VISVESVARAYA TECHNOLOGICAL UNIVERSITY
"JnanaSangama", Belgaum -590014, Karnataka.

LAB REPORT
on

ARTIFICIAL INTELLIGENCE (23CS5PCAIN)

Submitted by

PRANAV SRINIVAS
[1BM22CS203]

in partial fulfillment for the award of the degree of
BACHELOR OF ENGINEERING
in
COMPUTER SCIENCE AND ENGINEERING

Submitted to

Prof. Swathi Sridharan
Assistant Professor

B.M.S. COLLEGE OF ENGINEERING
(Autonomous Institution under VTU)
BENGALURU – 560019.
September -- 2024 to January – 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 **Pranav Srinivas (1BM22CS203),** 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.

Signature of the Batch-In-Charge

Prof. Sneha S Bagalkot
Assistant Professor,
Department of Computer Science and Engineering,
B. M. S. College of Engineering.

Signature of the HOD

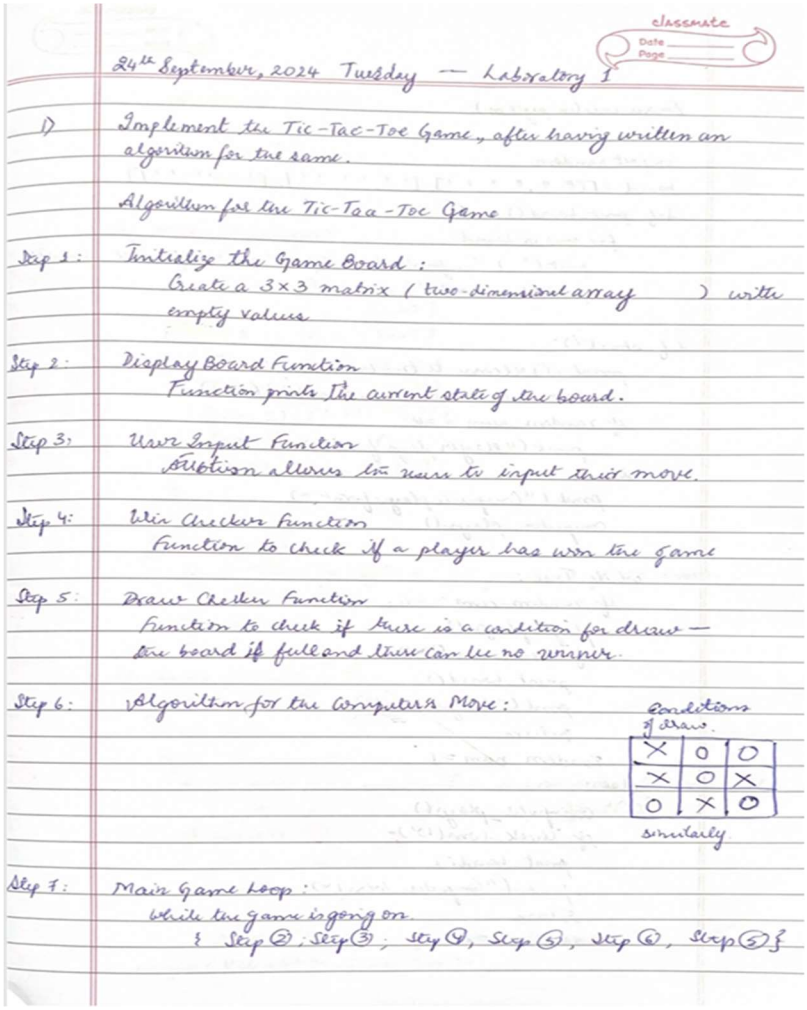
Dr. Kavitha Sooda
Professor & Head,
Department of Computer Science and Engineering,
B. M. S. College of Engineering.

Index

Github Link: https://github.com/pranavsrinivasdof/ARTIFICIAL-INTELLIGENCE-LABORATORY

Program 1
Implement Tic –Tac –Toe Game

Algorithm:



24th September, 2024 Tuesday — Laboratory 1

1) Implement the Tic-Tac-Toe Game, after having written an algorithm for the same.

Algorithm for the Tic-Tac-Toe Game

Step 1: Initialize the Game Board:
Create a 3×3 matrix (two-dimensional array) with empty values

Step 2: Display Board Function
Function prints the current state of the board.

Step 3: User Input Function
Function allows the users to input their move.

Step 4: Win Checker Function
Function to check if a player has won the game

Step 5: Draw Checker Function
Function to check if there is a condition for draw — the board if full and there can be no winner.

Step 6: Algorithm for the Computer's Move:

Conditions of draw.

