states representing the solution path, or None if no solution exists.
    """
    open_list = []
    root_node = PuzzleNode(initial_state)
    heapq.heappush(open_list, (root_node.cost + root_node.manhattan_distance, root_node))
    closed_list = set()
    closed_list.add(initial_state)
    while open_list:
        f_score, current_node = heapq.heappop(open_list)
        if current_node.is_goal_state():
            path = []
            while current_node:
                path.append(current_node.state)
                current_node = current_node.parent
            return path[::-1]
        for neighbor in current_node.get_neighbors():
            if neighbor.state not in closed_list:
                closed_list.add(neighbor.state)
                heapq.heappush(open_list, (neighbor.cost + neighbor.manhattan_distance, neighbor))
    return None
# Example Usage:
if __name__ == "__main__":
    # A solvable initial state.
    initial_puzzle = (1, 2, 3, 4, 0, 5, 7, 8, 6)
    solution_path = solve_8_puzzle(initial_puzzle)
    if solution_path:
        print("Solution found! Path of states:")
```

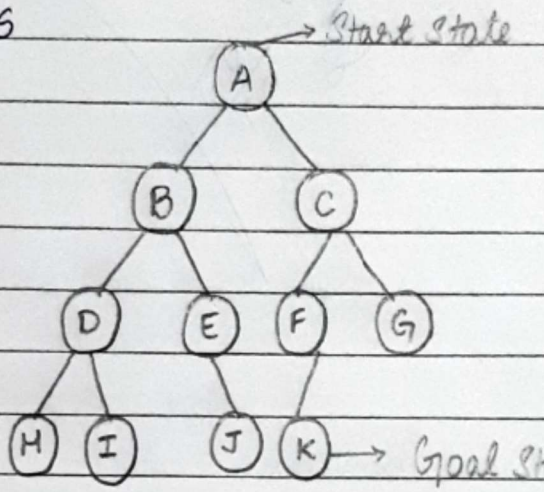
```
    for i, state in enumerate(solution_path):
        print(f"Move {i}:")
        grid = tuple_to_grid(state)
        for row in grid:
            print(row)
        print()
else:
    print("No solution exists for this puzzle.")
```

Output:

```
Solution found! Path of states:
Move 0:
[1, 2, 3]
[4, 0, 5]
[7, 8, 6]

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

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

Algorithm:

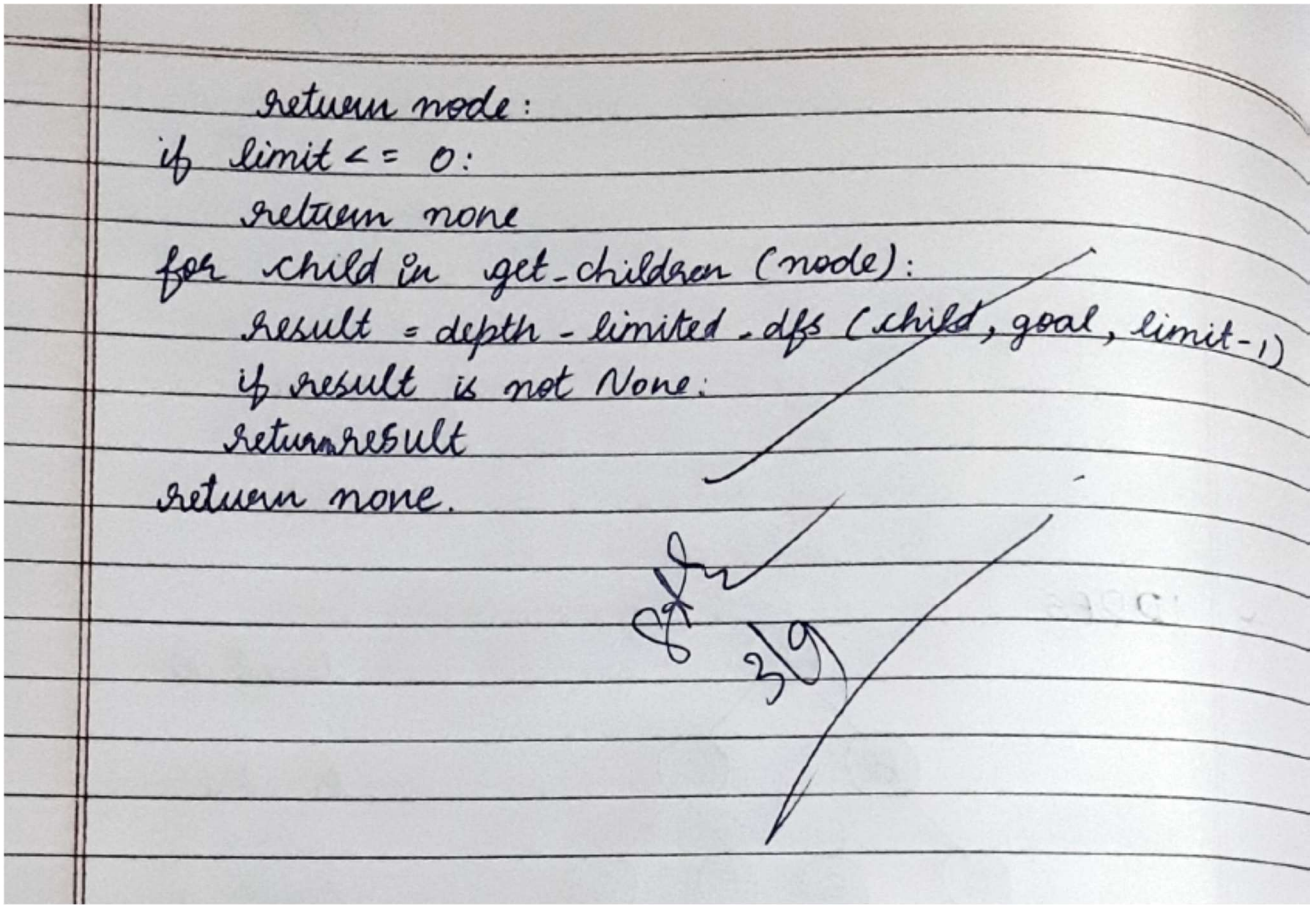
→ IDDFS



level 0 → A
level 1 → A B C
level 2 → A B D E C F G
level 3 → A B D H I E J C F K

Pseudocode:

```
def iterative_deepening_dfs(start, goal, depth)
    for depth in range(max_depth+1):
        found = depth_limited_dfs(start, goal, depth)
        if found:
            return found
    return None

def depth_limited_dfs(node, goal, limit):
    if node == goal:
```



Code:

#IDDFS algorithm by Abhay NY

```
def dfs_limited(graph, start, goal, limit):
    """
    Performs a Depth-First Search with a depth limit.
    Args:
        graph (dict): The graph represented as an adjacency list.
        start: The starting node.
        goal: The goal node to find.
        limit (int): The maximum depth to search.
    Returns:
        list: The path from start to goal if found, otherwise None.
    """
    stack = [(start, [start])]
```

```
    while stack:
        current_node, path = stack.pop()
        # Check if the goal is found
        if current_node == goal:
            return path
        # Check if the depth limit has been reached
        if len(path) > limit:
            continue
        # Explore neighbors
        for neighbor in reversed(graph.get(current_node, [])):
            if neighbor not in path:
                new_path = list(path)
                new_path.append(neighbor)
                stack.append((neighbor, new_path))
    return None
def iddfs(graph, start, goal):
    """
    Performs an Iterative Deepening Depth-First Search.
    Args:
        graph (dict): The graph represented as an adjacency list.
        start: The starting node.
        goal: The goal node to find.
    Returns:
        list: The path from start to goal, or None if the goal is unreachable.
    """
    depth_limit = 0
    while True:
        print(f"Searching with depth limit: {depth_limit}")
        result = dfs_limited(graph, start, goal, depth_limit)

        if result is not None:
```

```
            return result
        depth_limit += 1
# Example Usage:
if __name__ == "__main__":
    # Define a simple graph
    graph = {
        'A': ['B', 'C'],
        'B': ['D', 'E'],
        'C': ['F','G'],
        'D': ['H', 'I'],
        'E': ['J'],
        'F': ['K'],
        'G': [],
    }
    start_node = 'A'
    goal_node = 'K'
    path = iddfs(graph, start_node, goal_node)
    if path:
        print("\nGoal found!")
        print(f"Path: {' -> '.join(path)}")
    else:
        print("\nGoal not reachable from the start node.")
```

