VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB REPORT on

Artificial Intelligence (23CS5PCAIN)

Submitted by

Abhinav Sanjay (1BM23CS009)

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 **Aug-2025 to Dec-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 **Abhinav Sanjay (1BM23CS009),** 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.

Swati Sridharan Associate/Assistant Professor Department of CSE, BMSCE	Dr. Kavitha Sooda Professor & HOD Department of CSE, BMSCE

Index

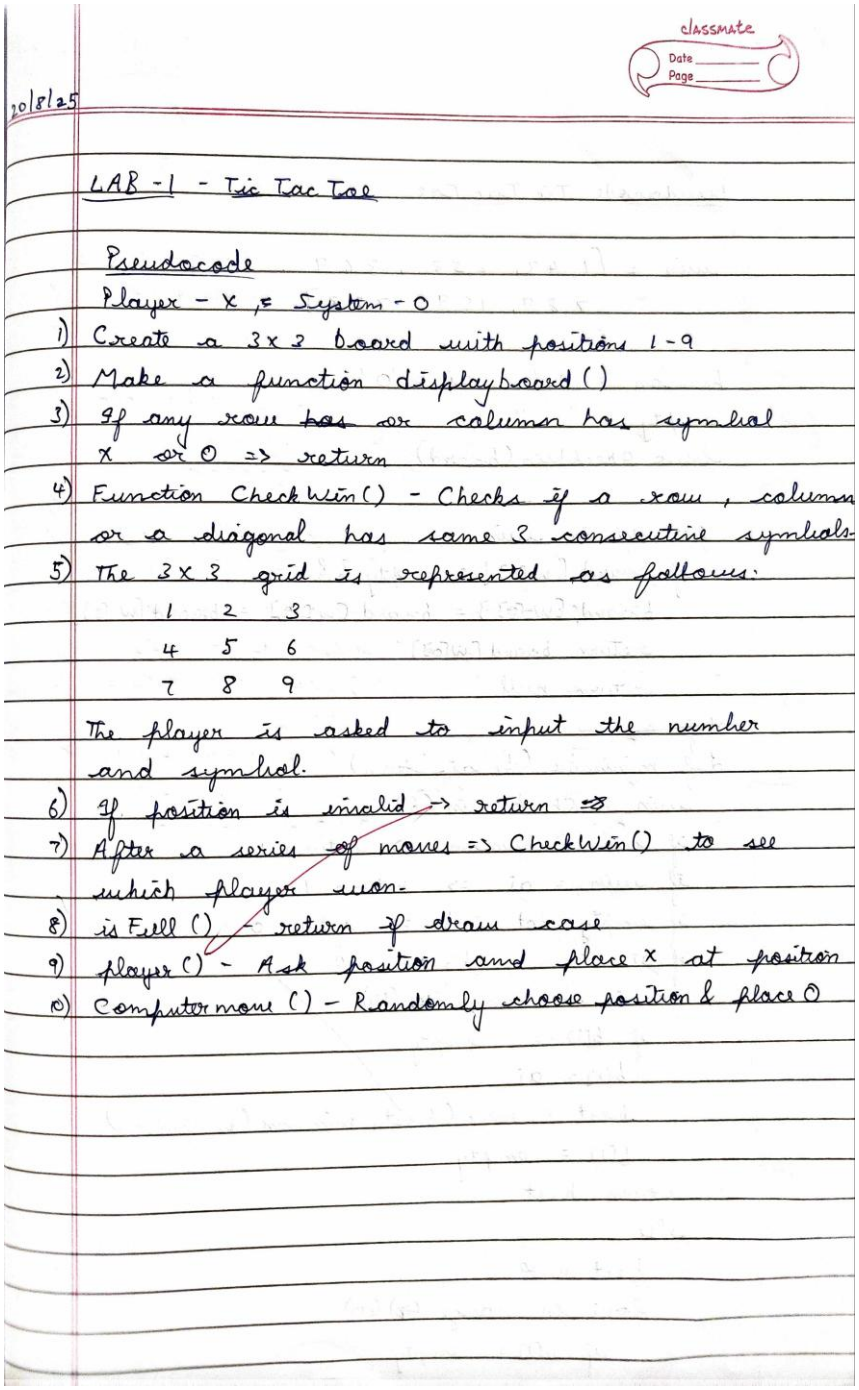
Github Link:
https://github.com/AbhinavSanjay24/AI-LAB

Program 1

Implement Tic –Tac –Toe Game

Implement vacuum cleaner agent

Algorithm:



10/8/25

LAB - 1 - Tic Tac Toe

Pseudocode

Player - X , System - O

1) Create a 3 x 3 board with positions 1-9
2) Make a function displayboard()
3) If any row has or column has symbol X or O => return
4) Function CheckWin() - Checks if a row, column or a diagonal has same 3 consecutive symbols
5) The 3 x 3 grid is represented as follows:

1 2 3
4 5 6
7 8 9

The player is asked to input the number and symbol.

6) If position is invalid -> return =>
7) After a series of moves => CheckWin() to see which player won.
8) isFull() - return if draw case
9) playerMove() - Ask position and place X at position
10) ComputerMove() - Randomly choose position & place O

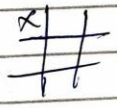
Pseudocode Tic Tac Toe

```
win = [1 4 7, 2 5 8, 3 6 9, 1 2 3, 4 5 6,
       7 8 9, 1 5 9, 7 5 3]

human = 'x', ai = 'o'
empty = " "
func checkWin(board)

 for w in win
   if board[w[0]] != empty &&
      board[w[0]] = board[w[1]] = board[w[2]]
      return board[w[0]]
      return null
~~def minm~~
def minimax(b, ai_turn)
   win = checkwin(b)
   if win = human => return 1
   if win = ai => return 1
   if empty not in b => return 0
   if ai_turn, best = -∞
   for i in range (0) (10)
      if b[i] == empty
        b[i] = ai
        best = max(best, minmax(b, false))
        b[i] = empty
   return best
   else
      best = ∞
      for i in range (0) (10)
         if b[i] = empty
```

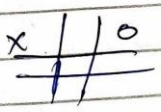
```
b[i] = human
best = min(best, minimax(b, false))
b[i] = empty
return best
def ai_move(b)
    score = -∞
    move = -1
    for i in range(10)
        if b[i] = empty
            b[i] = a
        best = minimax(b, false)
        b[i] = empty
        if best > score
            score = best
            move = i
    return move
```

Output

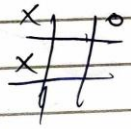
Player = 1



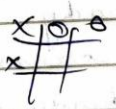
Ai = 3



Player = 4



Ai = 2



Player 7

Player wins

20/9/25

Vaccuum cleaner Algorithm

```
count = 0
pos = 1
while (count < 4)
if (room [pos] == 0) {
        clear room
  print ("Room {pos} cleaned")
}
else {
     print ("Room already cleaned")
}
 count ++
  move to next pos
}
print ("All rooms cleaned")

func clean (n) {
    c [n] = 1;
}
```

1	2
$C_1 = 0$	$C_2 = 1$
$C_4 = 0$	$C_3 = 1$
4	3

Room 0,0 was dirty, now cleaned
Room 0,1 already clean
Room 1,1 is cleaned
Room 1,0 already cleaned
All rooms cleaned

Code:

Tic Tac Toe:

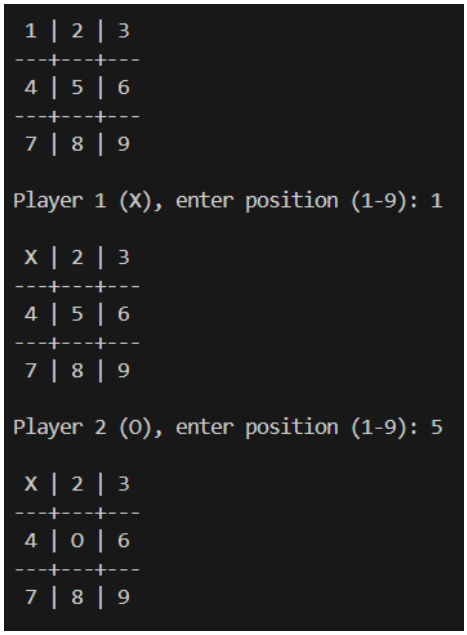
```
# 3x3 Tic Tac Toe Board

board = [['1', '2', '3'],

        ['4', '5', '6'],

        ['7', '8', '9']]


# Function to display the board

def displayBoard():

   print()

   for i in range(3):

      for j in range(3):

         print(f" {board[i][j]} ", end="")

         if j < 2:

            print("|", end="")

      print()

      if i < 2:

         print("---+---+---")

   print()


# Function to check if a player has won

def checkWin():

   # Check rows and columns

   for i in range(3):
```

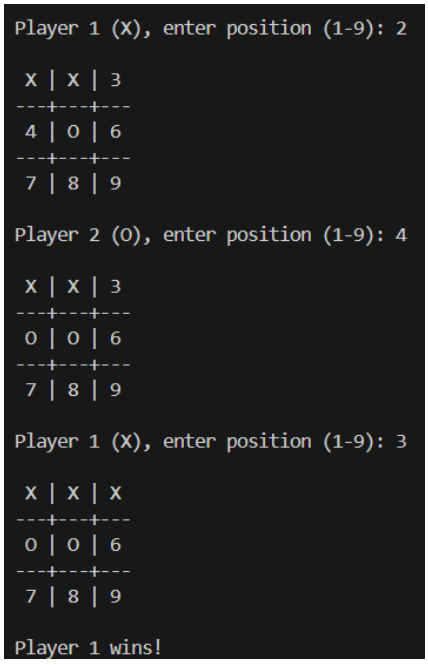
```
        if board[i][0] == board[i][1] == board[i][2]:
            return True
        if board[0][i] == board[1][i] == board[2][i]:
            return True

    # Check diagonals
    if board[0][0] == board[1][1] == board[2][2]:
        return True
    if board[0][2] == board[1][1] == board[2][0]:
        return True

    return False

# Function to check if board is full (draw)
def isFull():
    for i in range(3):
        for j in range(3):
            if board[i][j] not in ['X', 'O']:
                return False
    return True

# Main game loop
player = 1
```

```
while True:
    displayBoard()
    mark = 'X' if player == 1 else 'O'

    try:
        choice = int(input(f"Player {player} ({mark}), enter position (1-9): "))
    except ValueError:
        print("Invalid input, enter a number.")
        continue

    if choice < 1 or choice > 9:
        print("Invalid move, try again.")
        continue
    row = (choice - 1) // 3
    col = (choice - 1) % 3

    if board[row][col] in ['X', 'O']:
        print("Position already taken, try again.")
        continue
    board[row][col] = mark

    if checkWin():
        displayBoard()
        print(f"Player {player} wins!")
```

```
        break


    if isFull():
        displayBoard()
        print("It's a draw!")
        break
    player = 2 if player == 1 else 1
```