X	O	O
X	O	X
O	X	O

similarly

Step 7: Main Game Loop:
While the game is going on
{ Step ②, Step ③, Step ④, Step ⑤, Step ⑥, Step ⑤ }

Code:

```
import random
def check(row, col):
    if arr[row][col] == '_':
        return True
    else:
        return False

def check2():
    for i in range(3):
        for j in range(3):
            if arr[i][j] == '_':
                return True
```

```
    return False

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

def check_win(player):
    for i in range(3):
        if all([arr[i][j] == player for j in range(3)]):
            return True
        if all([arr[j][i] == player for j in range(3)]):
            return True
    if arr[0][0] == player and arr[1][1] == player and arr[2][2] == player:
        return True
    if arr[0][2] == player and arr[1][1] == player and arr[2][0] == player:
        return True
    return False

arr = [['_', '_', '_'],
       ['_', '_', '_'],
       ['_', '_', '_']]
print_board()
while check2():
    row = int(input("Enter row (0-2): "))
    col = int(input("Enter column (0-2): "))
    while not check(row, col):
        print("Place already occupied, try again.")
        row = int(input("Enter row (0-2): "))
        col = int(input("Enter column (0-2): "))
    arr[row][col] = 'X'
    print_board()
    if check_win('X'):
        print("Congratulations! You win!")
        break
    if not check2():
        print("It's a draw!")
        break
    print("Computer's turn...")
    row, col = random.randint(0, 2), random.randint(0, 2)
    while not check(row, col):
        row, col = random.randint(0, 2), random.randint(0, 2)
    arr[row][col] = 'O'
    print_board()
    if check_win('O'):
        print("Computer wins! Better luck next time.")
        break
```

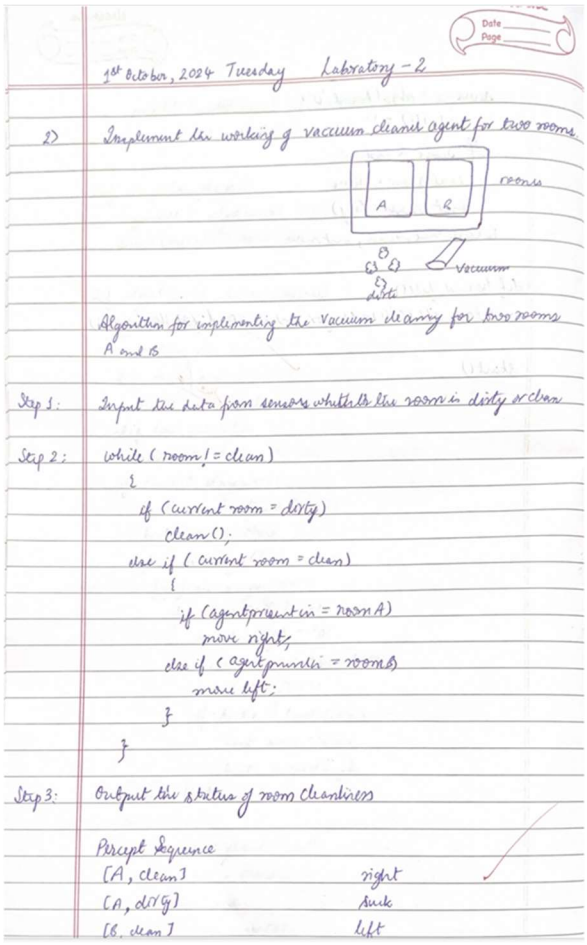
```
    if not check2():
        print("It's a draw!")
        break
```

Output Snapshot:

```
_ | _ | _
---------
_ | _ | _
---------
_ | _ | _
---------
Enter row (0-2): 1
Enter column (0-2): 1
_ | _ | _
---------
_ | X | _
---------
_ | _ | _
---------
Computer's turn...
_ | O | _
---------
_ | X | _
---------
_ | _ | _
---------
Enter row (0-2): 2
Enter column (0-2): 2
_ | O | _
---------
_ | X | _
---------
_ | _ | X
---------
Computer's turn...
_ | O | _
---------
_ | X | _
---------
O | _ | X
---------
Enter row (0-2): 0
Enter column (0-2): 0
X | O | _
---------
_ | X | _
---------
O | _ | X
---------
Congratulations! You win!
```

Implement vacuum cleaner agent

Algorithm:


1st October, 2024 Tuesday Laboratory - 2

2) Implement the working of vacuum cleaner agent for two rooms

Algorithm for implementing the vacuum cleaning for two rooms A and B

Step 1: Input the data from sensors whether the room is dirty or clean

Step 2:
```
while ( room1 = clean)
  {
    if (current room = dirty)
      clean();
    else if ( current room = clean)
      {
        if (agentpresent in = room A)
          move right;
        else if ( agentpresent = room B)
          move left;
      }
  }
```