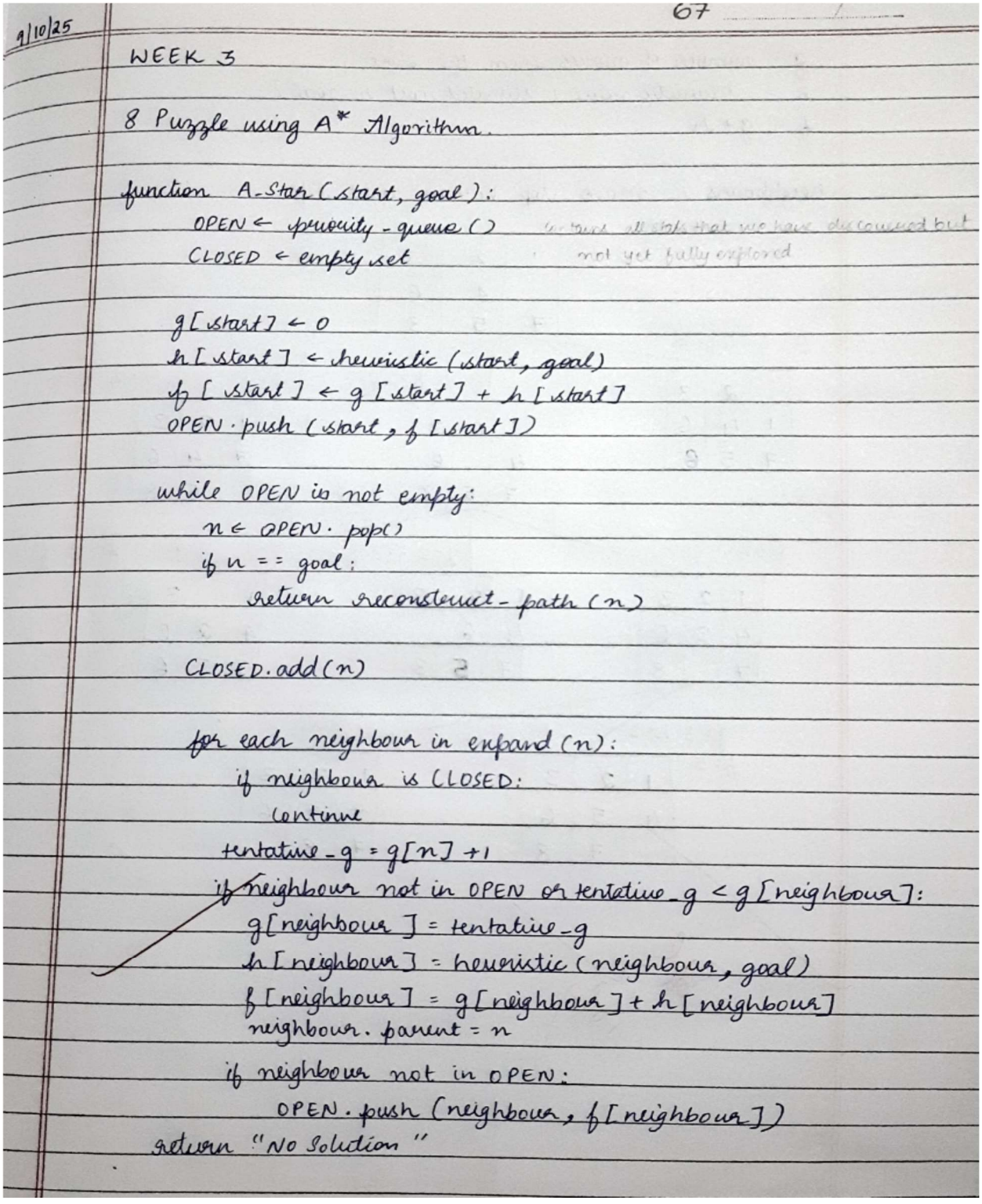
Output:

```
Searching with depth limit: 0
Searching with depth limit: 1
Searching with depth limit: 2
Searching with depth limit: 3

Goal found!
Path: A -> C -> F -> K
```

Program 3

Implement A* search algorithm.

Algorithm:



9/10/25

67

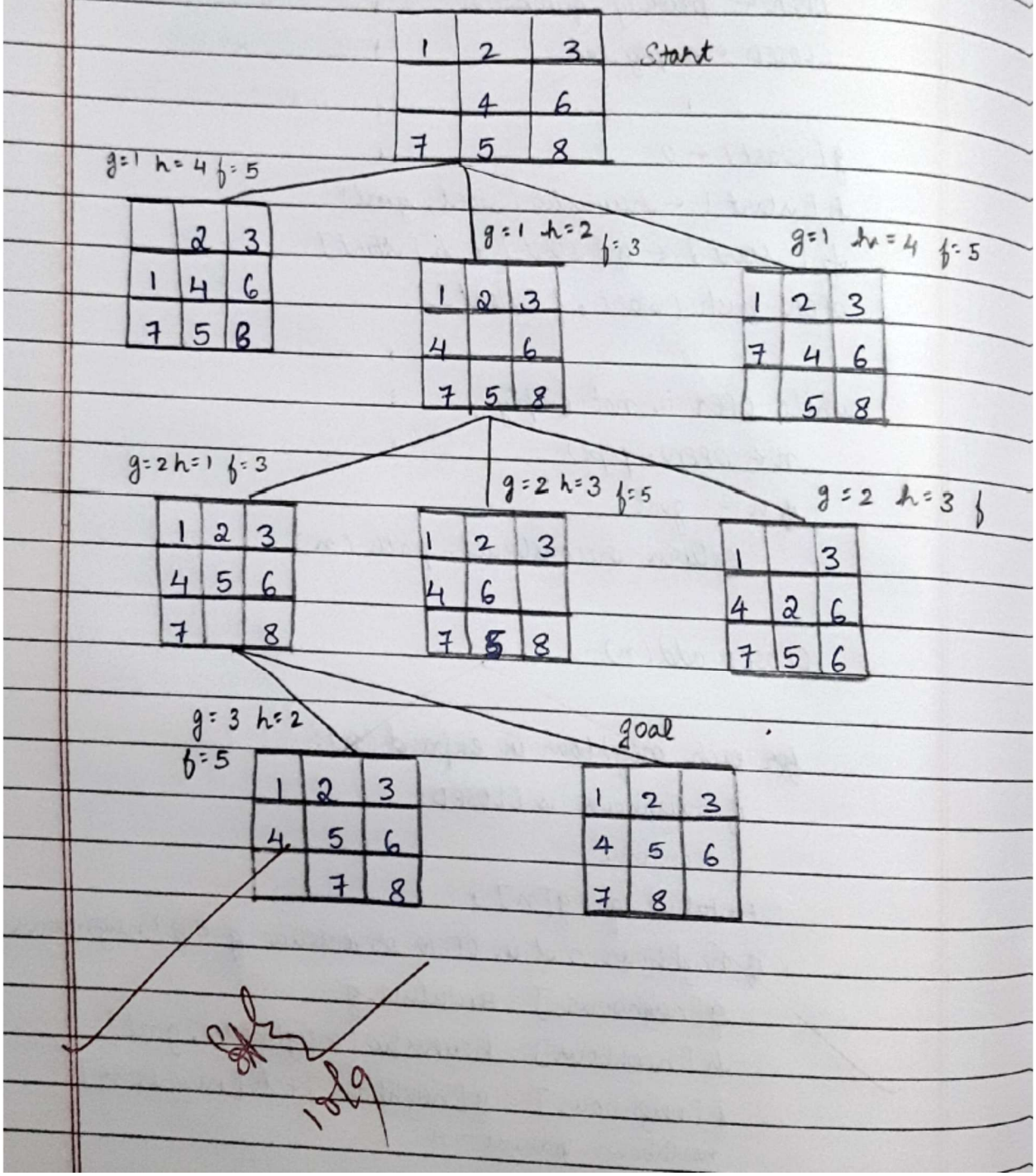
WEEK 3

8 Puzzle using A* Algorithm.

```
function A_Star(start, goal):
    OPEN ← priority_queue()        # nodes that we have discovered but not yet fully explored
    CLOSED ← empty set

    g[start] ← 0
    h[start] ← heuristic(start, goal)
    f[start] ← g[start] + h[start]
    OPEN.push(start, f[start])

    while OPEN is not empty:
        n ← OPEN.pop()
        if n == goal:
            return reconstruct_path(n)

        CLOSED.add(n)

        for each neighbour in expand(n):
            if neighbour is CLOSED:
                continue
            tentative_g = g[n] + 1
            if neighbour not in OPEN or tentative_g < g[neighbour]:
                g[neighbour] = tentative_g
                h[neighbour] = heuristic(neighbour, goal)
                f[neighbour] = g[neighbour] + h[neighbour]
                neighbour.parent = n
                if neighbour not in OPEN:
                    OPEN.push(neighbour, f[neighbour])
    return "No Solution"
```

g = number of moves from the start
h = heuristic value (estimated cost to goal)
f = g + h

neighbours : move top bottom right left.



Summary table

Method	Time Complexity	Notes
Misplaced tiles	$O(b^d)$	Expands many nodes, weak heuristic
Manhattan distance	$O(b^{(d-\epsilon)})$	Strong heuristic, fewer nodes expanded
IDDFS	$O(b^d)$	Low memory but re Expands nodes.
A^*	$O(b^d)$ (general)	Practical efficiency depends on heuristic.

Output:

Enter the initial state of the puzzel (use 0 for blank):
Row1 (Space seperated): 1 2 3
Row2 (Space seperated): 0 4 3
Row3 (Space seperated): 7 5 8

Solution found in 3 moves:

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

Code :

#8 puzzle game using A* algorithm by Abhay NY

```
import heapq
# Directions for movement: up, down, left, right
moves = [(-1, 0), (1, 0), (0, -1), (0, 1)]
# Goal state of the 8 puzzle
goal_state = [[1, 2, 3],
              [4, 5, 6],
              [7, 8, 0]]   # 0 represents the blank space
# Manhattan distance heuristic
def manhattan_distance(state):
    distance = 0
    for i in range(3):
        for j in range(3):
            value = state[i][j]
            if value != 0:  # Skip blank tile
                target_x = (value - 1) // 3
                target_y = (value - 1) % 3
                distance += abs(i - target_x) + abs(j - target_y)
    return distance
# Function to check if the state is goal
def is_goal(state):
    return state == goal_state
# Function to get neighbors (possible moves)
def get_neighbors(state):
    neighbors = []
    # Find blank (0)
    x, y = [(ix, iy) for ix, row in enumerate(state) for iy, i in enumerate(row) if i == 0][0]
    for dx, dy in moves:
        new_x, new_y = x + dx, y + dy
        if 0 <= new_x < 3 and 0 <= new_y < 3:
```

```
            # Swap blank with new position
            new_state = [row[:] for row in state]
            new_state[x][y], new_state[new_x][new_y] = new_state[new_x][new_y], new_state[x][y]
            neighbors.append(new_state)
    return neighbors
# Convert state to tuple for hashing
def state_to_tuple(state):
    return tuple(tuple(row) for row in state)
# A* algorithm
def a_star(start_state):
    open_set = []
    heapq.heappush(open_set, (manhattan_distance(start_state), 0, start_state, []))
    visited = set()
    while open_set:
        f, g, current, path = heapq.heappop(open_set)
        if is_goal(current):
            return path + [current]
        visited.add(state_to_tuple(current))
        for neighbor in get_neighbors(current):
            if state_to_tuple(neighbor) not in visited:
                heapq.heappush(open_set, (g + 1 + manhattan_distance(neighbor), g + 1, neighbor, path +
[current]))
    return None  # No solution
# Print state in nice format
def print_state(state):
    for row in state:
        print(row)
    print()
# Main program
if __name__ == "__main__":
    print("Enter the initial state of the puzzle (use 0 for blank):")
    start_state = []
```

```
    for i in range(3):
        row = list(map(int, input(f"Row {i+1} (space separated): ").split()))
        start_state.append(row)
    print("\nInitial State:")
    print_state(start_state)
    solution = a_star(start_state)
    if solution:
        print("Solution found in", len(solution) - 1, "moves:")
        for step in solution:
            print_state(step)
    else:
        print(" No solution exists.")
```