Output:





Code for Vacuum Cleaner:

```
def vacuum_cleaner(room_array):
    for room in room_array:
        label, status = room[0], room[1].lower()
        print(f"Currently in Room {label}")
        if status == "dirty":
            print(f"Room {label} is cleaned")
        else:
            print(f"Room {label} is already clean")
        print("Moving to next room...\n")
# Example input: list of rooms with their status
rooms = [('A', 'dirty'), ('B', 'clean'), ('A', 'clean'), ('B', 'dirty')]
vacuum_cleaner(rooms)
```

Output:

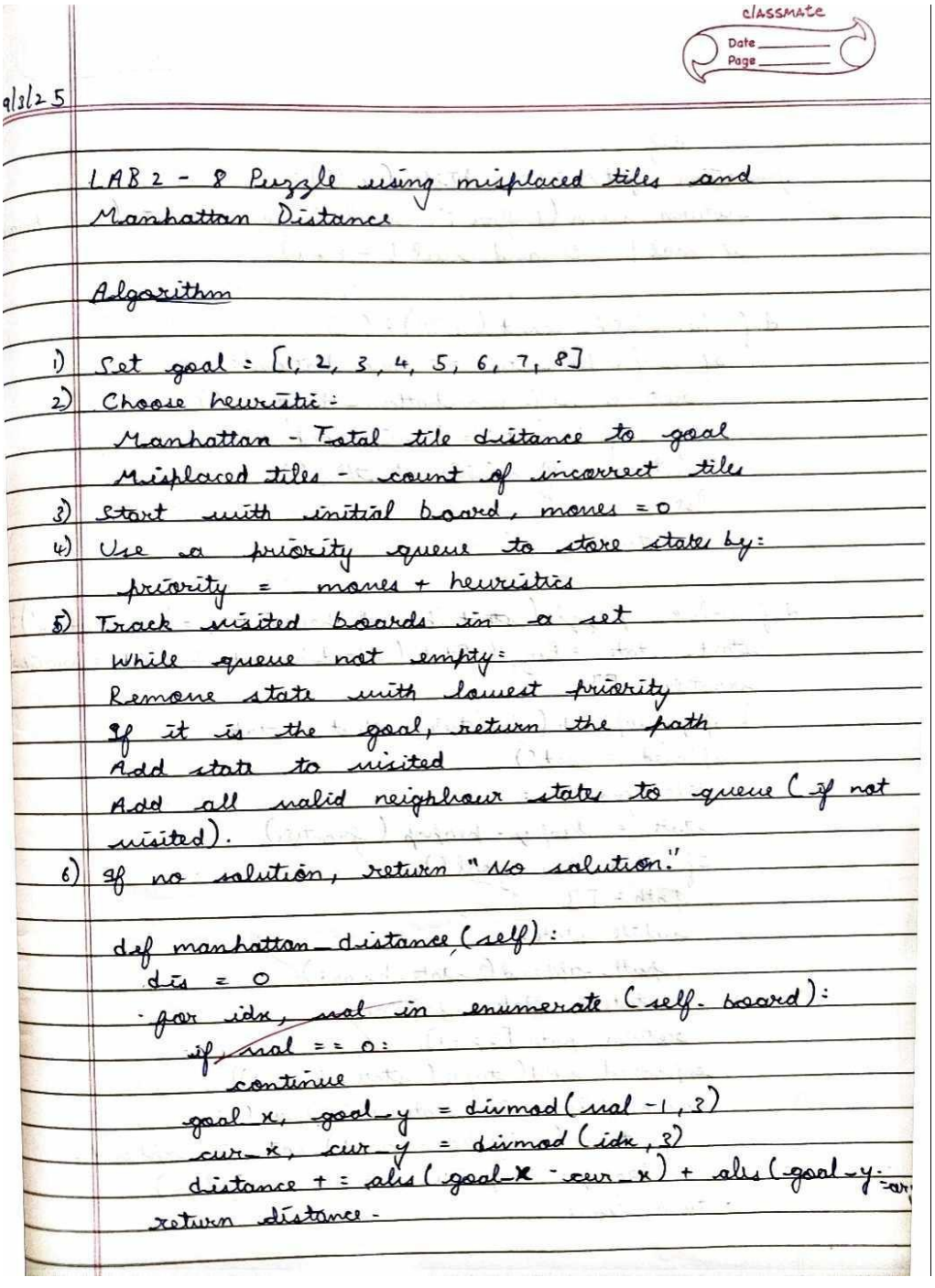
```
Currently in Room A
Room A is cleaned
Moving to next room...

Currently in Room B
Room B is already clean
Moving to next room...

Currently in Room A
Room A is already clean
Moving to next room...

Currently in Room B
Room B is cleaned
Moving to next room...
```

Program 2

Implement 8 puzzle problems using Depth First Search (DFS)

Implement Iterative deepening search algorithm

Algorithm:



9/3/25

LAB 2 - 8 Puzzle using misplaced tiles and Manhattan Distance

Algorithm

1) Set goal = [1, 2, 3, 4, 5, 6, 7, 8]
2) Choose heuristic:
   Manhattan - Total tile distance to goal
   Misplaced tiles - count of incorrect tiles
3) Start with initial board, moves = 0
4) Use a priority queue to store state by:
   priority = moves + heuristics
5) Track visited boards in a set
   While queue not empty:
   Remove state with lowest priority
   If it is the goal, return the path
   Add state to visited
   Add all valid neighbour states to queue (if not visited).
6) If no solution, return "No solution."

```
def manhattan_distance(self):
    dis = 0
    for idx, val in enumerate(self.board):
        if val == 0:
            continue
        goal_x, goal_y = divmod(val-1, 3)
        cur_x, cur_y = divmod(idx, 3)
        distance += abs(goal_x - cur_x) + abs(goal_y:
    return distance
```

```
def
function get misplacedTiles(self):
    return sum(1 for i, val in enumerate(self.board)
    if val != 0 and val != i+1)

def heuristic_cost(self):
    if self.heuristic == 'manhattan':
        return self.manhattan_distance()
    elif self.heuristic == 'misplaced':
        return self.misplaced_tiles()
    else:
        return 0

def solve_puzzle(start_board, heuristic='manhattan'):
    start_state = PuzzleState(start_board, heuristic=heuristic
    frontier = []
    heapq.heappush(frontier, start_state
    explored = set()
    while frontier:
        state = heapq.heappop(frontier)
        if state.is_goal():
            path = []
            while state:
                path.append(state.board)
                state = state.previous
            return path[::-1]
        explored.add(tuple(state.board))
        for neighbor in state.neighbors():
            if tuple(neighbor.board) not in explored:
                heapq.heappush(frontier, neighbor)
    return None
```

Output

(1, 2, 3) $g(n) = 0$

(4, 5, 6) $h(n) = 2$

(0, 7, 8) $f = g + h = 2$

(1, 2, 3) → (1, 2, 3) $g(n) = 2$

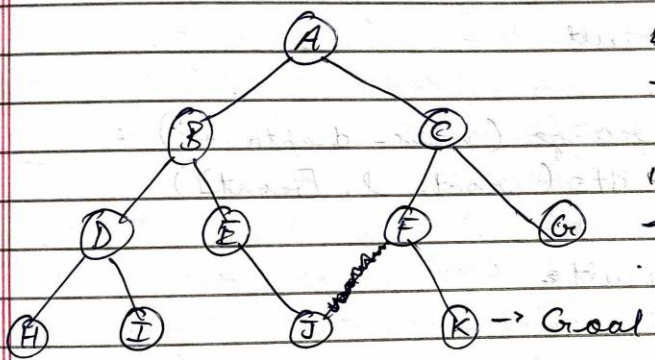
(4, 5, 6) → (4, 5, 6) $h(n) = 0$

(7 0 8) (7, 8, 0) $f = 2$

$g(n) = 1$

$h(n) = 1$

IDDFS $f = g + h = 2$



~~1 2 3~~

~~4 5 6~~

~~- 7 8~~

~~1 2 3~~

~~5 6~~

1	2	3
4	5	6
-	7	8

A

A B C

A B D E C F G

A B D H I J C F G K

A → ~~C~~ → F → K

1	2	3
-	5	6
4	7	8

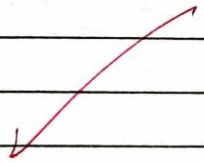
1	2	3
4	5	6
7	-	8

1	2	3
4	-	6
7	5	8

1	2	3
4	5	6
7	8	-

IDDFS Pseudocode

```
def IDDFS (root, goal, graph, max_depth):
    def dfs (node, depth, path):
        if node == goal:
            return path
        if depth == 0
            return None
    for child in graph.get(node, []):
    result = dfs(child, depth-1, path + [child])
    if result:
        return result
    return None
    for l in range (max_depth + 1):
        result = dfs (root, l, [root])
    if result:
        return result
    return None
```



Code:

8 Puzzle using DFS:

```
# -------- 8 Puzzle DFS Solver (Corrected Version) --------

# Goal configuration (0 = blank)
goal = "123456780"

# Valid swap positions for each tile index
neighbors = {
   0:[1,3], 1:[0,2,4], 2:[1,5],
   3:[0,4,6], 4:[1,3,5,7], 5:[2,4,8],
   6:[3,7], 7:[4,6,8], 8:[5,7]
}

# Print board in puzzle format (3x3)
def printPuzzle(state):
   for i in range(0, 9, 3):
      row = state[i:i+3].replace("0", " ")
      print(" ".join(row))
   print()

# DFS Search
def dfs(start):
   stack = [(start, [start])]  # store full path
   visited = set()

   while stack:
      state, path = stack.pop()

      if state == goal:
         return path

      visited.add(state)

      zero_pos = state.index("0")

      for n in neighbors[zero_pos]:
         lst = list(state)
         lst[zero_pos], lst[n] = lst[n], lst[zero_pos]
         new_state = "".join(lst)
         if new_state not in visited:
```

```
            stack.append((new_state, path + [new_state]))

    return None

# -------- CHANGE START STATE HERE --------
start = "283164705"

# -----------------------------------------

solution = dfs(start)

if solution:
    print("☑ Solution found in", len(solution)-1, "moves!\n")

    for i, step in enumerate(solution):
        print("Step", i)
        printPuzzle(step)

else:
    print("❌ No solution found.")
```

Output:

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

Step 1
1 2 3
4   6
7 5 8

Step 2
1 2 3
4 5 6
7   8

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

Code for IDDFS:

```
import time
from collections import deque
import heapq

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

def manhattan_distance(state):
    distance = 0
    for i in range(9):
        if state[i] != 0:
            goal_x, goal_y = (state[i] - 1) // 3, (state[i] - 1) % 3
            current_x, current_y = i // 3, i % 3
            distance += abs(goal_x - current_x) + abs(goal_y - current_y)
    return distance

def is_goal(state):
    return state == GOAL_STATE

def get_neighbors(state):
    neighbors = []
    zero_pos = state.index(0)
    x, y = zero_pos // 3, zero_pos % 3
    directions = [(-1, 0), (1, 0), (0, -1), (0, 1)]

    for dx, dy in directions:
        new_x, new_y = x + dx, y + dy
        if 0 <= new_x < 3 and 0 <= new_y < 3:
            new_pos = new_x * 3 + new_y
            new_state = list(state)
            new_state[zero_pos], new_state[new_pos] = new_state[new_pos], new_state[zero_pos]
            neighbors.append(tuple(new_state))

    return neighbors

def dfs(start_state):
    stack = [(start_state, [])]
    visited = set()
    visited.add(start_state)
```

```
    while stack:
        current_state, path = stack.pop()

        if is_goal(current_state):
            return path + [current_state]

        for neighbor in get_neighbors(current_state):
            if neighbor not in visited:
                visited.add(neighbor)
                stack.append((neighbor, path + [current_state]))
    return None

def best_first_search(start_state):
    open_list = []
    heapq.heappush(open_list, (manhattan_distance(start_state), start_state, []))
    visited = set()
    visited.add(start_state)

    while open_list:
        _, current_state, path = heapq.heappop(open_list)
        if is_goal(current_state):
            return path + [current_state]

        for neighbor in get_neighbors(current_state):
            if neighbor not in visited:
                visited.add(neighbor)
                heapq.heappush(open_list, (manhattan_distance(neighbor), neighbor, path +
[current_state]))
    return None

def iddfs(start_state):
    def dfs_with_depth_limit(state, depth, path):
        if depth == 0:
            return None
        if is_goal(state):
            return path + [state]
        for neighbor in get_neighbors(state):
            result = dfs_with_depth_limit(neighbor, depth - 1, path + [state])
            if result:
                return result
```

```
        return None

    depth = 0
    while True:
        result = dfs_with_depth_limit(start_state, depth, [])
        if result:
            return result
        depth += 1

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

print("DFS Solution:")
start_time = time.time()
dfs_solution = dfs(start_state)
end_time = time.time()

if dfs_solution:
    for step in dfs_solution:
        print(step)
else:
    print("No solution found using DFS.")
print(f"Time taken for DFS: {end_time - start_time:.6f} seconds")

print("\nBest-First Search Solution:")
start_time = time.time()
best_first_solution = best_first_search(start_state)
end_time = time.time()
if best_first_solution:
    for step in best_first_solution:
        print(step)
else:
    print("No solution found using Best-First Search.")
print(f"Time taken for Best-First Search: {end_time - start_time:.6f} seconds")

print("\nIDDFS Solution:")
start_time = time.time()
iddfs_solution = iddfs(start_state)
end_time = time.time()
if iddfs_solution:
    for step in iddfs_solution:
```

```
        print(step)
else:
    print("No solution found using IDDFS.")
print(f"Time taken for IDDFS: {end_time - start_time:.6f} seconds")
```

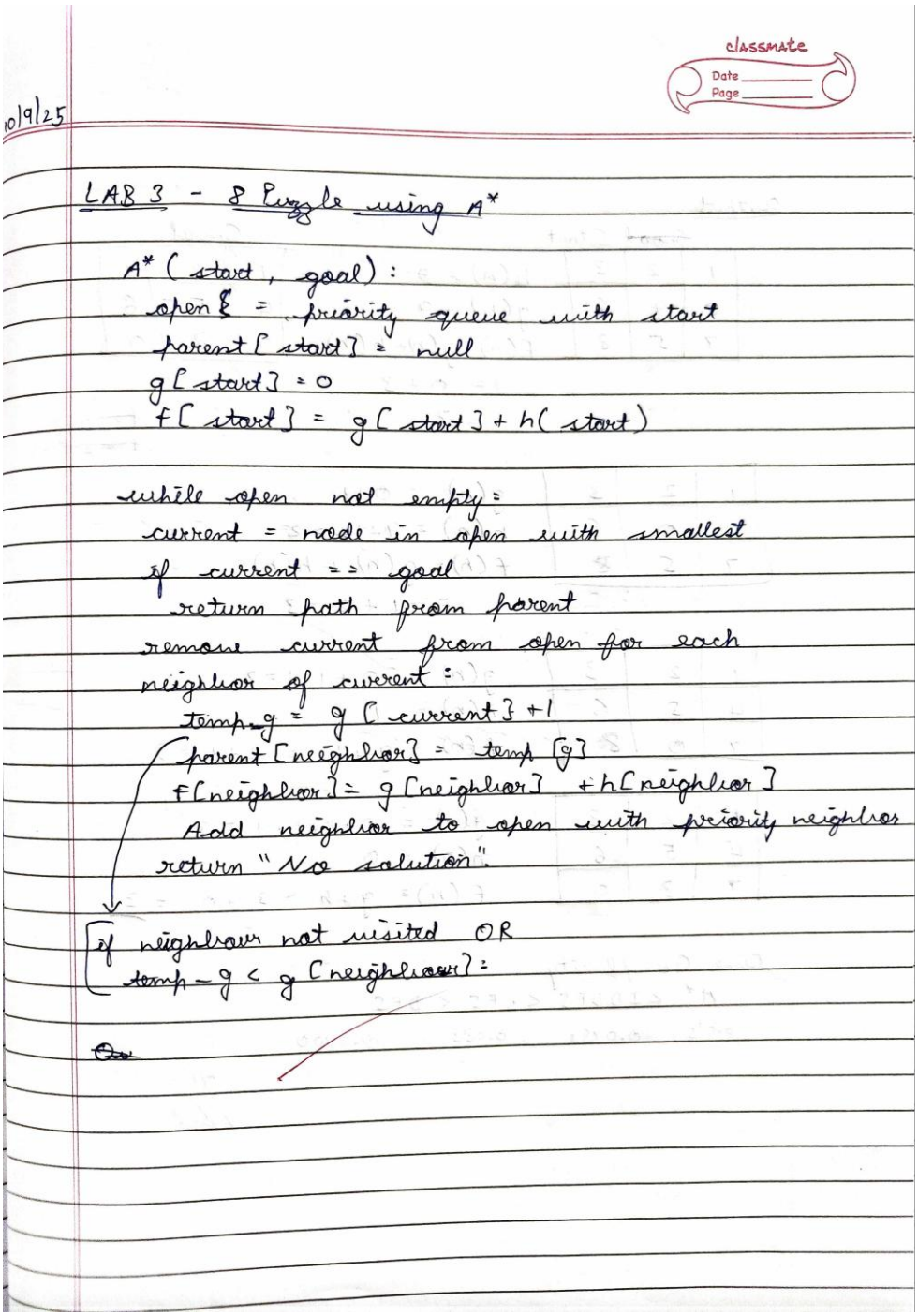
Output:

```
DFS Solution:
(5, 6, 3, 1, 2, 0, 4, 7, 8)
(5, 6, 3, 1, 0, 2, 4, 7, 8)
(5, 0, 3, 1, 6, 2, 4, 7, 8)
(0, 5, 3, 1, 6, 2, 4, 7, 8)
(1, 5, 3, 0, 6, 2, 4, 7, 8)
...
(1, 2, 3, 4, 5, 6, 7, 8, 0)
Time taken for DFS: 0.154329 seconds

Best-First Search Solution:
(5, 6, 3, 1, 2, 0, 4, 7, 8)
(5, 6, 3, 1, 2, 8, 4, 7, 0)
(5, 6, 3, 1, 2, 8, 4, 0, 7)
(5, 6, 3, 1, 0, 8, 4, 2, 7)
...
(1, 2, 3, 4, 5, 6, 7, 8, 0)
Time taken for Best-First Search: 0.003812 seconds

IDDFS Solution:
Depth limit = 0
Depth limit = 1
Depth limit = 2
Depth limit = 3
Depth limit = 4
...
(5, 6, 3, 1, 2, 0, 4, 7, 8)
(5, 6, 3, 1, 0, 2, 4, 7, 8)
(5, 0, 3, 1, 6, 2, 4, 7, 8)
...
(1, 2, 3, 4, 5, 6, 7, 8, 0)
Time taken for IDDFS: 2.918440 seconds
```

Program 3

Implement A* search algorithm

Algorithm:



10/9/25

LAB 3 - 8 Puzzle using A*

A* (start, goal):
open = priority queue with start
parent[start] = null
g[start] = 0
f[start] = g[start] + h(start)

while open not empty:
current = node in open with smallest
if current == goal
return path from parent
remove current from open for each
neighbor of current:
temp_g = g[current] + 1
parent[neighbor] = temp[g]
f[neighbor] = g[neighbor] + h[neighbor]
Add neighbor to open with priority neighbor
return "No solution".

if neighbour not visited OR
temp_g < g[neighbour]:

Output

~~Goal~~ Start

1	2	3
0	4	6
7	5	8

$h(n) = 1+1+1 = 3$

$g(n) = 0$

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

$= 0 + 3$

$= 3$

Goal

1	2	3
4	5	6
7	8	0

~~$f = h - g$~~

~~$f = 3 - 1 = 2$~~

1	2	3
4	0	6
7	5	8

$g(n) = 0 + 1$

$h(n) = 1 + 1 = 2$

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

$= 1 + 2 = 3$

1	2	3
4	5	6
7	0	8

$g(n) = 0 + 1 + 1 = 2$

$h(n) = 1$

$f(n) = g + h = 2 + 1 = 3$

~~$3 - 2 = 1$~~

~~$3 - 3 = 0$~~

1	2	3
4	5	6
7	8	0

$g(n) = 0 + 1 + 1 + 1 = 3$

$h(n) = 0$

$f(n) = g + h = 3 + 0 = 3$

Time Complexity

$A^* < IDDFS < BFS < DFS$

0.015 0.0755 0.083 0.2100

Code:

```
import heapq

GOAL = (1, 2, 3,
        4, 5, 6,
        7, 8, 0)   # Goal state (0 = blank)

# Manhattan Distance Heuristic
def manhattan(state):
    dist = 0
    for i, val in enumerate(state):
        if val == 0:
            continue
        goal_idx = val - 1
        r, c = divmod(i, 3)
        gr, gc = divmod(goal_idx, 3)
        dist += abs(r - gr) + abs(c - gc)
    return dist

# Generate neighbors
def neighbors(state):
    res = []
    i = state.index(0)
    r, c = divmod(i, 3)
    moves = []
    if r > 0: moves.append((-3, "Up"))
    if r < 2: moves.append((3, "Down"))
    if c > 0: moves.append((-1, "Left"))
    if c < 2: moves.append((1, "Right"))
    for d, move in moves:
        new = list(state)
        new[i], new[i+d] = new[i+d], new[i]
        res.append((tuple(new), move))
    return res

# A* Search Algorithm
def astar(start):
    open_list = []
    heapq.heappush(open_list, (manhattan(start), 0, start, []))
    visited = set()
```

```
    while open_list:
        f, g, state, path = heapq.heappop(open_list)
        if state in visited:
            continue
        visited.add(state)

        if state == GOAL:
            return path

        for neigh, move in neighbors(state):
            if neigh not in visited:
                heapq.heappush(open_list,
                               (g+1+manhattan(neigh), g+1, neigh, path+[move]))
    return None

# Pretty print the board
def print_state(state):
    for i in range(0, 9, 3):
        row = state[i:i+3]
        print(" ".join(str(x) if x != 0 else " " for x in row))
    print()

# -------- MAIN --------
if __name__ == "__main__":
    # ☑ Start state exactly 3 moves away
    initial = (1, 2, 3,
               4, 5, 6,
               7, 0, 8)

    print("Initial State:")
    print_state(initial)

    solution = astar(initial)

    if solution:
        print("Solution found in", len(solution), "moves.\n")

        # Print steps as puzzle boards
        state = initial
        print("Step 0:")
        print_state(state)
```

```
    for i, move in enumerate(solution, 1):
        idx = state.index(0)
        for neigh, mv in neighbors(state):
            if mv == move:
                state = neigh
                break
        print("Step", i, "-", move)
        print_state(state)

    print("Moves:", " -> ".join(solution))
else:
    print("No solution exists.")
```

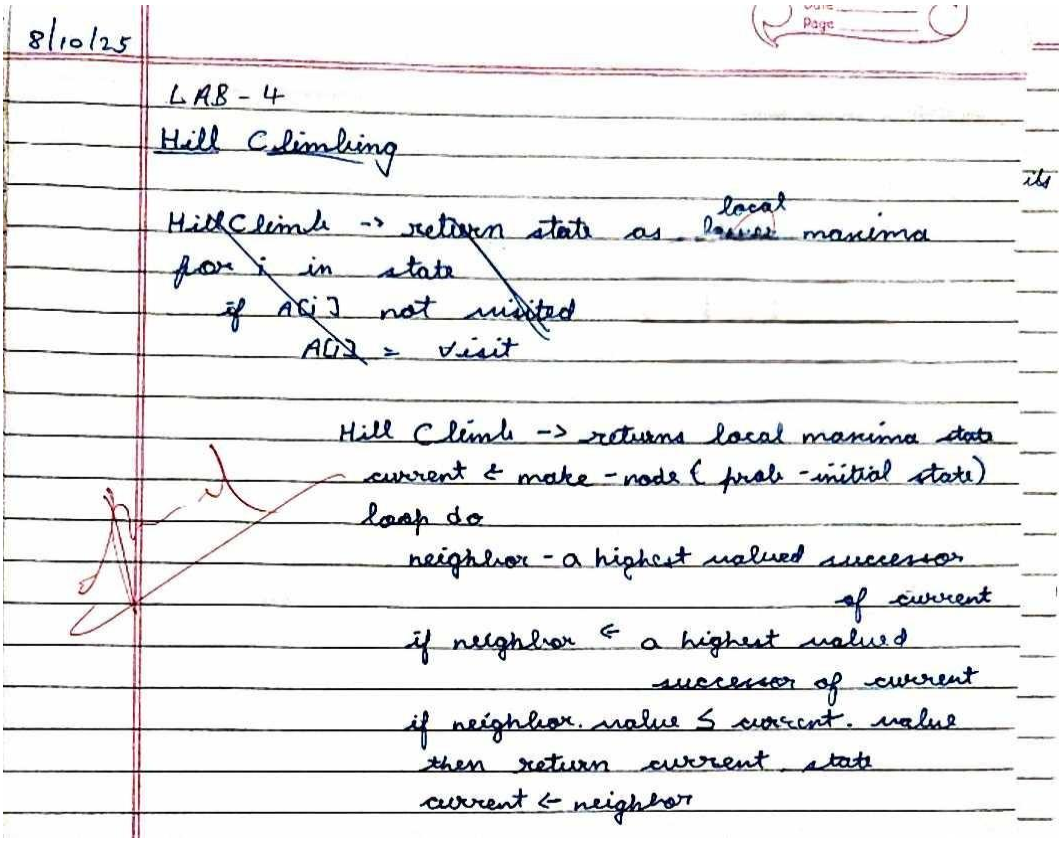
Output:

```
Initial State:
1 2 3
4 5 6
7   8

Solution found in 1 moves.

Step 0:
1 2 3
4 5 6
7   8

Step 1 - Right
1 2 3
4 5 6
7 8
```

Program 4

Implement hill climb search algorithm to solve N Queens problem

Algorithm:



8/10/25

LAB - 4

Hill Climbing

HillClimb -> return state as local maxima

for i in state

if A[i] not visited

A[i] = visit

HillClimb -> returns local maxima state

current ← make-node (prob-initial state)

loop do

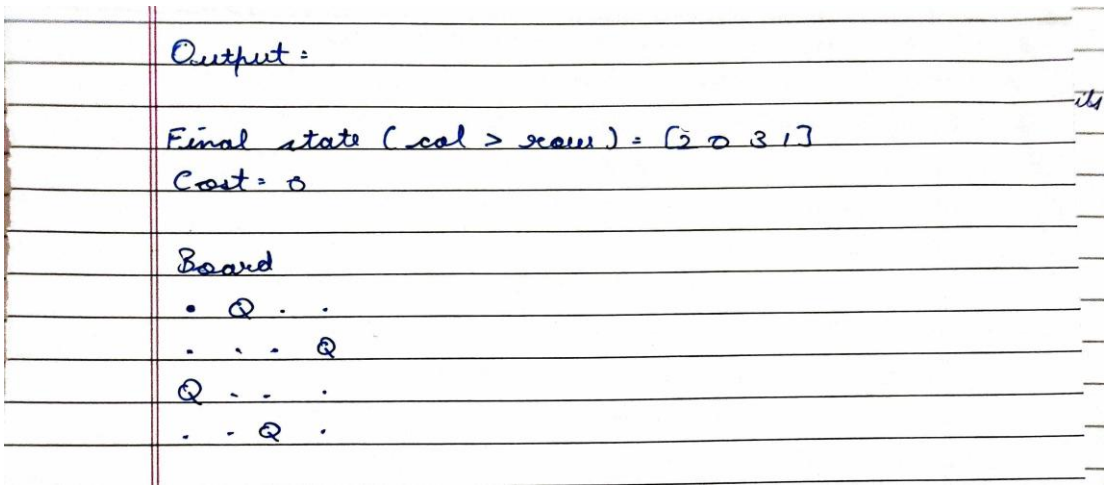
neighbor - a highest valued successor of current

if neighbor ← a highest valued successor of current

if neighbor.value ≤ current.value

then return current.state

current ← neighbor



Output :

Final state (col > rows) : [2 0 3 1]

Cost : 0

Board

. Q . .

. . . Q

Q . . .

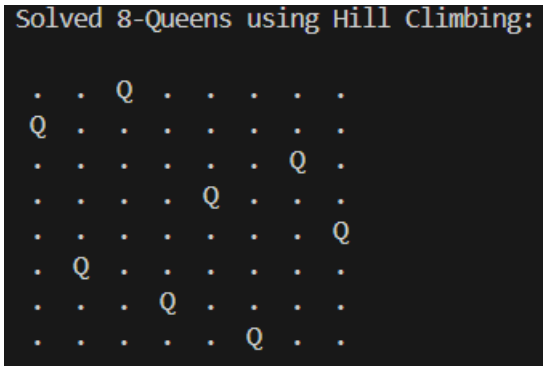
. . Q .

Code:

```
import random

# Generate a random board (one queen per row)
def random_board(n):
    return [random.randint(0, n-1) for _ in range(n)]

# Count number of attacking queen pairs
def heuristic(board):
    h = 0
    n = len(board)
    for i in range(n):
        for j in range(i+1, n):
            if board[i] == board[j] or abs(board[i]-board[j]) == abs(i-j):
                h += 1
    return h

# Get best neighbor state
def best_neighbor(board):
    n = len(board)
    best = board[:]
    best_h = heuristic(board)

    for row in range(n):
        for col in range(n):
            if board[row] != col:
                new_board = board[:]
                new_board[row] = col
                h = heuristic(new_board)
                if h < best_h:
                    best = new_board
                    best_h = h
    return best, best_h

# Hill climbing with random restarts
def hill_climbing(n):
    current = random_board(n)
    current_h = heuristic(current)

    while current_h != 0:
        neighbor, neighbor_h = best_neighbor(current)
```

```
        if neighbor_h >= current_h:  # local maxima / plateau
            current = random_board(n)   # restart
            current_h = heuristic(current)
        else:
            current = neighbor
            current_h = neighbor_h

    return current

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

# -------- MAIN ----------
if __name__ == "__main__":
    N = 8   # change this for any N
    solution = hill_climbing(N)

    print(f"Solved {N}-Queens using Hill Climbing:\n")
    print_board(solution)
```

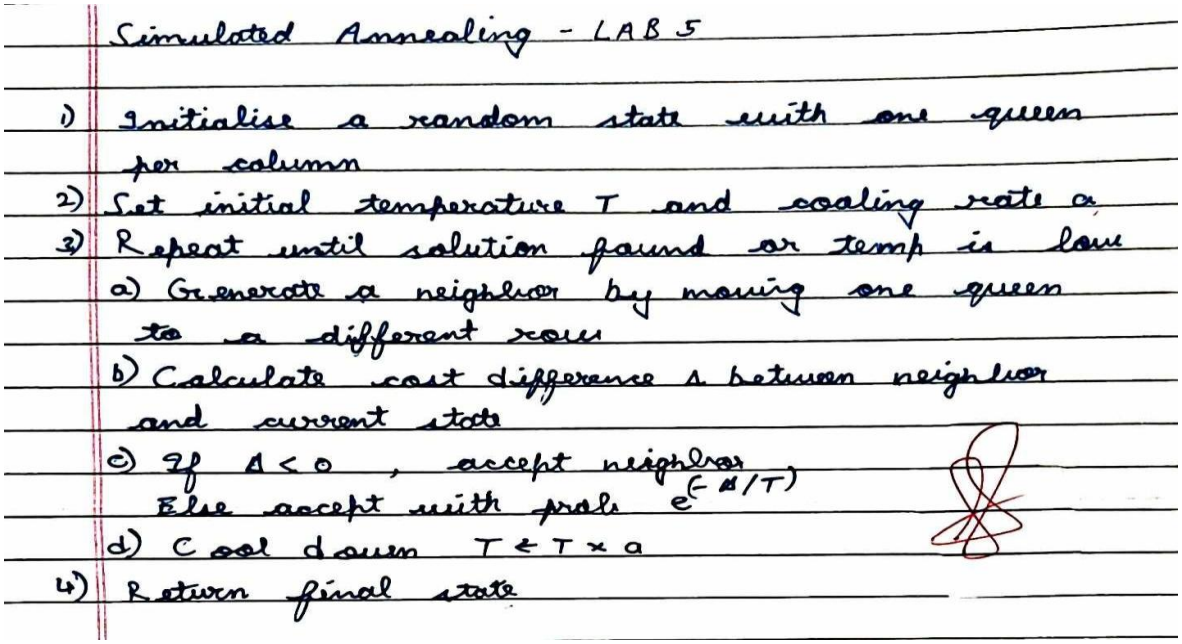
Output:

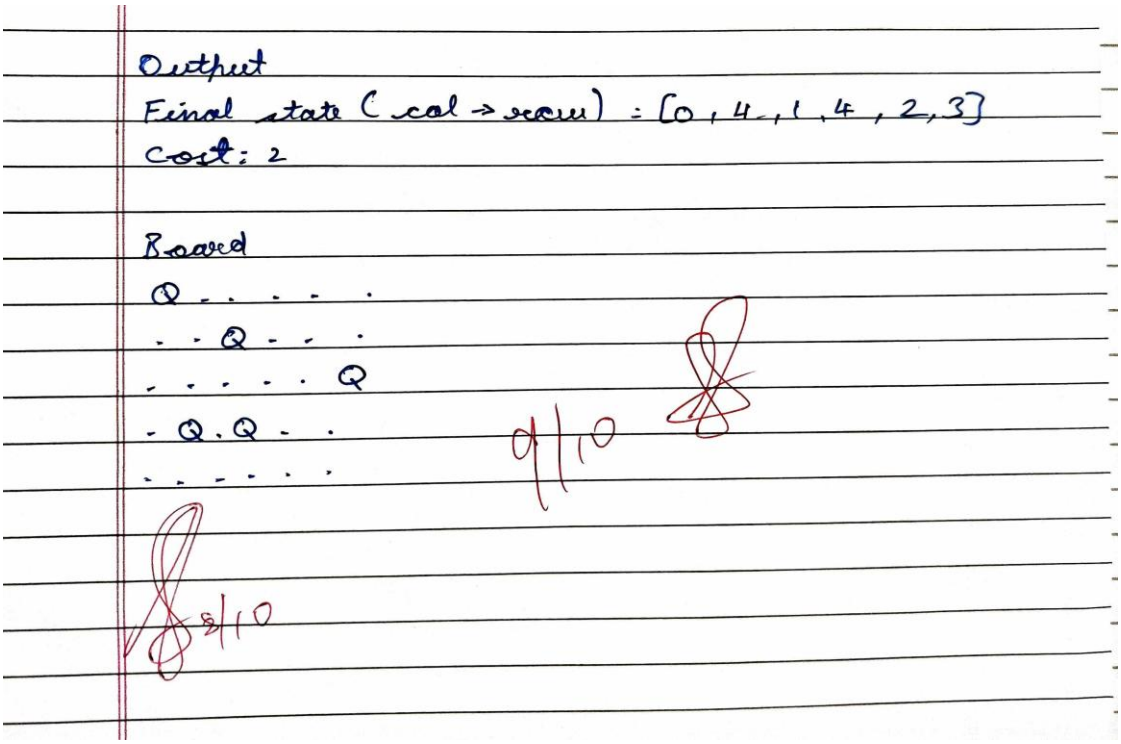


Program 5

Simulated Annealing to Solve 8-Queens problem

Algorithm:





Code:

```
import random
import math

# Number of queens
N = 4

def cost(state):
    """Compute number of attacking queen pairs."""
    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 random_neighbor(state):
    """Generate a neighbor by moving one queen to another row."""
    neighbor = state.copy()
    col = random.randrange(N)
    new_row = random.randrange(N-1)
    if new_row >= neighbor[col]:
        new_row += 1
    neighbor[col] = new_row
    return neighbor

def simulated_annealing(
    T0=5.0, alpha=0.995, Tmin=1e-6, max_iters=50000
):
    # Start with a random placement of queens
    state = [random.randrange(N) for _ in range(N)]
    current_cost = cost(state)
    T = T0
    it = 0

    while T > Tmin and it < max_iters and current_cost != 0:
        neighbor = random_neighbor(state)
        neighbor_cost = cost(neighbor)
        delta = neighbor_cost - current_cost

        if delta <= 0 or random.random() < math.exp(-delta / T):
```

```
            state, current_cost = neighbor, neighbor_cost

        T *= alpha
        it += 1

    return state, current_cost

def print_board(state):
    """Pretty-print the board."""
    for r in range(N):
        row = ""
        for c in range(N):
            row += "Q " if state[c] == r else ". "
        print(row)
    print()

# Run SA for 4-queens
solution, c = simulated_annealing()

print("Final state (col -> row):", solution)
print("Cost:", c)
print("\nBoard:")
print_board(solution)
```

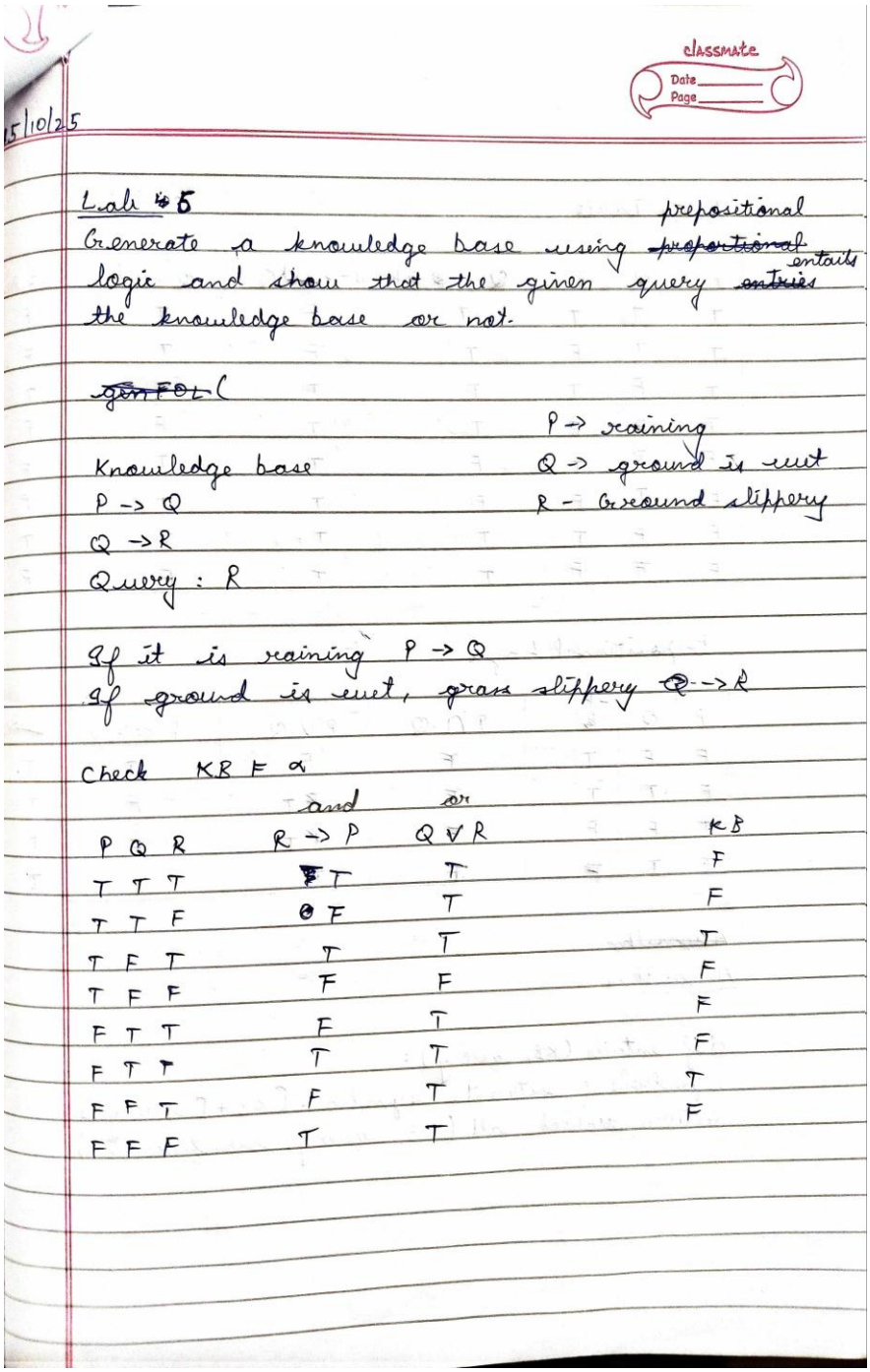
Output:

```
Final state (col -> row): [1, 3, 0, 2]
Cost: 0

Board:
. . Q .
Q . . .
. . . Q
. Q . .
```

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

Lab #5

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

~~gen FOL~~ (

P -> raining
Q -> ground is wet
R - Ground slippery

Knowledge base
P -> Q
Q -> R
Query : R

If it is raining P -> Q
If ground is wet, grass slippery Q -> R

Check KB ⊨ α

P	Q	R	R -> P (and)	Q ∨ R (or)	KB
T	T	T	T	T	T
T	T	F	F	T	F
T	F	T	T	T	T
T	F	F	F	F	F
F	T	T	F	T	F
F	T	F	T	T	F
F	F	T	F	T	F
F	F	F	T	F	F

Truth Table

P	Q	R	Q -> P	P -> not Q	Q or R	KB
T	T	T	T	F	T	F
T	T	F	T	F	T	F
T	F	T	T	T	T	T
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
F	F	F	T	T	F	F

Propositional Logic

P	Q	~P	P ∩ Q (and)	P ∨ Q (or)	P <=> Q	PL
F	F	T	F	F	T	T
F	T	T	F	T	F	T
T	F	F	F	T	F	F
T	T	F	T	T	T	T

~~Algorithm~~

Algorithm

```
def entails(KB, query):
    symbols = extract_symbols(KB + [query])
    return tt_check_all(KB, query, symbols, {})
```

```
def tt_check_all(KB, query, symbols, model)
if not symbols:
    if all(eval_formula(s, model) for s in KB)
        return eval_formula(query, model)
    else:
        return true
else:
    P = symbols[0]

rest = symbols[1:]
return (tt_check_all(KB, query, rest{ model, P: True}) and
tt_check_all(KB, query, rest{ ** model, P: False})
```