Step 3: Output the status of room cleanliness

Percept Sequence

[A, clean]	right
[A, dirty]	suck
[B, clean]	left

[A, clean] [A, clean]	left
[A, clean] [A, dirty]	suck
[B, clean] [B, clean]	right
[B, clean] [B, dirty]	suck
[A, clean] [B clean]	exit

Code:

```
class VacuumCleaner:
    def __init__(self, room_a_dirt, room_b_dirt, starting_room):
        self.current_state = (room_a_dirt, room_b_dirt, starting_room)

    def is_goal_state(self):
        return self.current_state[0] == 0 and self.current_state[1] == 0

    def clean(self):
        if self.current_state[0] == 1:
            self.current_state = (0, self.current_state[1], self.current_state[2])
            print("Cleaned room A.")
        elif self.current_state[1] == 1:
            self.current_state = (self.current_state[0], 0, self.current_state[2])
            print("Cleaned room B.")
```

```
    def move(self):
        if self.current_state[2] == 'A':
            self.current_state = (self.current_state[0], self.current_state[1], 'B')
            print("Moved to room B.")
        else:
            self.current_state = (self.current_state[0], self.current_state[1], 'A')
            print("Moved to room A.")

    def run(self):
        while not self.is_goal_state():
            print(f"Current state: {self.current_state}")
            self.clean()
            if not self.is_goal_state():
                self.move()
        print("Both rooms are clean!")

def get_initial_state():
    room_a_dirt = int(input("Is room A dirty? (1 for yes, 0 for no): "))
    room_b_dirt = int(input("Is room B dirty? (1 for yes, 0 for no): "))
    starting_room = input("Which room is the vacuum cleaner in? (A or B): ").strip().upper()
    if starting_room not in ['A', 'B'] or room_a_dirt not in [0, 1] or room_b_dirt not in [0, 1]:
        print("Invalid input. Please enter the correct values.")
        return get_initial_state()

    return room_a_dirt, room_b_dirt, starting_room

initial_state = get_initial_state()
vacuum = VacuumCleaner(*initial_state)
vacuum.run()
```

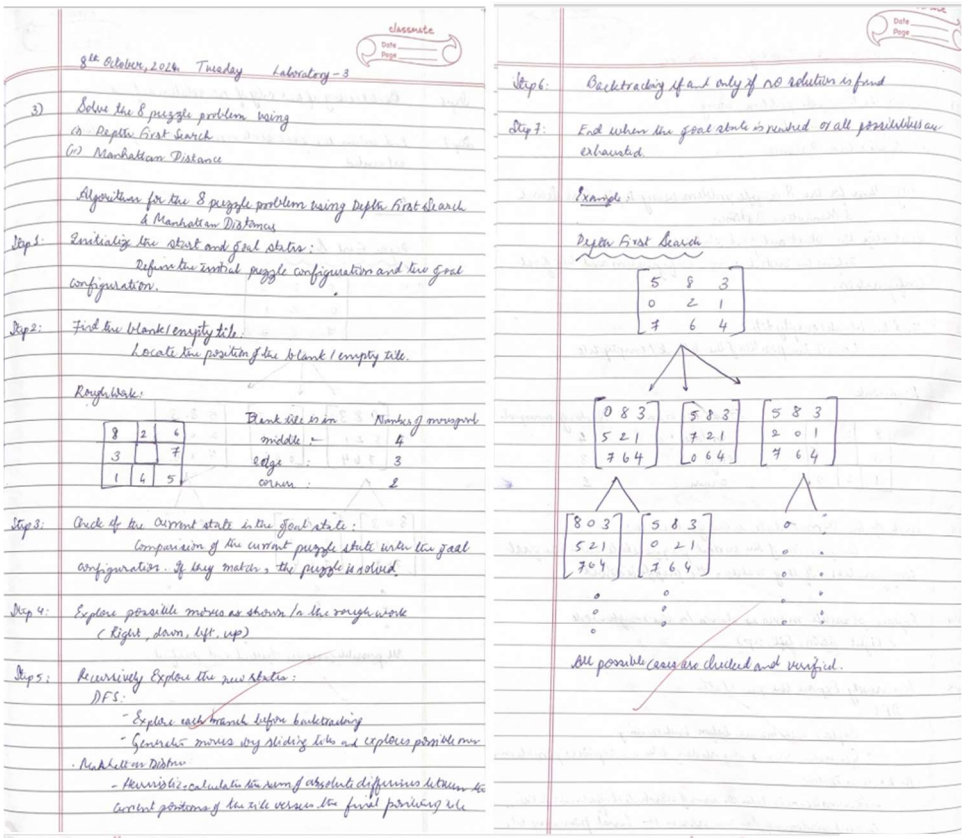
Output Snapshot:

```
Is room A dirty? (1 for yes, 0 for no): 1
Is room B dirty? (1 for yes, 0 for no): 1
Which room is the vacuum cleaner in? (A or B): A
Current state: (1, 1, 'A')
Cleaned room A.
Moved to room B.
Current state: (0, 1, 'B')
Cleaned room B.
Both rooms are clean!
```