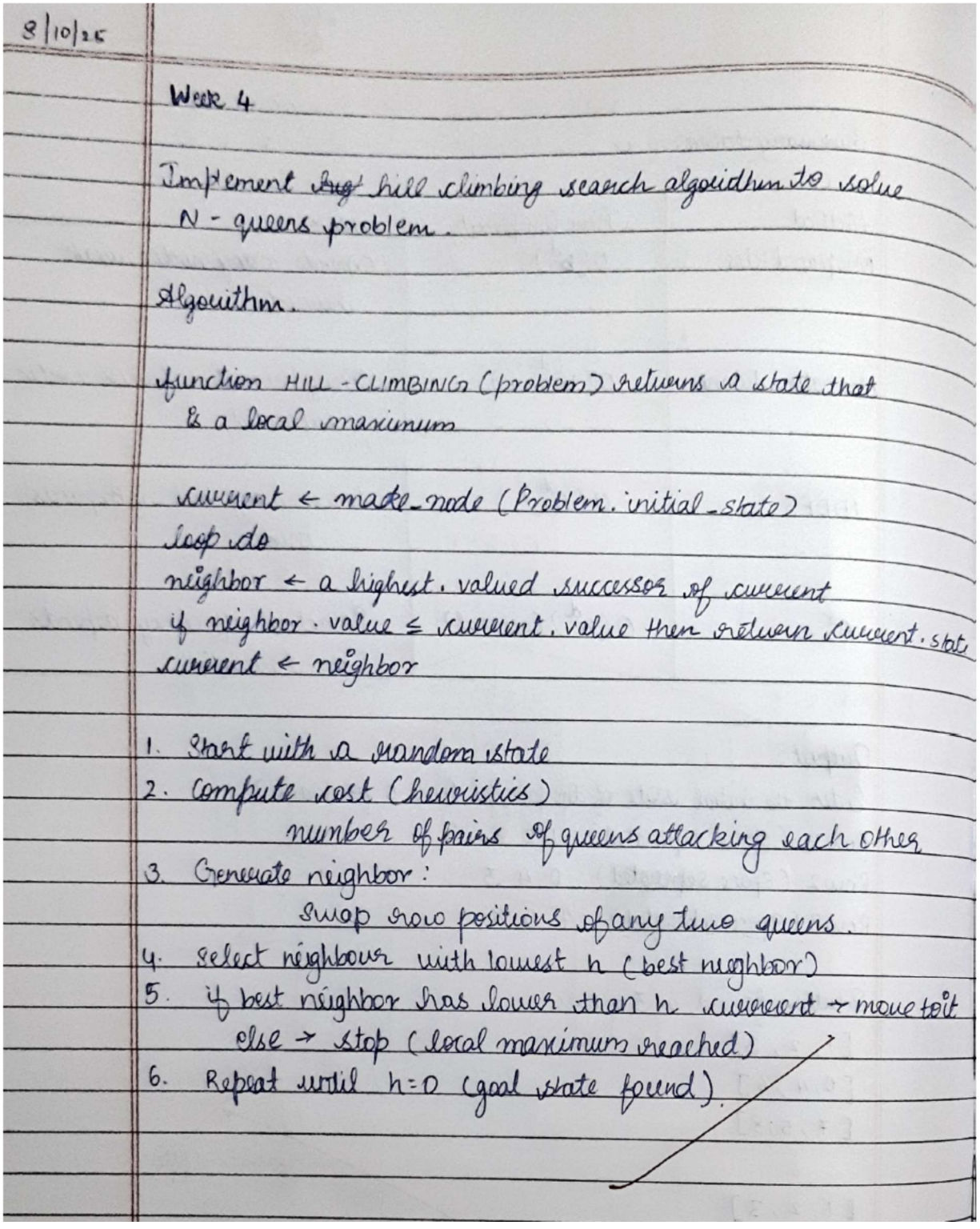
Output :

```
Enter the initial state of the puzzle (use 0 for blank):
Row 1 (space separated):  1 2 3
Row 2 (space separated):  0 4 6
Row 3 (space separated):  7 5 8

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

✅ Solution found in 3 moves:
[1, 2, 3]
[0, 4, 6]
[7, 5, 8]

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

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

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

Program 4

Implement Hill Climbing search algorithm to solve N-Queens problem.

Algorithm:



8/10/25

Week 4

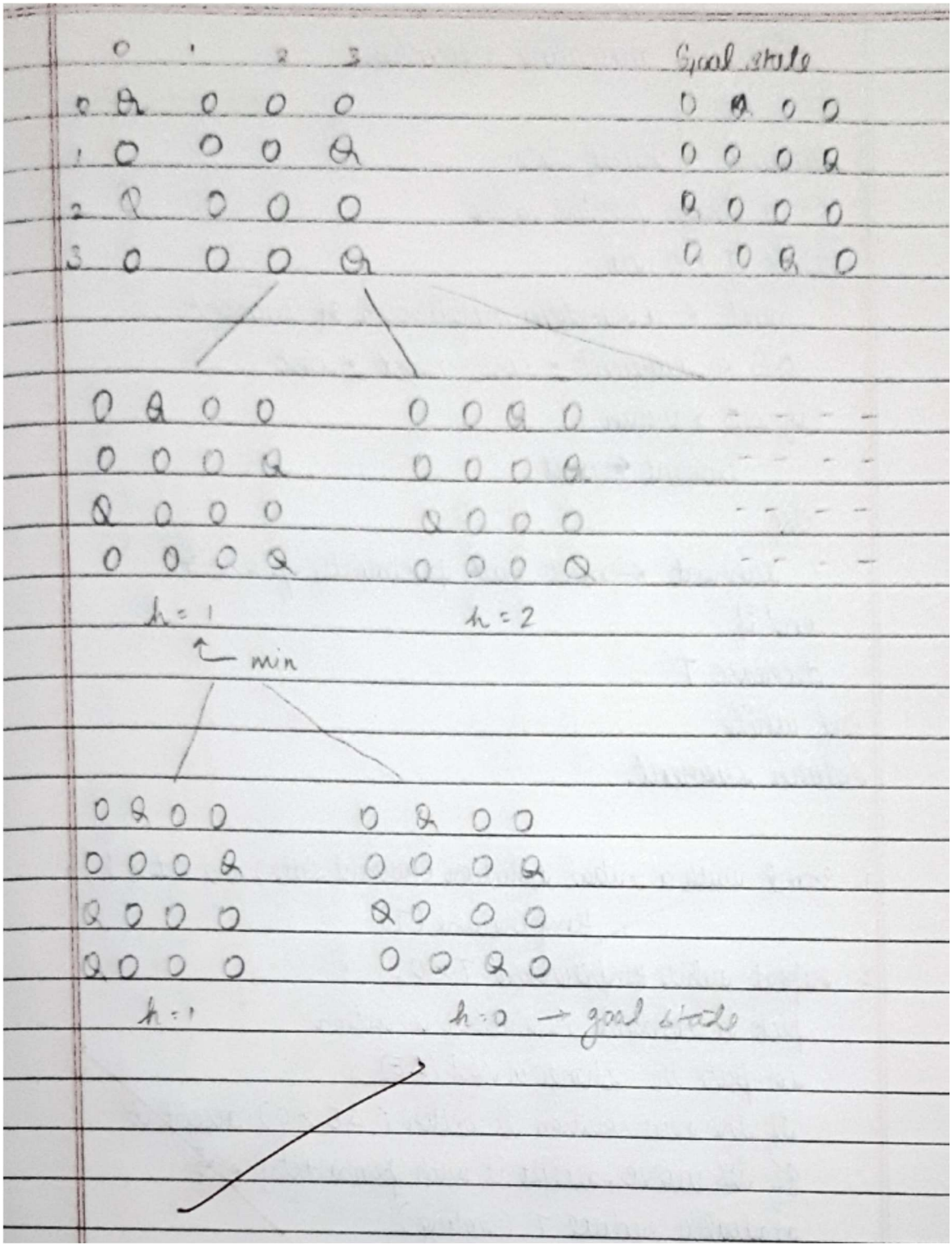
Implement ~~Big~~ hill climbing search algorithm to solve N-queens problem.

Algorithm.

function HILL-CLIMBING (problem) returns a state that is a local maximum

current ← make_node (Problem.initial_state)
loop do
neighbor ← a highest_valued successor of current
if neighbor.value ≤ current.value then return current.state
current ← neighbor

1. Start with a random state
2. Compute cost (heuristics)
   number of pairs of queens attacking each other
3. Generate neighbor:
   swap two positions of any two queens.
4. Select neighbour with lowest h (best neighbor)
5. if best neighbor has lower than h current → move to it
   else → stop (local maximum reached)
6. Repeat until h=0 (goal state found).


Goal state
h = 1
min
h = 2
h = 1
h = 0 → goal state

Code :