(i) KB ⊨ R
(ii) KB ⊭ (R → P)
(iii) KB ⊨ (Q → R)

15/10

(i) If KB is T, R is T => entails

(ii) KB does not entail R → P because for KB is T, R → P is false F

(iii) KB entails Q → R because for every true of KB, Q → R is true

Output

The knowledge base entails the query
'(A V C)' ∧ '(B V ~C)' |= 'A V B'

Symbols are ['A', 'B', 'C']

Knowledge base (KB) = (A V C) ∧ (B V ~C)
Query (α) = A V B

A	B	C	A V C	B V ~C	KB	α
F	F	F	F	T	F	F
F	F	T	T	F	F	F
F	T	F	F	T	F	T
F	T	T	T	T	T	T
T	F	F	T	T	T	T
T	F	T	T	F	F	T
T	T	F	T	T	T	T
T	T	T	T	T	T	T

Conclusion:
Entails the given condition

15/10

Code:

```
import re
from itertools import product

def pl_true(sentence, model):
    try:
        return eval(sentence, model)
    except NameError:
        return False

def tt_entails(kb, alpha):
    kb = kb.replace('¬', 'not ').replace('∧', ' and ').replace('∨', ' or ')
    alpha = alpha.replace('¬', 'not ').replace('∧', ' and ').replace('∨', ' or ')
    symbols = sorted(list(set(re.findall(r'[A-Z]', kb + alpha))))

    print(f"Symbols found: {symbols}")
    for values in product([True, False], repeat=len(symbols)):
        model = dict(zip(symbols, values))
        if pl_true(kb, model):
            if not pl_true(alpha, model):
                print(f"Counterexample found: {model}")
                return False
    return True
if __name__ == "__main__":
    kb_formula = "(A ∨ C) ∧ (B ∨ ¬C)"
    alpha_formula = "A ∨ B"
    print(f"Knowledge Base (KB): {kb_formula}")
    print(f"Query (α): {alpha_formula}\n")
    result = tt_entails(kb_formula, alpha_formula)
    print("\n------ RESULT ------")
    if result:
        print(f"The Knowledge Base entails the Query.")
        print(f"   '{kb_formula}' |= '{alpha_formula}'")
    else:
        print(f"The Knowledge Base does NOT entail the Query.")
        print(f"   '{kb_formula}' |/= '{alpha_formula}'")
```

Output:

Knowledge Base (KB): (A ∨ C) ∧ (B ∨ ¬C)
Query (α): A ∨ B

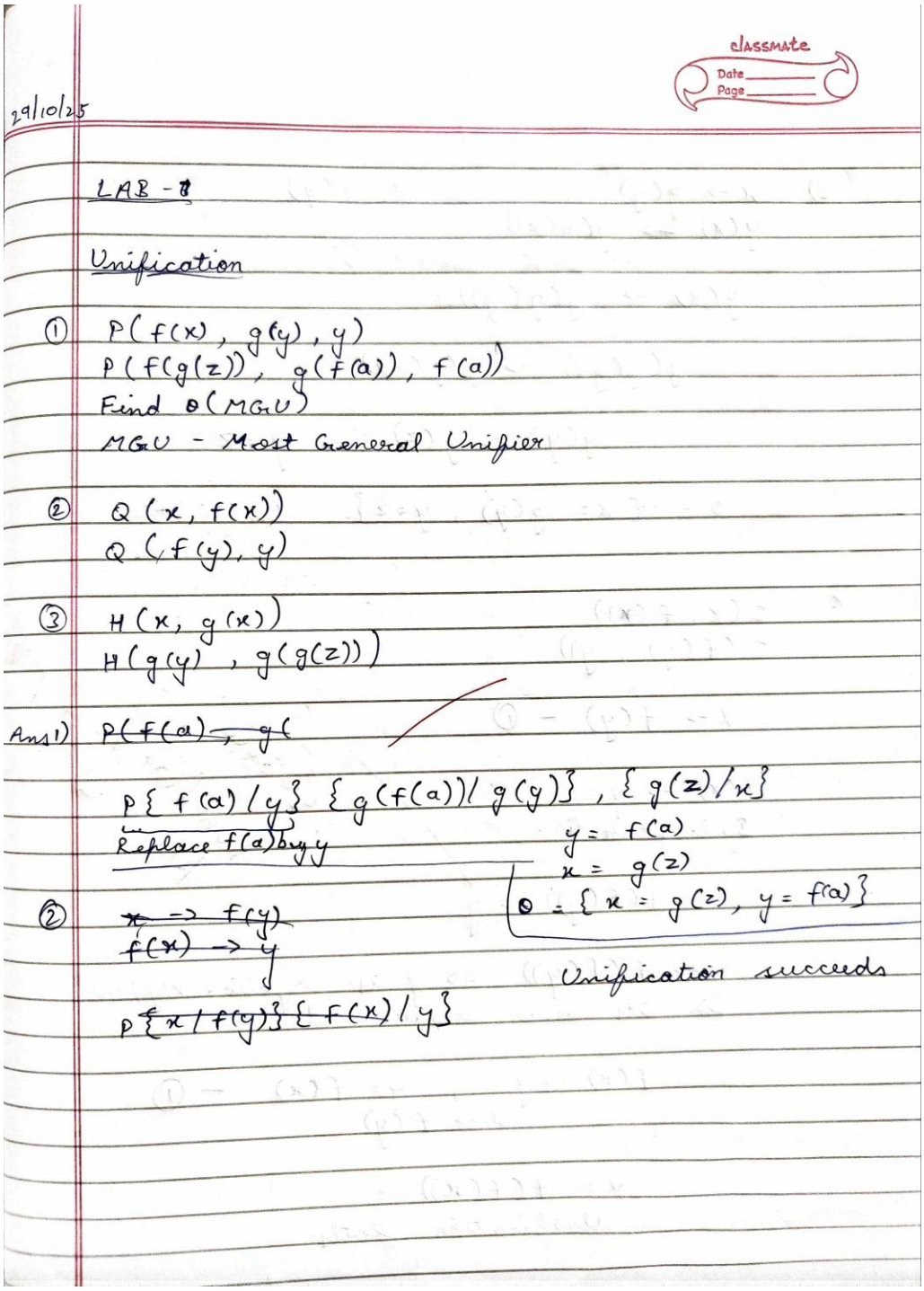
Symbols found: ['A', 'B', 'C']

------ RESULT ------
The Knowledge Base entails the Query.
'(A ∨ C) ∧ (B ∨ ¬C)' |= 'A ∨ B'

Program 7

Implement unification in first order logic

Algorithm:



3) $x \to g(y)$ $\quad x = g(y)$

$g(x) \Rightarrow g(g(z))$

$g(x) \to g(g(y))$

$g(g(y)) \to g(g(z))$

$g(y) \to g(z) \to y = z$

$\theta = \{x = g(y), y = z\}$ Unification succeeds

② $Q(x, f(x))$
$Q(f(y), y)$

$x \to f(y)$ - ①

$f(x) \to y$
Substitute ①

$f(f(y)) \to y$

literate
Min Max
AB

$y \to f(f(y))$ => y is cyclic. Appears in its own replacement

$f(x) \to y$, $y = f(x)$ - ①
$x \to f(y)$

$x = f(f(x))$
Unification fails

Unification Algorithm

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

First argument - x and b, x is variable, b is constant.
Substitute $\{x/b\}$. Apply it to $P(b, y, a)$ and $P(b, z, a)$

Second argument - y and z. Both are variables
Substitute $\{y/z\}$
Apply it to $P(b, z, a)$ and $P(b, z, a)$

Third argument - a and a. Both are identical constants

Unification succeeds
$\{x/b, y/z\}$

~~Examples~~

~~Output~~

Output
Unification Succeeds

Code:

```
# ---------- UNIFICATION IN FIRST ORDER LOGIC ----------

# Check if a symbol is a variable
def is_variable(x):
    return isinstance(x, str) and x.startswith("?")

# Check if term is compound (e.g., ("f", "a", "?x"))
def is_compound(x):
    return isinstance(x, tuple) and len(x) > 1

# Occur check (prevents infinite recursive substitutions)
def occur_check(var, term, subst):
    if var == term:
        return True
    if is_variable(term) and term in subst:
        return occur_check(var, subst[term], subst)
    if is_compound(term):
        return any(occur_check(var, arg, subst) for arg in term[1:])
    return False

# Apply substitution to a term
def substitute(term, subst):
    if is_variable(term):
        while is_variable(term) and term in subst:
            term = subst[term]
        return term
    if is_compound(term):
        return (term[0],) + tuple(substitute(a, subst) for a in term[1:])
    return term

# Unification algorithm
def unify(x, y, subst=None):
    if subst is None:
        subst = {}

    x = substitute(x, subst)
    y = substitute(y, subst)
```

```
    if x == y:
        return subst
    if is_variable(x):
        if occur_check(x, y, subst):
            return None
        subst[x] = y
        return subst
    if is_variable(y):
        if occur_check(y, x, subst):
            return None
        subst[y] = x
        return subst
    if is_compound(x) and is_compound(y) and x[0] == y[0] and len(x) == len(y):
        for a, b in zip(x[1:], y[1:]):
            subst = unify(a, b, subst)
            if subst is None:
                return None
        return subst
    return None

# ---------- SAMPLE TEST CASES ----------
tests = [
    (("P", "?x"), ("P", "a")),
    (("P", "?x", "b"), ("P", "a", "?y")),
    (("f", "?x", ("g", "?x")), ("f", "a", "?y")),
    (("f", "?x"), ("f", ("g", "?x")))
]

print("\n=== SAMPLE UNIFICATION OUTPUT ===\n")
for t1, t2 in tests:
    result = unify(t1, t2, {})
    print(f"{t1} ↔ {t2} => Substitution: {result}")
```