Program2:

Implement 8 puzzle problems using Depth First Search (DFS)

Algorithm:



8th October, 2024 Tuesday Laboratory - 3

3) Solve the 8 puzzle problem using
(a) Depth First Search
(b) Manhattan Distance

Algorithm for the 8 puzzle problem using Depth First Search & Manhattan Distance

Step 1: Initialize the start and goal states:
Define the initial puzzle configuration and the goal configuration.

Step 2: Find the blank / empty tile.
Locate the position of the blank / empty tile.

Rough Work:

8	2	6
3		7
1	4	5

Blank tile is in — Number of movement
middle : 4
edge : 3
corner : 2

Step 3: Check if the current state is the goal state:
Comparison of the current puzzle state with the goal configuration. If they match, the puzzle is solved.

Step 4: Explore possible moves as shown in the rough work
(Right, down, left, up)

Step 5: Recursively Explore the new states:
DFS:
- Explore each branch before backtracking
- Generate moves by sliding tiles and explores possible mov

Manhattan Distance
- Heuristic: calculate the sum of absolute differences between the current positions of the tile versus the final position tile

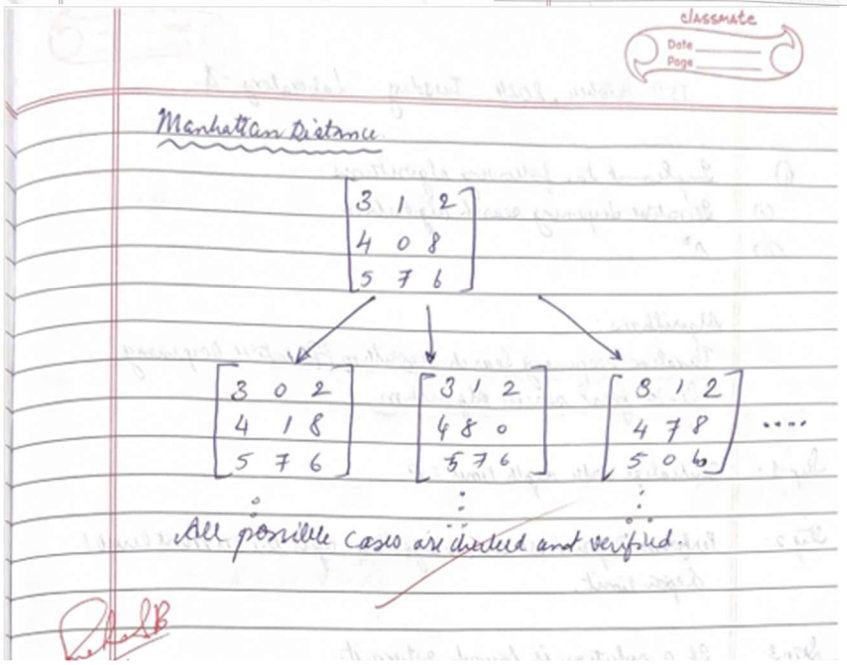
Step 6: Backtracking if and only if no solution is found

Step 7: End when the goal state is reached or all possibilities are exhausted.

Example:

Depth First Search

All possible cases are checked and verified.



Manhattan Distance

All possible cases are checked and verified.