```
# Hill climbing search algorithm to solve N - queens problem by Abhay NY
def print_board(state):
    """Prints the 4x4 board representation with 'Q' and '.'"""
    n = len(state)
    for row in range(n):
        for col in range(n):
            if state[col] == row:
                print("Q", end=" ")
            else:
                print(".", end=" ")
        print()
    print()
def calculate_cost(state):
    """Returns number of attacking pairs of queens."""
    cost = 0
    n = len(state)
    for i in range(n):
        for j in range(i + 1, n):
            # same row
            if state[i] == state[j]:
                cost += 1
            # same diagonal
            elif abs(state[i] - state[j]) == abs(i - j):
                cost += 1
    return cost
def get_neighbors(state):
    """Generates all neighbors by swapping two queen positions."""
    neighbors = []
    n = len(state)
    for i in range(n):
        for j in range(i + 1, n):
```

```
            neighbor = state.copy()
            neighbor[i], neighbor[j] = neighbor[j], neighbor[i]
            neighbors.append((neighbor, (i, j)))
    return neighbors
def hill_climbing(state):
    print("\nInitial State:", state)
    print_board(state)
    current_cost = calculate_cost(state)
    step = 1
    while True:
        print(f"Step {step}: Current cost = {current_cost}")
        neighbors = get_neighbors(state)
        neighbor_costs = []
        # Calculate cost for all neighbors
        for neighbor, swapped in neighbors:
            cost = calculate_cost(neighbor)
            neighbor_costs.append((cost, neighbor, swapped))
        # Sort by cost and then by smallest column pair as per rules
        neighbor_costs.sort(key=lambda x: (x[0], x[2][0], x[2][1]))
        # Display neighbor costs
        print("Neighbor states and their costs:")
        for cost, neighbor, swapped in neighbor_costs:
            print(f"Swap x{swapped[0]} & x{swapped[1]} => {neighbor}, Cost = {cost}")
        best_cost, best_state, swap = neighbor_costs[0]
        print("\nBest Neighbor after swap", swap, "is", best_state, "with cost =", best_cost)
        print_board(best_state)
        if best_cost >= current_cost:  # No improvement (local minimum)
            print("No better neighbor found. Hill Climbing terminated.")
            print("Final state:", state)
            print_board(state)
            break
        else:
```

```
            state = best_state
            current_cost = best_cost
        if current_cost == 0:
            print("Goal state reached!")
            print_board(state)
            break
        step += 1
if __name__ == "__main__":
    print("Hill Climbing for 4-Queens Problem")
    print("Enter the row positions of 4 queens (each between 0 and 3):")
    state = list(map(int, input("Example (1 2 0 3): ").split()))
    hill_climbing(state)
```

Output :

```
Hill Climbing for 4-Queens Problem
Enter the row positions of 4 queens (each between 0 and 3):
Example (1 2 0 3):  1 2 0 3

Initial State: [1, 2, 0, 3]
. . Q .
Q . . .
. Q . .
. . . Q

Step 1: Current cost = 1
Neighbor states and their costs:
Swap x1 & x3 => [1, 3, 0, 2], Cost = 0
Swap x0 & x2 => [0, 2, 1, 3], Cost = 2
Swap x0 & x3 => [3, 2, 0, 1], Cost = 2
Swap x1 & x2 => [1, 0, 2, 3], Cost = 2
Swap x0 & x1 => [2, 1, 0, 3], Cost = 4
Swap x2 & x3 => [1, 2, 3, 0], Cost = 4

Best Neighbor after swap (1, 3) is [1, 3, 0, 2] with cost = 0
. . Q .
Q . . .
. . . Q
. Q . .

Goal state reached!
. . Q .
Q . . .
. . . Q
. Q . .
```

Program 5

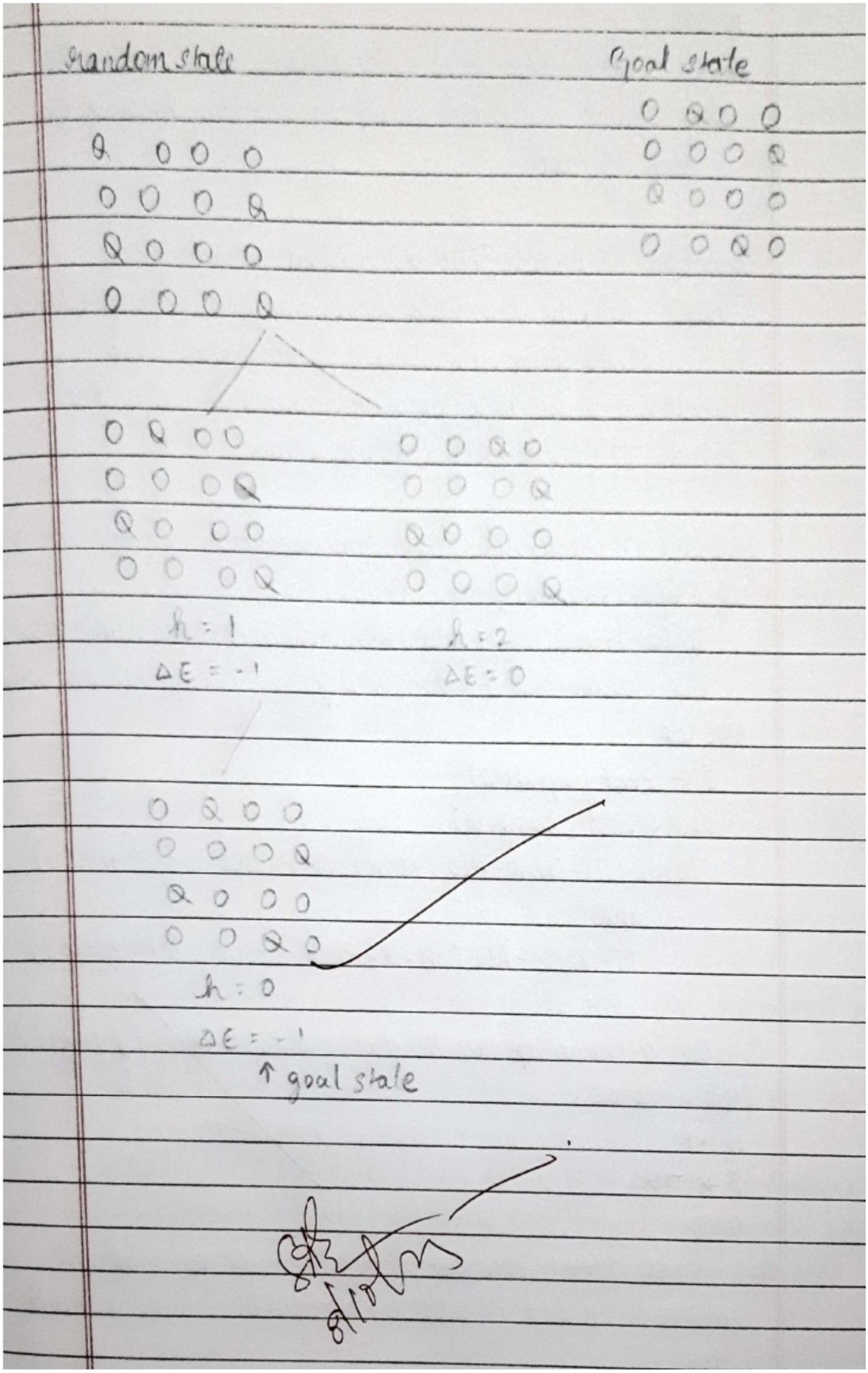
Simulated Annealing to Solve 8-Queens problem.

Algorithm:

Simulated annealing algorithm

current ← initial state
T ← a large positive value
while T > 0 do
  next ← a random neighbour of current
  ΔE ← current.cost - next.cost
  if ΔE > 0 then
    current ← next
  else
    current ← next with probability $p = e^{\frac{\Delta E}{T}}$
  end if
  decrease T
end while
return current.

1. Start with a initial solution (current state) and set a high temperature (T)
2. Repeat while temperature $T > 0$:
   pick a random neighbour solution
   compute the change in cost (ΔE)
   If the new solution is better ($\Delta E > 0$) accept it
   If its worse, accept it with probability $e^{\frac{\Delta E}{t}}$
   gradually reduce T (cooling)
3. return the best solution found (current state)

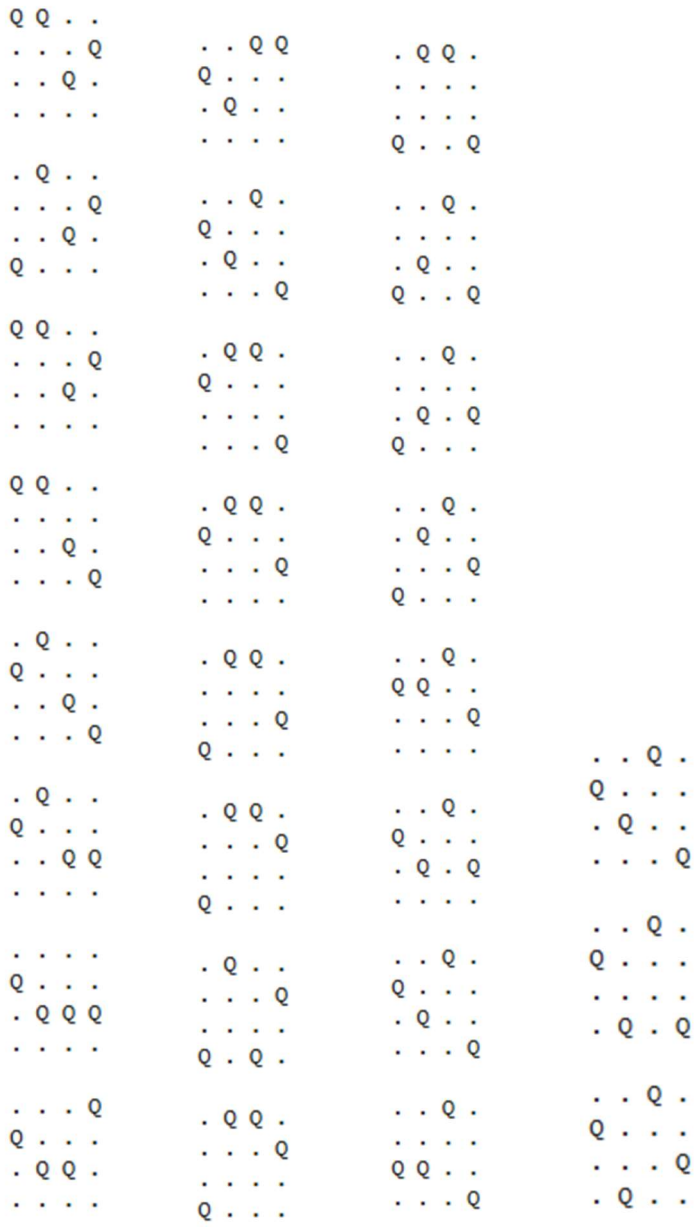

Random state
Goal state
h = 1
ΔE = -1
h = 2
ΔE = 0
h = 0
ΔE = -1
↑ goal state

Code :