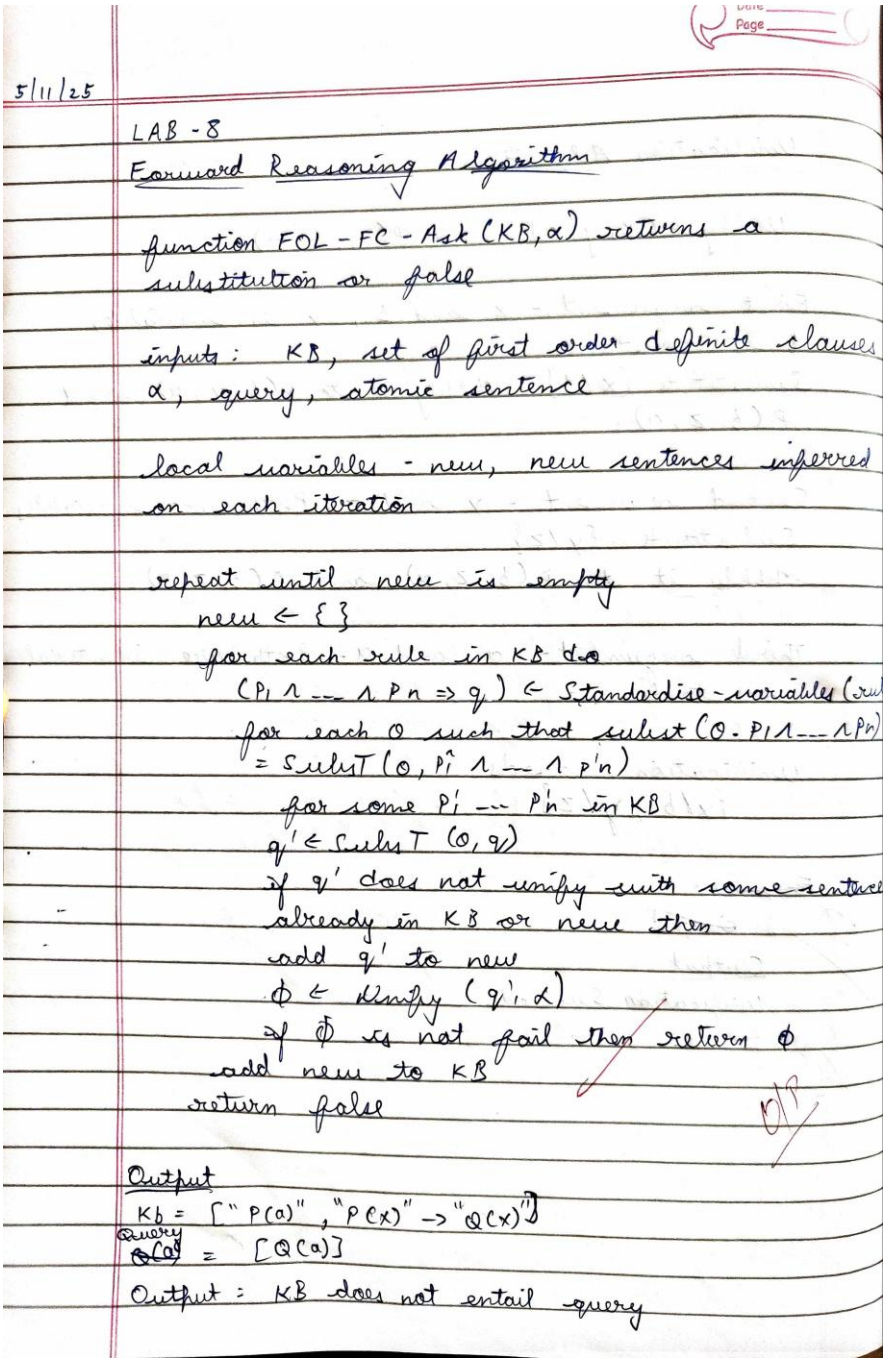
Output:

```
('P', '?x') ↔ ('P', 'a') => Substitution: {'?x': 'a'}
('P', '?x', 'b') ↔ ('P', 'a', '?y') => Substitution: {'?x': 'a', '?y': 'b'}
('f', '?x', ('g', '?x')) ↔ ('f', 'a', '?y') => Substitution: {'?x': 'a', '?y': ('g', 'a')}
('f', '?x') ↔ ('f', ('g', '?x')) => Substitution: None
```

Program 8

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

Algorithm:



5/11/25

LAB - 8

Forward Reasoning Algorithm

function FOL - FC - Ask (KB, α) returns a substitution or false

inputs : KB, set of first order definite clauses
α, query, atomic sentence

local variables - new, new sentences inferred on each iteration

repeat until new is empty
new ← { }
for each rule in KB do
(P1 ∧ ... ∧ Pn ⇒ q) ← Standardise - variables (rule)
for each θ such that subst (θ, P1 ∧ ... ∧ Pn) = Subst (θ, P'1 ∧ ... ∧ P'n)
for some P'1 ... P'n in KB
q' ← Subst (θ, q)
if q' does not unify with some sentence already in KB or new then
add q' to new
φ ← Unify (q', α)
if φ is not fail then return φ
add new to KB
return false

Output

KB = ["P(a)", "P(x)" -> "Q(x)"]
query = [Q(a)]
Output : KB does not entail query

Code:

```
# --------- FORWARD REASONING / FORWARD CHAINING ----------

# Knowledge Base: facts and rules
facts = [
    ("Human", "Socrates"),
    ("Human", "Plato"),
    ("Man", "Socrates")
]

# Rules represented as: (premises, conclusion)
rules = [
    ([("Human", "?x")], ("Mortal", "?x")),          # If Human(x) then Mortal(x)
    ([("Man", "?x")], ("Human", "?x")),             # If Man(x) then Human(x)
]

# Check if predicate matches with substitution
def match(pattern, fact):
    if pattern[0] != fact[0]:
        return None
    subst = {}
    for p_arg, f_arg in zip(pattern[1:], fact[1:]):
        if p_arg.startswith("?"):
            subst[p_arg] = f_arg
        elif p_arg != f_arg:
            return None
    return subst

# Apply substitution to a predicate
def substitute(pred, subst):
    return tuple(subst.get(arg, arg) for arg in pred)

# Forward chaining reasoning
def forward_reasoning(query):
    inferred = set(facts)
    added = True
    steps = []

    while added:
```

```
        added = False

        for premises, conclusion in rules:
            for fact in list(inferred):
                subst = match(premises[0], fact)
                if subst is not None:
                    new_fact = substitute(conclusion, subst)

                    if new_fact not in inferred:
                        inferred.add(new_fact)
                        steps.append((premises[0], new_fact))
                        if new_fact == query:
                            return True, steps

    return False, steps

# -------- MAIN TEST ---------

# Query to prove
query = ("Mortal", "Socrates")

proven, steps = forward_reasoning(query)

print("Knowledge Base Facts:")
for f in facts:
    print(" ", f)

print("\nReasoning Steps:")
for p, c in steps:
    print(f"From {p} infer {c}")

print("\nQuery:", query)

if proven:
    print("☑ RESULT: Query is proven using forward reasoning!")
else:
    print("❌ RESULT: Query cannot be proven.")
```

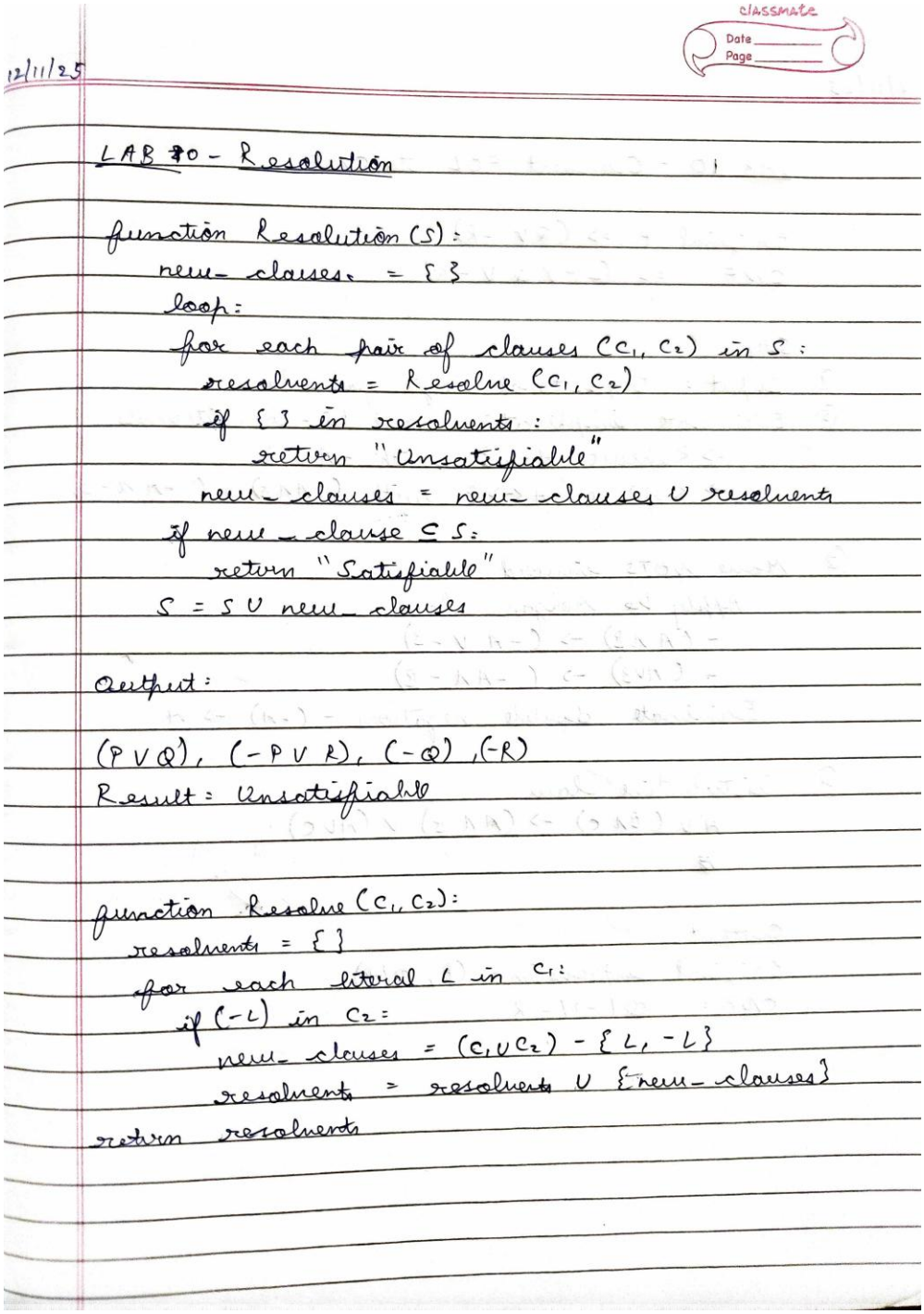
Output:

```
nowledge Base Facts:
  ('Human', 'Socrates')
  ('Human', 'Plato')
  ('Man', 'Socrates')

Reasoning Steps:
From ('Human', '?x') infer ('Mortal', 'Plato')
From ('Human', '?x') infer ('Mortal', 'Socrates')

Query: ('Mortal', 'Socrates')
RESULT: Query is proven using forward reasoning!
```

Program 9

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

Algorithm:



12/11/25

LAB 10 - Resolution

```
function Resolution(S):
    new_clauses = { }
    loop:
      for each pair of clauses (C1, C2) in S:
        resolvents = Resolve(C1, C2)
        if { } in resolvents:
          return "Unsatisfiable"
        new_clauses = new_clauses ∪ resolvents
      if new_clause ⊆ S:
        return "Satisfiable"
      S = S ∪ new_clauses
```

Output:

(P ∨ Q), (-P ∨ R), (-Q), (-R)

Result: Unsatisfiable

```
function Resolve(C1, C2):
    resolvents = { }
    for each literal L in C1:
      if (-L) in C2:
        new_clauses = (C1 ∪ C2) - {L, -L}
        resolvents = resolvents ∪ {new_clauses}
return resolvents
```

Code:

```
# ---------- UNIFICATION + RESOLUTION IN FOL ----------

# Check if a variable
def is_var(x):
    return isinstance(x, str) and x.startswith("?")

# Unify two literals
def unify(x, y, subst=None):
    if subst is None:
        subst = {}
    if x == y:
        return subst
    if is_var(x):
        subst[x] = y
        return subst
    if is_var(y):
        subst[y] = x
        return subst
    if isinstance(x, tuple) and isinstance(y, tuple) and x[0] == y[0]:
        for a, b in zip(x[1:], y[1:]):
            subst = unify(a, b, subst)
            if subst is None:
                return None
        return subst
    return None

# Apply substitution
def substitute(literal, subst):
    if not subst:
        return literal
    pred = literal[0]
    args = []
    for a in literal[1:]:
        args.append(subst.get(a, a))
    return (pred,) + tuple(args)

# Resolution step
def resolve(c1, c2):
```

```
    resolvents = []
    for l1 in c1:
        for l2 in c2:
            # Check complementary literals
            if l1[0] == "~" + l2[0] or l2[0] == "~" + l1[0]:
                subst = unify(l1[1:], l2[1:])
                if subst is not None:
                    new_clause = set(
                        substitute(l, subst) for l in (c1 | c2)
                        if l != l1 and l != l2
                    )
                    resolvents.append(frozenset(new_clause))
    return resolvents

# ---------- KNOWLEDGE BASE ----------

# CNF clauses represented as sets of literals
# Predicates stored as tuples: ("P", "a"), ("~P", "a")

KB = [
    frozenset({("~Human", "Socrates"), ("Mortal", "Socrates")}),
    frozenset({("~Man", "Socrates"), ("Human", "Socrates")}),
    frozenset({("Man", "Socrates")})
]

# Query
query = ("Mortal", "Socrates")

# Add negated query
KB.append(frozenset({("~Mortal", "Socrates")}))

# ---------- RESOLUTION PROCEDURE ----------

def resolution(KB):
    new = set()

    while True:
        pairs = [(KB[i], KB[j]) for i in range(len(KB)) for j in range(i+1, len(KB))]

        for (c1, c2) in pairs:
```

```
            resolvents = resolve(c1, c2)
            if frozenset() in resolvents:
                return True
            new.update(resolvents)

        if new.issubset(KB):
            return False

        for clause in new:
            if clause not in KB:
                KB.append(clause)

# ---------- MAIN ----------

print("\nKnowledge Base Clauses:")
for c in KB:
    print(" ", c)

result = resolution(KB)

print("\nQuery:", query)

if result:
    print("☑ RESULT: Query is proven by Resolution (empty clause derived)!")
else:
    print("❌ RESULT: Query cannot be proven.")
```

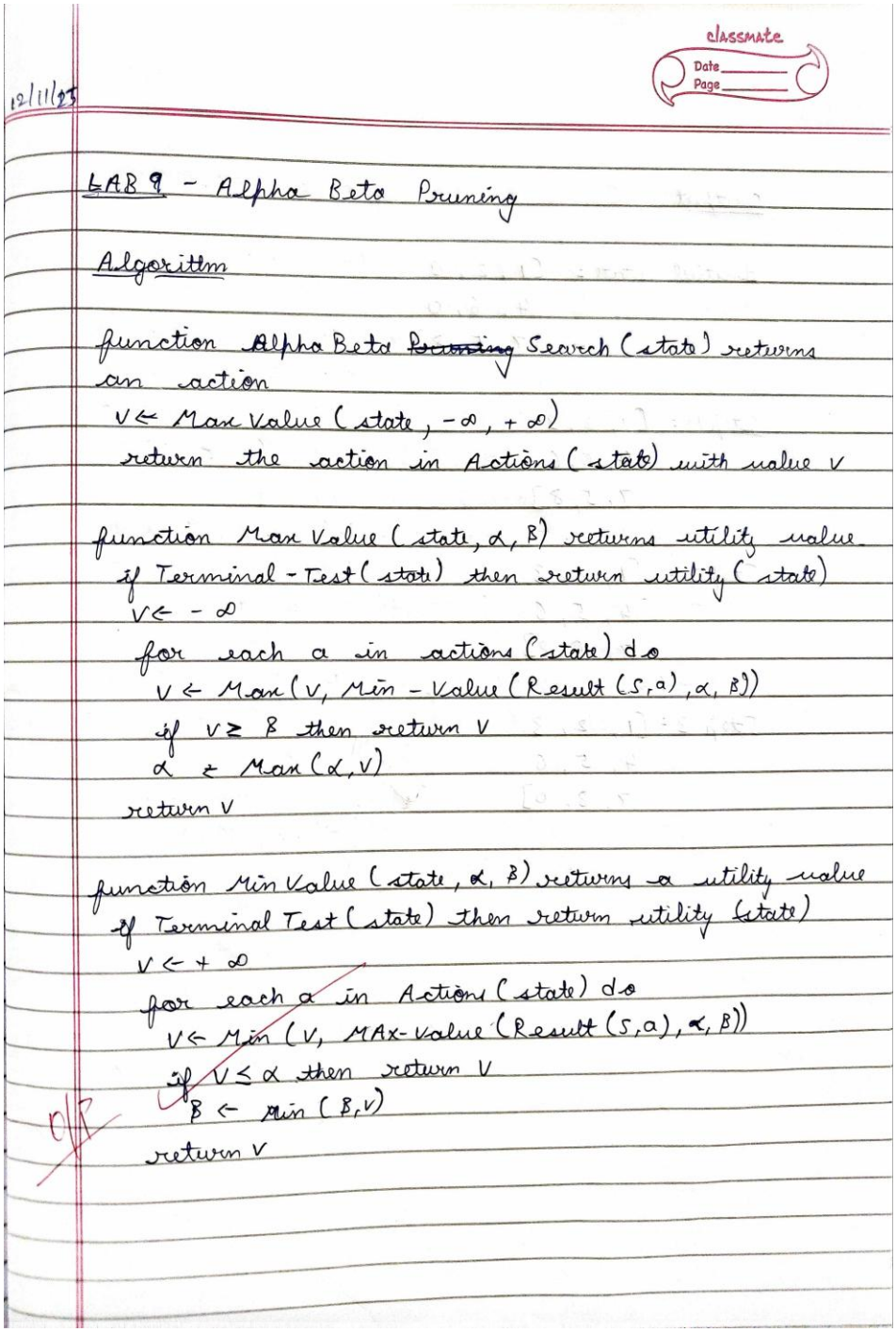
Output:

```
Knowledge Base Clauses:
  frozenset({('~Human', 'Socrates'), ('Mortal', 'Socrates')})
  frozenset({('Human', 'Socrates'), ('~Man', 'Socrates')})
  frozenset({('Man', 'Socrates')})
  frozenset({('~Mortal', 'Socrates')})

Query: ('Mortal', 'Socrates')
RESULT: Query is proven by Resolution (empty clause derived)!
```

Program 10

Implement Alpha Beta Pruning

Algorithm:



12/11/25

LAB 9 - Alpha Beta Pruning

Algorithm

function Alpha Beta ~~Pruning~~ 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 utility value
  if Terminal - Test (state) then return utility (state)
  v ← -∞
  for each a 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

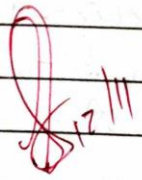
Output

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

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

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

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



Code:

```
import math
GOAL = (1,2,3,4,5,6,7,8,0)

# Manhattan distance heuristic
def manhattan(state):
    dist = 0
    for i,val in enumerate(state):
        if val == 0: continue
        goal = GOAL.index(val)
        r1,c1 = divmod(i,3)
        r2,c2 = divmod(goal,3)
        dist += abs(r1-r2) + abs(c1-c2)
    return dist

# Generate legal moves
def get_neighbors(state):
    moves = []
    zero = state.index(0)
    r,c = divmod(zero,3)
    dirs = []
    if r>0: dirs.append(-3)
    if r<2: dirs.append(3)
    if c>0: dirs.append(-1)
    if c<2: dirs.append(1)
    for d in dirs:
        new = list(state)
        new[zero], new[zero+d] = new[zero+d], new[zero]
        moves.append(tuple(new))
    return moves

# Pretty print puzzle board
def print_puzzle(state):
    for i in range(0,9,3):
        row = state[i:i+3]
        print(" ".join(str(x) if x!=0 else " " for x in row))
    print()
```

```
# Alpha Beta Minimax Search
def alphabeta(state, depth, alpha, beta, maximizing):
    print("State:")
    print_puzzle(state)
    print(f"Depth={depth}, α={alpha}, β={beta}")

    if depth == 0 or state == GOAL:
        score = -manhattan(state)
        print(f"Evaluation score: {score}\n")
        return score

    if maximizing:
        value = -math.inf
        for child in get_neighbors(state):
            value = max(value, alphabeta(child, depth-1, alpha, beta, False))
            alpha = max(alpha, value)
            if alpha >= beta:
                print("Pruned (MAX branch)\n")
                break
        return value
    else:
        value = math.inf
        for child in get_neighbors(state):
            value = min(value, alphabeta(child, depth-1, alpha, beta, True))
            beta = min(beta, value)
            if beta <= alpha:
                print("Pruned (MIN branch)\n")
                break
        return value

# -------- MAIN TEST ---------

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

print("\nStarting Alpha Beta Search on 8-Puzzle...\n")

best_value = alphabeta(initial, depth=3,
                       alpha=-math.inf,
```

```
                beta=math.inf,
                maximizing=True)

print("Final best evaluation:", best_value)
```

Output:

```
Starting Alpha Beta Search on 8-Puzzle...

State:
1 2 3
4 5 6
  7 8
Depth=3, α=-inf, β=inf

State:
1 2 3
4 5 6
7   8
Depth=2, α=-inf, β=inf

State:
1 2 3
4 5 6
7 8
Depth=1, α=-inf, β=inf
Evaluation score: 0
```