Code:

```
from copy import deepcopy
goal_state = [[0, 1, 2],
              [3, 4, 5],
              [6, 7, 8]]
moves = {
    'up': (-1, 0),
    'down': (1, 0),
    'left': (0, -1),
    'right': (0, 1)
}

def find_blank(board):
    for i in range(3):
        for j in range(3):
            if board[i][j] == 0:
                return i, j
    return None

def is_goal(state):
    return state == goal_state

def is_valid_move(x, y):
    return 0 <= x < 3 and 0 <= y < 3

def apply_move(board, move):
    x, y = find_blank(board)
    dx, dy = moves[move]
    new_x, new_y = x + dx, y + dy
    if is_valid_move(new_x, new_y):
        new_board = deepcopy(board)

        new_board[x][y], new_board[new_x][new_y] = new_board[new_x][new_y],
new_board[x][y]
        return new_board
    return None

def dfs(start):
    stack = [(start, [])]
    visited = set()

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

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

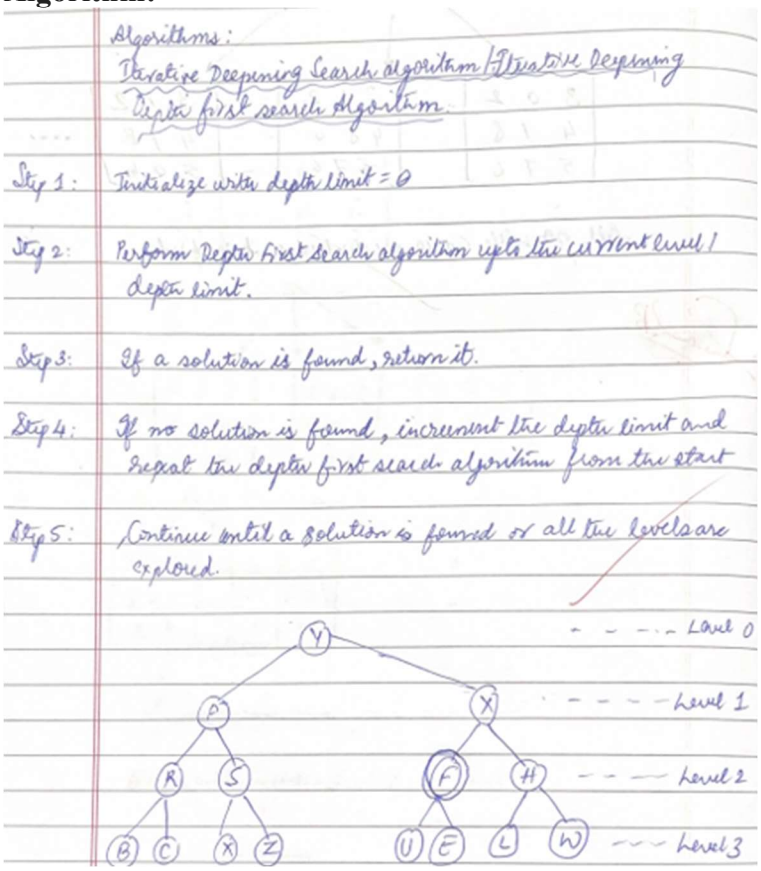
```
        visited.add(tuple(tuple(row) for row in current_state))

        for move in moves:
            new_state = apply_move(current_state, move)
            if new_state and tuple(tuple(row) for row in new_state) not in visited:
                stack.append((new_state, path + [current_state]))

    return None

def print_board(board):
    for row in board:
        print(row)
    print()

def print_solution(solution):
    if solution:
        for board in solution:
            print_board(board)
    else:
        print("No solution found")

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

solution = dfs(initial_state)

print_solution(solution)
```

Output Snapshot:

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

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

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

Implement Iterative deepening search algorithm

Algorithm:



Code:

```
from copy import deepcopy
goal_state = [[0, 1, 2],
              [3, 4, 5],
              [6, 7, 8]]
moves = {
    'up': (-1, 0),
    'down': (1, 0),
    'left': (0, -1),
    'right': (0, 1)
}

def find_blank(board):
    for i in range(3):
        for j in range(3):
            if board[i][j] == 0:
                return i, j
    return None
```

```
def is_goal(state):
    return state == goal_state

def is_valid_move(x, y):
    return 0 <= x < 3 and 0 <= y < 3

def apply_move(board, move):
    x, y = find_blank(board)
    dx, dy = moves[move]
    new_x, new_y = x + dx, y + dy
    if is_valid_move(new_x, new_y):
        new_board = deepcopy(board)
        new_board[x][y], new_board[new_x][new_y] = new_board[new_x][new_y],
new_board[x][y]
        return new_board
    return None

def dfs_limited(state, path, depth_limit, visited):
    if is_goal(state):
        return path + [state]

    if depth_limit == 0:
        return None

    visited.add(tuple(tuple(row) for row in state))

    for move in moves:
        new_state = apply_move(state, move)
        if new_state and tuple(tuple(row) for row in new_state) not in visited:
            result = dfs_limited(new_state, path + [state], depth_limit - 1, visited)
            if result:
                return result

    visited.remove(tuple(tuple(row) for row in state))
    return None

def ids(start):
    depth_limit = 0
    while True:
        visited = set()
        result = dfs_limited(start, [], depth_limit, visited)
        if result:
            return result
        depth_limit += 1

def print_board(board):
    for row in board:
```

```
        print(row)
    print()

def print_solution(solution):
    if solution:
        for board in solution:
            print_board(board)
    else:
        print("No solution found")