```
# Simulated Annealing algorithm by Abhay NY
import random, math
random.seed(42)
def heuristic(state):
    n = len(state)
    conflicts = 0
    for i in range(n):
        for j in range(i + 1, n):
            if state[i] == state[j] or abs(state[i] - state[j]) == abs(i - j):
                conflicts += 1
    return conflicts
def print_board(state):
    n = len(state)
    for r in range(n):
        print(' '.join('Q' if state[c] == r else '.' for c in range(n)))
    print()
def simulated_annealing(N=4, T0=5.0, alpha=0.95, max_iters=200):
    state = [random.randrange(N) for _ in range(N)]
    cur_h = heuristic(state)
    T = T0
    print_board(state)
    for _ in range(max_iters):
        col = random.randrange(N)
        new_row = random.randrange(N)
        while new_row == state[col]:
            new_row = random.randrange(N)
        new_state = state.copy()
        new_state[col] = new_row
        new_h = heuristic(new_state)
        delta = new_h - cur_h
        if delta <= 0 or random.random() < math.exp(-delta / T):
```

```
            state = new_state
            cur_h = new_h
            print_board(state)
            if cur_h == 0:
                break
        T *= alpha
simulated_annealing(4)
```

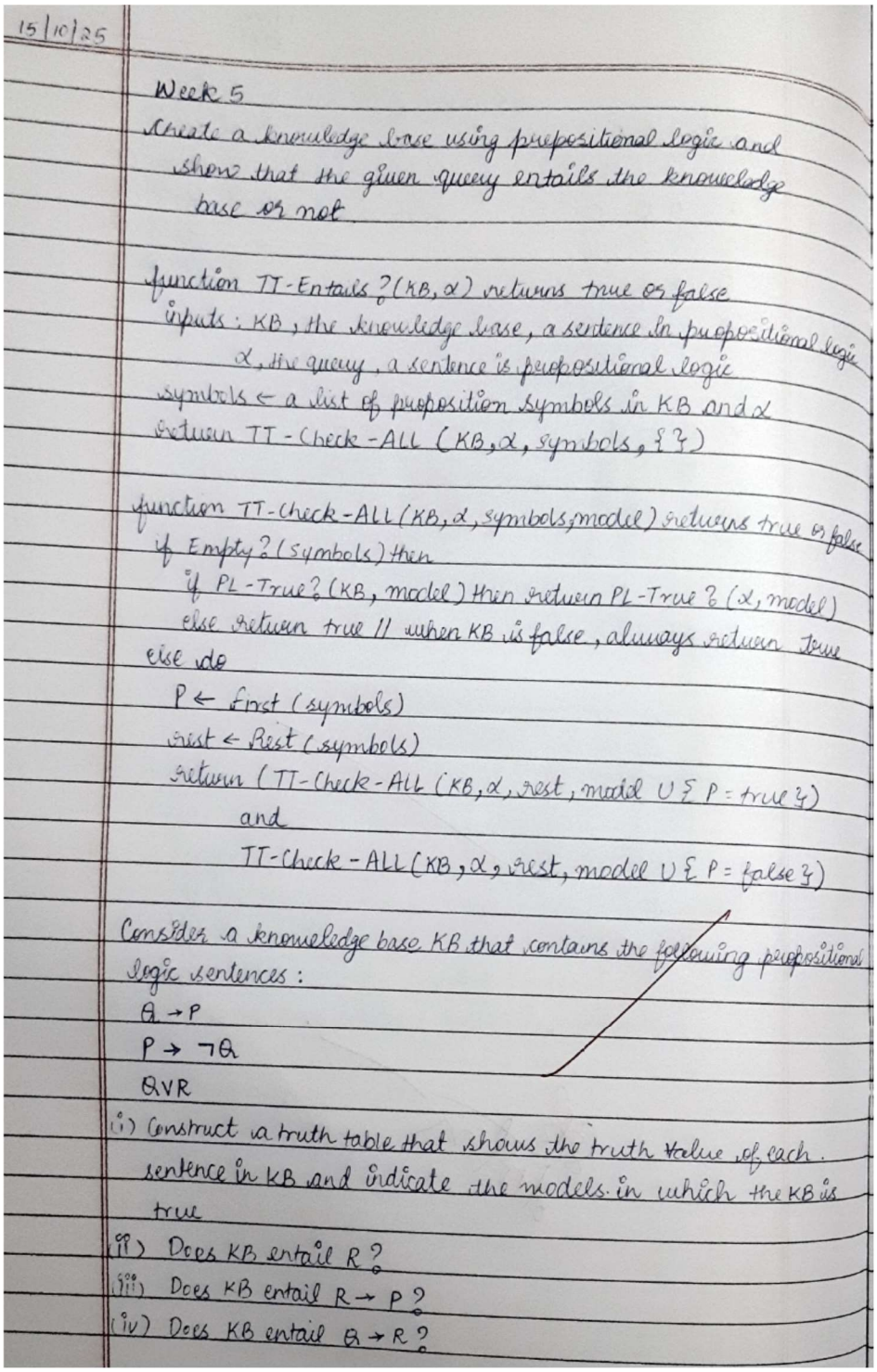
Output :



Program 6

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

Algorithm:



15/10/25

Week 5

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

function TT-Entails?(KB, α) returns true or false
inputs: KB, the knowledge base, a sentence in propositional logic
α, the query, a sentence in propositional logic
symbols ← a list of proposition 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 // when KB 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, rest, model ∪ {P = false}))

Consider a knowledge base KB that contains the following propositional logic sentences:
Q → P
P → ¬Q
Q ∨ R
(i) Construct a truth table that shows the truth value of each sentence in KB and indicate the models in which the KB is true
(ii) Does KB entail R?
(iii) Does KB entail R → P?
(iv) Does KB entail Q → R?

Q	P	
T	T	T
~~F~~	F	T
F	T	T
T	F	F

OR → + ∨
AND → × ∧

store
67 _____/_____

(i)

				S1	S2	S3	KB	
P	Q	R	¬Q	Q→P	P→¬Q	Q∨R	(S1∧S2∧S3)	
T	T	T	F	T	F	T	F	
T	T	~~F~~	F	T	F	T	F	
T	F	T	T	T	T	T	T	✓ M1
T	F	F	T	T	T	F	F	
F	T	T	F	F	T	T	F	
F	T	F	F	F	T	T	F	
F	F	T	T	T	T	T	T	✓ M2
F	F	F	T	T	T	F	F	

The models in which the KB is true are the rows where the final KB column is T. There are two such model
model 1 : P is true, Q is false, R is true
model 2 : P is false, Q is false, R is true

(ii) To check if the KB entails R, we need to see if R is true in every model where the KB is true.
Model 1 (P = T, Q = F, R = T), R is true
Model 2 (P = F, Q = F, R = T) R is true
since R is true in all models of the KB, yes, KB entails R.

(iii) we check if the sentence R → P is true in every model where the KB is true
model 1 (P = T, Q = F, R = T) the query R → P becomes T → T which is true
Model 2 (P = F, Q = F, R = T) the query R → P becomes T → F which is False.
because query is false in one of the KB's model no. KB does not entail R → P

15/10/25

Code :

 # Knowledge base_using_pre_positional_logic problem by Abhay NY

```
import itertools
def solve_and_print_table():
    """
    Constructs a full truth table for the given KB, finds its models,
    and checks for entailment of several queries.
    """
    symbols = ['P', 'Q', 'R']
    # --- 1. Define Knowledge Base (KB) and Queries ---
    # Implication A -> B is logically equivalent to (not A or B).
    s1 = lambda m: not m['Q'] or m['P']        # Q -> P
    s2 = lambda m: not m['P'] or not m['Q']    # P -> ~Q
    s3 = lambda m: m['Q'] or m['R']            # Q v R
    # The KB is true only if all its sentences are true.
    kb_true = lambda m: s1(m) and s2(m) and s3(m)
    queries = {
        "R": lambda m: m['R'],
        "R -> P": lambda m: not m['R'] or m['P'],
        "Q -> R": lambda m: not m['Q'] or m['R']
    }
    # --- 2. Generate and Print the Truth Table ---
    print("--- i) Complete Truth Table ---")
    # Helper to format boolean values for printing
    def format_bool(val):
        return "T" if val else "F"

    # Print table header
    header = f"| {'P':^3} | {'Q':^3} | {'R':^3} || {'Q->P':^6} | {'P->~Q':^7} | {'QvR':^5} || {'KB':^4} |"
    print(header)
    print("-" * len(header))
```