initial_state = [[1, 2, 5],
                 [0, 3, 8],
                 [6, 4, 7]]
solution = ids(initial_state)
print_solution(solution)
```

Output Snapshot:

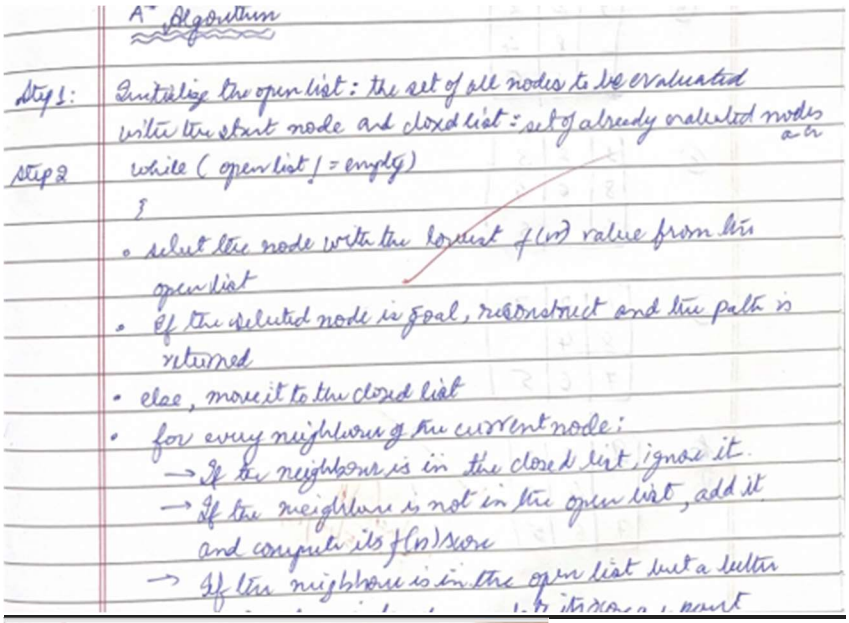
```
[1, 2, 5]
[0, 3, 8]
[6, 4, 7]

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

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

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

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

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

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

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

Program3:

Implement A* search algorithm

i) Misplaced tiles

Algorithm:



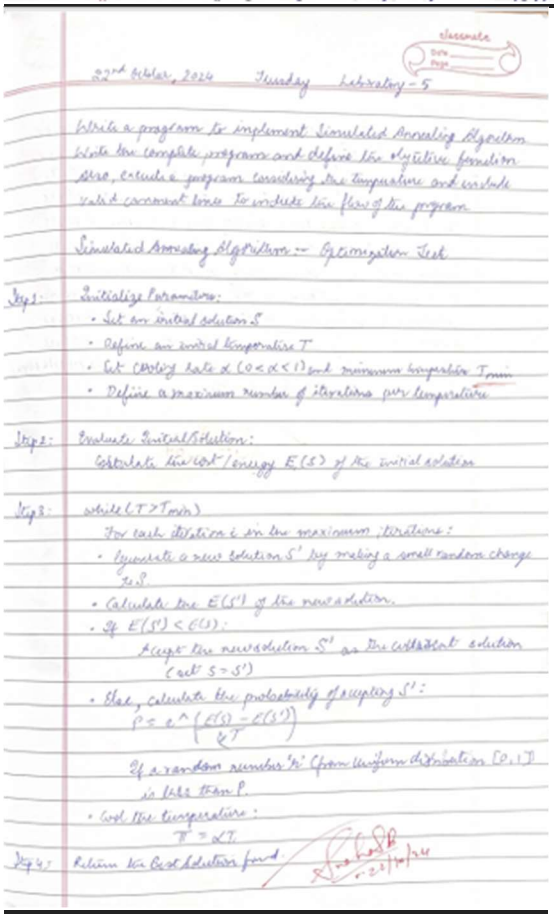
A* Algorithm

Step 1: Initialize the open list: the set of all nodes to be evaluated with the start node and closed list: set of already evaluated nodes

Step 2: while (open list != empty)

{

- select the node with the lowest f(n) value from the open list
- If the selected node is goal, reconstruct and the path is returned
- else, move it to the closed list
- for every neighbour of the current node:
  - → If the neighbour is in the closed list, ignore it.
  - → If the neighbour is not in the open list, add it and compute its f(n) score
  - → If the neighbour is in the open list but a better



Date Page

22nd October, 2024 Tuesday Laboratory - 5

Write a program to implement Simulated Annealing Algorithm. Write the complete program and define the objective function. Also, include a program considering the temperature and include valid comment lines to indicate the flow of the program.

Simulated Annealing Algorithm -- Optimization Task

Step 1: Initialize Parameters:

- Let an initial solution S
- Define an initial temperature T
- Let cooling rate α ($0 < \alpha < 1$) and minimum temperature T_{min}
- Define a maximum number of iterations per temperature

Step 2: Evaluate Initial Solution:

Calculate the cost/energy E(S) of the initial solution

Step 3: while ($T > T_{min}$)

For each iteration i in the maximum iterations:

- Generate a new solution S' by making a small random change to S
- Calculate the E(S') of the new solution
- If E(S') < E(S):

  Accept the new solution S' as the current solution (set S = S')
- Else, calculate the probability of accepting S':

  $P = e^{\left(\frac{E(S) - E(S')}{kT}\right)}$

  If a random number 'r' (from uniform distribution [0,1]) is less than P.
- Cool the temperature:

  $T = \alpha T$

Step 4: Return the best solution found

Code:

```
class Node:
    def __init__(self, data, level, fval):
        self.data = data
        self.level = level
        self.fval = fval

    def generate_child(self):
        x, y = self.find_blank()
        moves = [(x, y-1), (x, y+1), (x-1, y), (x+1, y)]
        children = []
        for new_x, new_y in moves:
            child_data = self.move_blank(x, y, new_x, new_y)
            if child_data:
                child_node = Node(child_data, self.level + 1, 0)
                children.append(child_node)
        return children

    def move_blank(self, x1, y1, x2, y2):
        if 0 <= x2 < len(self.data) and 0 <= y2 < len(self.data[0]):
            new_data = [row[:] for row in self.data]
            new_data[x1][y1], new_data[x2][y2] = new_data[x2][y2], new_data[x1][y1]
            return new_data
        return None

    def find_blank(self):
        for i, row in enumerate(self.data):
            if '_' in row:
                return i, row.index('_')


class Puzzle:
    def __init__(self, size):
        self.size = size
        self.open = []
        self.closed = []

    def get_input(self):
        return [input().split() for _ in range(self.size)]

    def f(self, start, goal):
        return start.level + self.h(start.data, goal)

    def h(self, start_data, goal):
        return sum(start_data[i][j] != goal[i][j] and start_data[i][j] != '_' for i in
range(self.size) for j in range(self.size))
```

```
    def process(self):
        print("Enter the start state matrix:")
        start_data = self.get_input()
        print("Enter the goal state matrix:")
        goal = self.get_input()

        start_node = Node(start_data, 0, 0)
        start_node.fval = self.f(start_node, goal)
        self.open.append(start_node)

        while self.open:
            current_node = self.open.pop(0)
            self.display_state(current_node, goal)

            if self.h(current_node.data, goal) == 0:
                print("Goal reached!")
                break

            children = current_node.generate_child()
            for child in children:
                child.fval = self.f(child, goal)
                self.open.append(child)

            self.closed.append(current_node)
            self.open.sort(key=lambda node: node.fval)

    def display_state(self, node, goal):
        print("\nNext step:")
        for row in node.data:
            print(" ".join(row))
        heuristic_value = self.h(node.data, goal)
        print(f"Heuristic (h): {heuristic_value}")
        print(f"Depth (g): {node.level}")
        print(f"Function value (f = g + h): {node.fval}")
puz = Puzzle(3)
puz.process()
```