```
    # Generate all possible models (rows of the table)
    all_models = [dict(zip(symbols, V)) for V in itertools.product([True, False], repeat=len(symbols))]
    kb_models = []
    for model in all_models:
        # Calculate truth value for each sentence in the current model
        s1_val = s1(model)
        s2_val = s2(model)
        s3_val = s3(model)
        kb_val = kb_true(model)
        # If the KB is true in this model, save it for the entailment check
        if kb_val:
            kb_models.append(model)
            # Add a marker to highlight the KB models in the table
            marker = "<- MODEL"
        else:
            marker = ""
        # Print the current row of the truth table
        row = (f"| {format_bool(model['P']):^3} | {format_bool(model['Q']):^3} |
{format_bool(model['R']):^3} || "
               f"{format_bool(s1_val):^6} | {format_bool(s2_val):^7} | {format_bool(s3_val):^5} || "
               f"{format_bool(kb_val):^4} | {marker}")
        print(row)

    print("-" * len(header))
    print(f"\n The models where the KB is true are:")
    for i, model in enumerate(kb_models):
        print(f"  Model {i+1}: {model}")
    # --- 3. Check Entailment for Each Query ---
    print("\n\n--- ii), iii), iv) Entailment Checks ---")
    for name, query_func in queries.items():
        # Entailment holds if the query is true in ALL models of the KB.
        entails = all(query_func(model) for model in kb_models)
```

```
    print(f"\n Does KB entail '{name}'?")
    if entails:
        print(f"    Yes. The query '{name}' is true in all models of the KB.")
    else:
        # Find a specific counterexample to show why it fails
        counterexample = next(m for m in kb_models if not query_func(m))
        print(f"   No. A counterexample was found.")
        print(f"   In model {counterexample}, the KB is true but the query '{name}' is false.")
solve_and_print_table()
```

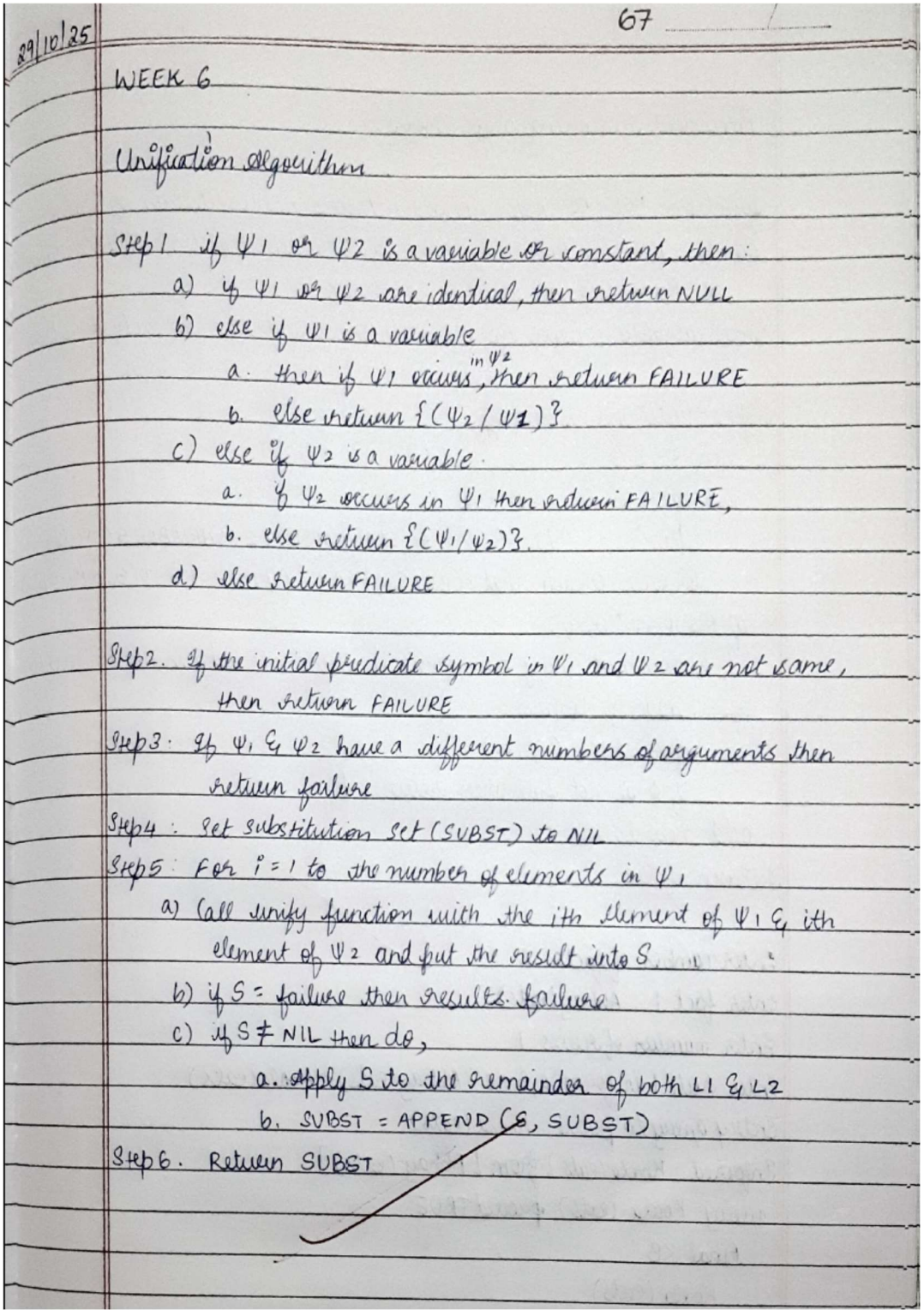
Output :

```
--- i) Complete Truth Table ---
|  P  |  Q  |  R  ||  Q->P  |  P->~Q  |  QvR  ||  KB  |
-----------------------------------------------------
|  T  |  T  |  T  ||   T    |    F    |   T   ||  F   |
|  T  |  T  |  F  ||   T    |    F    |   T   ||  F   |
|  T  |  F  |  T  ||   T    |    T    |   T   ||  T   | <- MODEL
|  T  |  F  |  F  ||   T    |    T    |   F   ||  F   |
|  F  |  T  |  T  ||   F    |    T    |   T   ||  F   |
|  F  |  T  |  F  ||   F    |    T    |   T   ||  F   |
|  F  |  F  |  T  ||   T    |    T    |   T   ||  T   | <- MODEL
|  F  |  F  |  F  ||   T    |    T    |   F   ||  F   |
-----------------------------------------------------

 The models where the KB is true are:
  Model 1: {'P': True, 'Q': False, 'R': True}
  Model 2: {'P': False, 'Q': False, 'R': True}


--- ii), iii), iv) Entailment Checks ---

 Does KB entail 'R'?
    Yes. The query 'R' is true in all models of the KB.

 Does KB entail 'R -> P'?
   No. A counterexample was found.
   In model {'P': False, 'Q': False, 'R': True}, the KB is true but the query 'R -> P' is false.

 Does KB entail 'Q -> R'?
    Yes. The query 'Q -> R' is true in all models of the KB.
```

Program 7

Implement unification in first order logic.

Algorithm:



29/10/25

67

WEEK 6

Unification Algorithm

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 NULL
- 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 & Ψ_2 have a different numbers 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 & ith element of Ψ_2 and put the result into S.
- b) if S = failure then results failure
- c) if S ≠ NIL then do,
  - a. Apply S to the remainder of both L1 & L2
  - b. SUBST = APPEND (S, SUBST)

Step 6. Return SUBST

Unification Algorithm problem.

1. $P(f(x), g(y), y)$
$P(f(g(z)), g(f(a), f(a))$
find θ (MGU)

2. $Q(x, f(x))$
$Q(f(y), y)$

3. $H(x, g(x))$
$H(g(y), g(g(z)))$

1. Compare first arguments
we need to unify $y(x)$ and $y(g)$
function symbol y matches
unify their arguments x & z
substitute z for x (or x for z) lets choose x/z
Substitution $\theta_1 = \{x/z\}$
current MGU $\theta = \{x/z\}$

Compare second arguments
we need to unify $g(y)$ & $g(f(a))$
symbol (g) match
unify their argument as y is a variable & $f(a)$ is a term
substitution $\theta_2 = \{y/f(a)\}$
update MGU we compose θ & θ_2.
$\theta = \theta \cup \theta_2 = \{x/z, y/f(a)\}$

Compare 3rd argument
must unify the 3rd argument after applying the current
MGU $\theta = \{x/z, y/f(a)\}$
third argument of E1 : y
applying θ, $y\{\theta\} \rightarrow f(a)$
third argument of E2 : f(a)
applying θ, $f(a)\{\theta\} \rightarrow f(a)$
the two resulting arguments $f(a)$ & $f(a)$ are identical
Since both expression become identical the unification is correct

2. ~~Q(x, f(x))~~
~~Q(f(y), g(g(z)))~~ Q(f(y), y)

3. ~~Not~~ Unifiable
cannot substitute the variable y with the term f(y) because the variable y occurs inside the term f(y)
this is a failure of the occurs check.

2. this is known as occurs check attempting this substitution would lead to an infinite loop ($y \rightarrow f(y) \rightarrow f(f(y)) \rightarrow \ldots$)

final result because the occurs check fails these two expressions cannot be unified
No MGU (most general unifier) exists

3/11/23

Code :

#Unifiation code by Abhay NY

```
import re
def is_variable(term):
    """A variable starts with a lowercase letter."""
    return term[0].islower()
def parse_predicate(expr):
    """Parse predicate like P(x,y) -> ('P', ['x', 'y'])"""
    expr = expr.strip()
    match = re.match(r"(\w+)\((.*)\)", expr)
    if not match:
        return expr, []  # constant or symbol
    pred_name = match.group(1)
    args = [arg.strip() for arg in match.group(2).split(",")]
    return pred_name, args
def unify(expr1, expr2, subs=None):
    """Main unification algorithm."""
    if subs is None:
        subs = {}
    # Apply current substitutions
    for var in subs:
        expr1 = expr1.replace(var, subs[var])
        expr2 = expr2.replace(var, subs[var])
    name1, args1 = parse_predicate(expr1)
    name2, args2 = parse_predicate(expr2)
    # If predicates are different, fail
    if name1 != name2:
        return None
    # If no arguments (constants)
    if not args1 and not args2:
        return subs if name1 == name2 else None
```

```
    # If number of arguments differs
    if len(args1) != len(args2):
        return None
    for a1, a2 in zip(args1, args2):
        if a1 == a2:
            continue
        elif is_variable(a1):
            subs[a1] = a2
        elif is_variable(a2):
            subs[a2] = a1
        else:
            # Check recursively for complex terms
            res = unify(a1, a2, subs)
            if res is None:
                return None
    return subs
# -------- MAIN PROGRAM --------
if __name__ == "__main__":
    print("=== FIRST ORDER LOGIC UNIFICATION ===")
    expr1 = input("Enter first expression: ")
    expr2 = input("Enter second expression: ")
    result = unify(expr1, expr2)
    print("\n-----------------------------")
    if result:
        print("Unification Successful!")
        print("Substitutions:")
        for var, val in result.items():
            print(f"  {var} / {val}")
    else:
        print("Unification Failed.")
    print("-----------------------------")
```

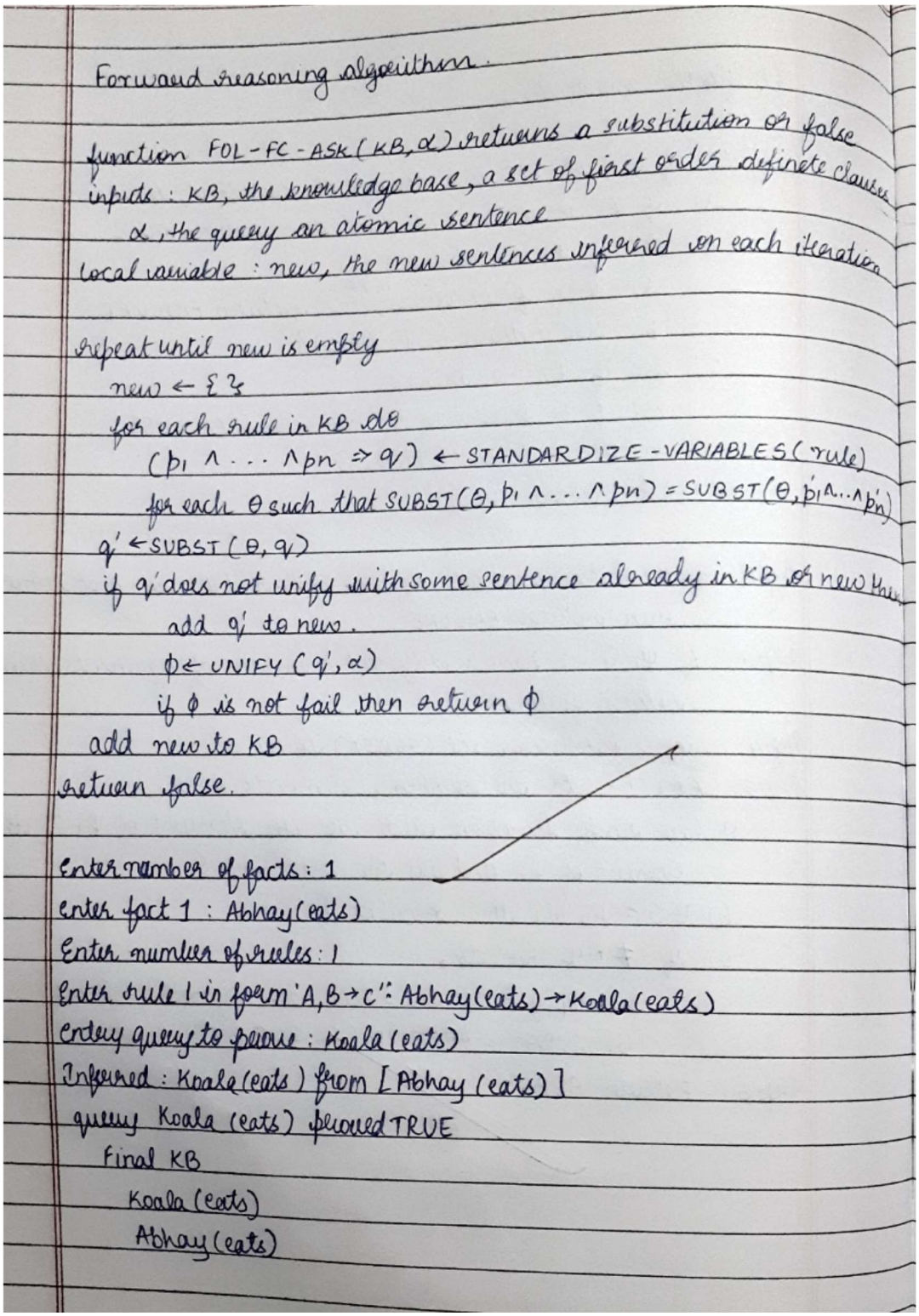
Output :

```
=== FIRST ORDER LOGIC UNIFICATION ===
Enter first expression:  P(f(x),g(y),y)
Enter second expression:  P(f(g(z)),g(f(a)),f(a))

------------------------------
Unification Successful!
Substitutions:
  f(x) / f(g(z))
  g(y) / g(f(a))
  y / f(a)
```

Program 8

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

Algorithm:



Forward reasoning algorithm.

function FOL-FC-ASK(KB, α) returns a substitution or false
inputs : KB, the knowledge base, a set of first order definite clauses
α, the query an atomic sentence
local variable : new, the new sentences inferred on each iteration

repeat until new is empty
new ← { }
for each rule in KB do
$(p_1 \wedge \ldots \wedge p_n \Rightarrow q) \leftarrow$ STANDARDIZE-VARIABLES(rule)
for each θ such that SUBST(θ, $p_1 \wedge \ldots \wedge p_n$) = SUBST(θ, $p'_1 \wedge \ldots \wedge p'_n$)
$q' \leftarrow$ SUBST(θ, q)
if q' does not unify with some sentence already in KB or new then
add q' to new.
$\phi \leftarrow$ UNIFY(q', α)
if ϕ is not fail then return ϕ
add new to KB
return false.

Enter number of facts : 1
enter fact 1 : Abhay(eats)
Enter number of rules : 1
Enter rule 1 in form 'A,B→C' : Abhay(eats)→Koala(eats)
enter query to prove : Koala(eats)
Inferred : Koala(eats) from [Abhay(eats)]
query Koala(eats) proved TRUE
Final KB
Koala(eats)
Abhay(eats)

Code :

 # Simple Forward Chaining (Forward Reasoning) System By Abhay NY

```
class KnowledgeBase:
    def __init__(self):
        self.facts = set()
        self.rules = []
    def add_fact(self, fact):
        self.facts.add(fact.strip())
    def add_rule(self, premise, conclusion):
        premises = [p.strip() for p in premise]
        self.rules.append((premises, conclusion.strip()))
    def forward_chain(self, query):
        query = query.strip()
        new_inferred = True
        print("\nStarting Forward Chaining...\n")
        while new_inferred:
            new_inferred = False
            for premise, conclusion in self.rules:
                if all(p in self.facts for p in premise) and conclusion not in self.facts:
                    print(f"Inferred: {conclusion} from {premise}")
                    self.facts.add(conclusion)
                    new_inferred = True
                    if query in self.facts:
                        print(f"\n Query '{query}' proved TRUE!")
                        return True

        print(f"\n Query '{query}' could NOT be proved.")
        return False
if __name__ == "__main__":
    kb = KnowledgeBase()
    print("=== FORWARD REASONING SYSTEM ===")
```

```
# Input Facts
n = int(input("Enter number of facts: "))
for i in range(n):
    fact = input(f"Enter fact {i+1}: ")
    kb.add_fact(fact)
# Input Rules
m = int(input("\nEnter number of rules: "))
for i in range(m):
    rule_input = input(f"Enter rule {i+1} in form 'A,B -> C': ")
    lhs, rhs = rule_input.split("->")
    premises = [p.strip() for p in lhs.split(",") if p.strip()]
    conclusion = rhs.strip()
    kb.add_rule(premises, conclusion)
# Input Query
query = input("\nEnter query to prove: ")
# Run forward reasoning
kb.forward_chain(query)
# Show final facts
print("\nFinal Knowledge Base Facts:")
for fact in kb.facts:
    print(" -", fact)
```

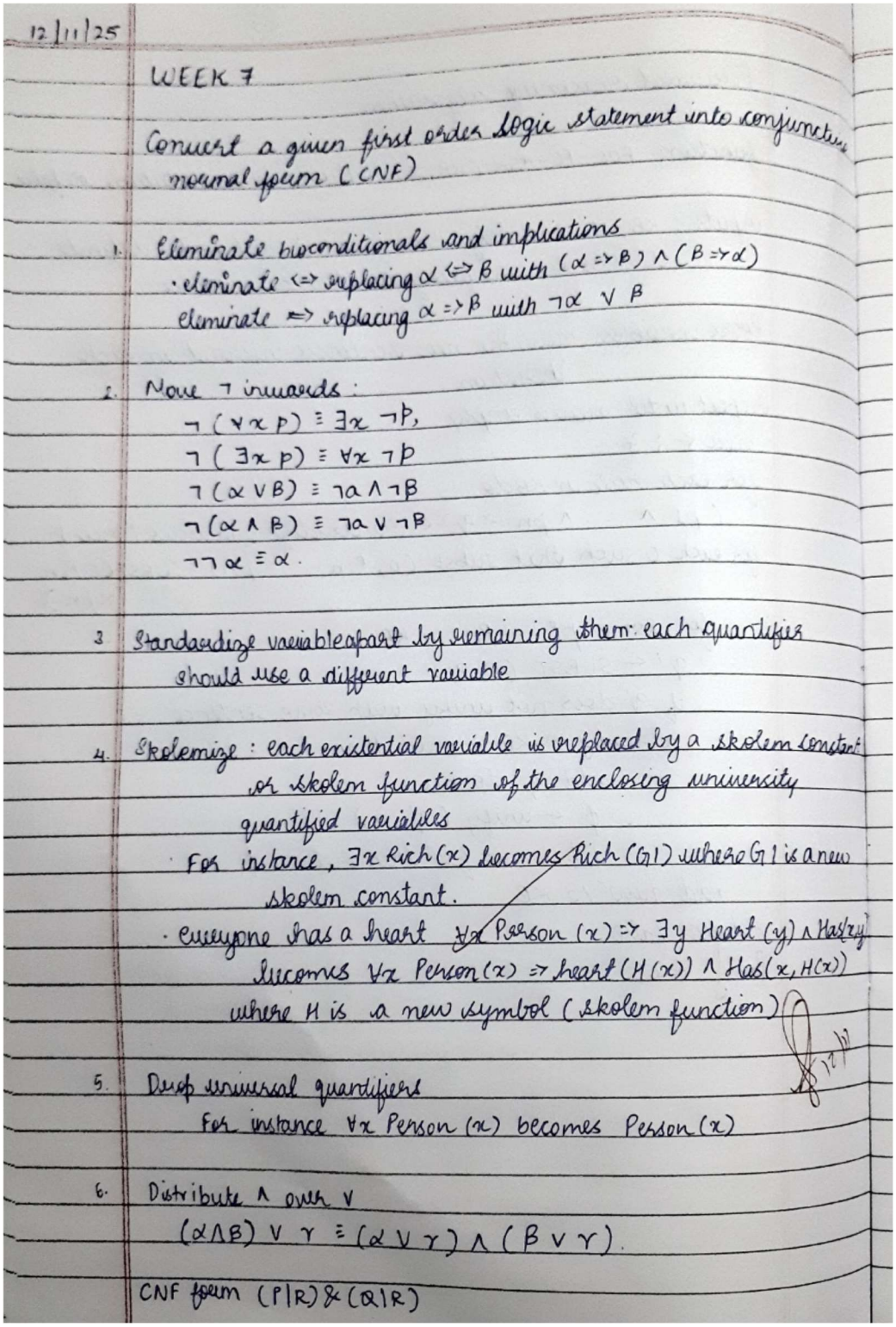
Output :

```
=== FORWARD REASONING SYSTEM ===
Enter number of facts:  1
Enter fact 1:  Abhay(eats)

Enter number of rules:  1
Enter rule 1 in form 'A,B -> C':  Abhay(eats)->Koala(eats)

Enter query to prove:  Koala(eats)

Starting Forward Chaining...

Inferred: Koala(eats) from ['Abhay(eats)']

 Query 'Koala(eats)' proved TRUE!

Final Knowledge Base Facts:
 - Koala(eats)
 - Abhay(eats)
```

Program 9

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

Algorithm:



12/11/25

WEEK 7

Convert a given first order logic statement into conjunctive normal form (CNF)

1. Eliminate biconditionals and implications
   - eliminate $\Leftrightarrow$ replacing $\alpha \Leftrightarrow \beta$ with $(\alpha \Rightarrow \beta) \wedge (\beta \Rightarrow \alpha)$
   - eliminate $\Rightarrow$ replacing $\alpha \Rightarrow \beta$ with $\neg\alpha \vee \beta$

2. Move $\neg$ inwards:
   $\neg(\forall x\, p) \equiv \exists x\, \neg p$,
   $\neg(\exists x\, p) \equiv \forall x\, \neg p$
   $\neg(\alpha \vee \beta) \equiv \neg\alpha \wedge \neg\beta$
   $\neg(\alpha \wedge \beta) \equiv \neg\alpha \vee \neg\beta$
   $\neg\neg\alpha \equiv \alpha$.

3. Standardize variable apart by renaming them: each quantifier should use a different variable

4. Skolemize: each existential variable is replaced by a skolem constant or skolem function of the enclosing universally quantified variables
   - For instance, $\exists x$ Rich(x) becomes Rich(G1) where G1 is a new skolem constant.
   - Everyone has a heart $\forall x$ Person(x) $\Rightarrow \exists y$ Heart(y) $\wedge$ Has(x,y)
     becomes $\forall x$ Person(x) $\Rightarrow$ Heart(H(x)) $\wedge$ Has(x, H(x))
     where H is a new symbol (skolem function)

5. Drop universal quantifiers
   For instance $\forall x$ Person(x) becomes Person(x)

6. Distribute $\wedge$ over $\vee$
   $(\alpha \wedge \beta) \vee \gamma \equiv (\alpha \vee \gamma) \wedge (\beta \vee \gamma)$.

CNF form (P|R) & (Q|R)

Code :

```
# CNF by Abhay NY
from sympy import symbols
from sympy.logic.boolalg import Implies, Equivalent, Not, And, Or, to_cnf
# Step 1: Define logical variables
P, Q, R = symbols('P Q R')
# Step 2: Define your logical statement
# Example: (P ⇒ Q) ⇔ (¬Q ⇒ ¬P)
statement = Equivalent(Implies(P, Q), Implies(Not(Q), Not(P)))
print("Original Statement:")
print(statement)
# Step 3: Convert to CNF using Sympy
cnf_form = to_cnf(statement, simplify=True)
print("\nConverted to CNF:")
print(cnf_form)
```

Output :

```
Original Statement:
Equivalent(Implies(P, Q), Implies(~Q, ~P))

Converted to CNF:
True
```

```
Original Formula:
(Implies(~R, P)) & (Implies(P, Q | R))

CNF Form:
(P | R) & (Q | R)
```

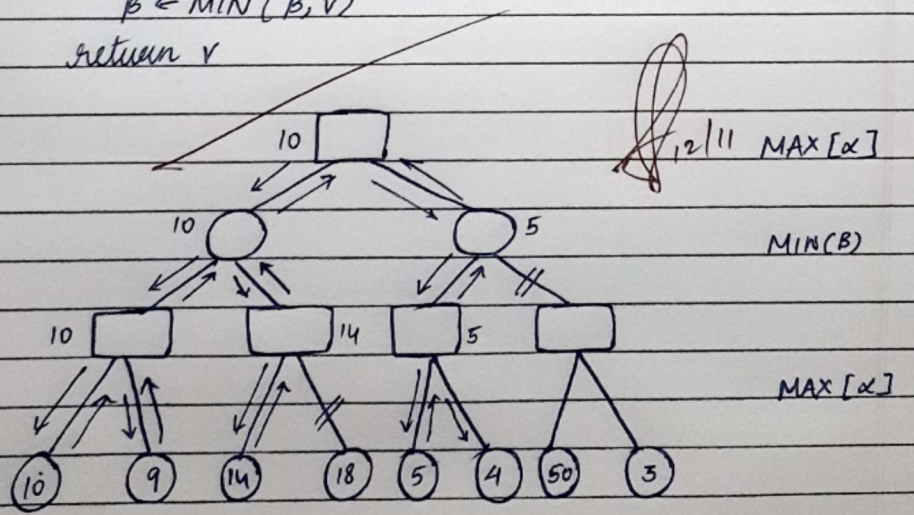
Program 10

Implement Alpha-Beta Pruning.

Algorithm:

Implement Alpha beta search algorithm

```
function ALPHA - BETA - SEARCH (state) returns an action
    V ← MAX ← VALUE (state, -∞, +∞)
return the action in ACTIONS (state) with value V

function MAX-VALUE(state, α, β) returns a utility value
    if TERMINAL - TEST (state) then return UTILITY (state)
      V ← -∞
    for each α in ACTIONS(state) do
        V ← MAX ( V, MIN - VALUE (RESULT (s,a), α, β))
        if V ≥ β then return V
        α ← MAX (α, V)
    return V

function MIN - VALUE (state, α, β) returns a utility value
    if TERMINAL - TEST (state) then return UTILITY (state)
    V ← +∞
    for each a in ACTIONS (state) do
        V ← MIN (V, MAX - VALUE (RESULT (s,a), α, β))
        if V ≤ α then return V
        β ← MIN (β, V)
    return V
```



The optimal value is 10

Code :

 # Alpha–Beta Pruning implementation for Minimax by Abhay NY

```
def alpha_beta_pruning(node, depth, alpha, beta, maximizing_player, tree):
    """
    node: current node index
    depth: current depth in the tree
    alpha: best value that the maximizer can guarantee
    beta: best value that the minimizer can guarantee
    maximizing_player: True if it's MAX's turn
    tree: dictionary {node: [child_nodes]}
    """
    # Base case: leaf node
    if node not in tree:
        return node  # leaf value (integer)
    if maximizing_player:
        max_eval = float('-inf')
        for child in tree[node]:
            eval = alpha_beta_pruning(child, depth + 1, alpha, beta, False, tree)
            max_eval = max(max_eval, eval)
            alpha = max(alpha, eval)
            if beta <= alpha:
                break  # prune
        return max_eval
    else:
        min_eval = float('inf')
        for child in tree[node]:
            eval = alpha_beta_pruning(child, depth + 1, alpha, beta, True, tree)
            min_eval = min(min_eval, eval)
            beta = min(beta, eval)
            if beta <= alpha:
                break  # prune
        return min_eval
```

```
# Example tree (based on your image)
# Structure: parent: [children]
game_tree = {
    'A': ['B', 'C'],       # Root (MAX)
    'B': ['D', 'E'],       # MIN node
    'C': ['F', 'G'],       # MIN node
    'D': [10, 9],          # MAX node
    'E': [14, 18],         # MAX node
    'F': [5, 4],           # MAX node
    'G': [50, 3]           # MAX node
}
# Start alpha-beta search
best_value = alpha_beta_pruning('A', 0, float('-inf'), float('inf'), True, game_tree)
print("The optimal value is:", best_value)
```

Output:

```
The optimal value is: 10
```

Certification on Artificial Intelligence Foundation

Infosys®
Navigate your next

CERTIFICATE OF ACHIEVEMENT

The certificate is awarded to

Abhay N Y

for successfully completing

Artificial Intelligence Foundation Certification

on November 19, 2025

Infosys | Springboard

Congratulations! You make us proud!

Satheesha B.N.
Satheesha B. Nanjappa
Senior Vice President and Head
Education, Training and Assessment
Infosys Limited

Issued on: Wednesday, November 19, 2025
To verify, scan the QR code at https://verify.onwingspan.com