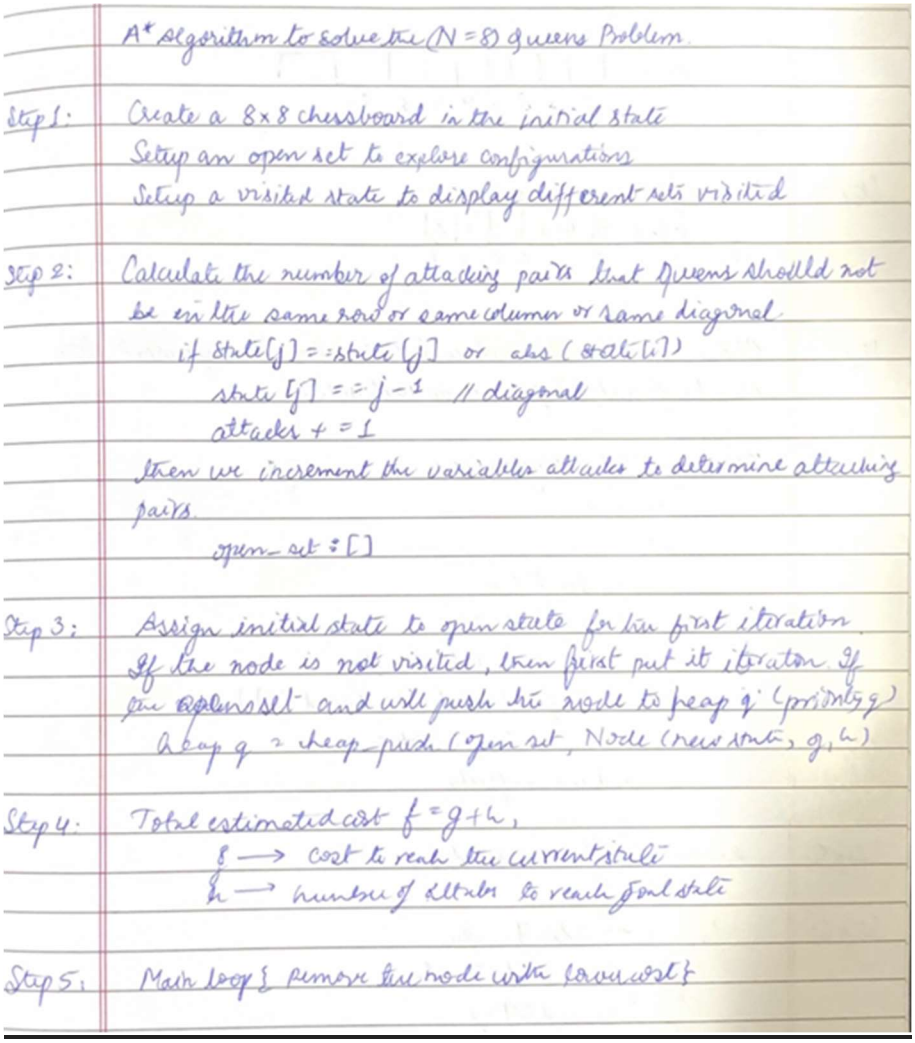
Output Snapshot:

```
Enter the start state matrix:
2 8 3
1 6 4
7 _ 5
Enter the goal state matrix:
1 2 3
8 _ 4
7 6 5

Next step:
2 8 3
1 6 4
7 _ 5
Heuristic (h): 4
Depth (g): 0
Function value (f = g + h): 4

Next step:
2 8 3
1 _ 4
7 6 5
Heuristic (h): 3
Depth (g): 1
Function value (f = g + h): 4

Next step:
2 8 3
_ 1 4
7 6 5
Heuristic (h): 3
Depth (g): 2
Function value (f = g + h): 5

Next step:
2 _ 3
1 8 4
7 6 5
Heuristic (h): 3
Depth (g): 2
Function value (f = g + h): 5
```

```
Next step:
_ 2 3
1 8 4
7 6 5
Heuristic (h): 2
Depth (g): 3
Function value (f = g + h): 5

Next step:
1 2 3
_ 8 4
7 6 5
Heuristic (h): 1
Depth (g): 4
Function value (f = g + h): 5

Next step:
1 2 3
8 _ 4
7 6 5
Heuristic (h): 0
Depth (g): 5
Function value (f = g + h): 5
Goal reached!
```

ii) Manhattan distance

Algorithm:


A* Algorithm to solve the (N=8) queens Problem

Step 1: Create a 8×8 chessboard in the initial state
Setup an open set to explore configurations
Setup a visited state to display different sets visited

Step 2: Calculate the number of attacking pairs that Queens should not be in the same row or same column or same diagonal.
if state[j] == state[j] or abs(state[i])
state[j] == j-1 // diagonal
attacks += 1
then we increment the variables attacks to determine attacking pairs.
open_set = []

Step 3: Assign initial state to open state for the first iteration
If the node is not visited, then first put it iteration of the open set and will push the node to heap q (priority q)
heap q = heap_push(open set, Node(new state, g, h))

Step 4: Total estimated cost f = g+h,
g → cost to reach the current state
h → number of attacks to reach goal state

Step 5: Main loop { remove the node with lowest f }

Code:

```
class Node:
    def __init__(self, data, level, fval):
        self.data = data
        self.level = level
        self.fval = fval

    def generate_child(self):
        x, y = self.find_blank()
        moves = [(x, y-1), (x, y+1), (x-1, y), (x+1, y)]
        children = []
        for new_x, new_y in moves:
```

```
            child_data = self.move_blank(x, y, new_x, new_y)
            if child_data:
                child_node = Node(child_data, self.level + 1, 0)
                children.append(child_node)
        return children

    def move_blank(self, x1, y1, x2, y2):
        if 0 <= x2 < len(self.data) and 0 <= y2 < len(self.data[0]):
            new_data = [row[:] for row in self.data]
            new_data[x1][y1], new_data[x2][y2] = new_data[x2][y2], new_data[x1][y1]
            return new_data
        return None

    def find_blank(self):
        for i, row in enumerate(self.data):
            if '_' in row:
                return i, row.index('_')


class Puzzle:
    def __init__(self, size):
        self.size = size
        self.open = []
        self.closed = []

    def get_input(self):
        return [input().split() for _ in range(self.size)]

    def f(self, start, goal):
        h_val = self.manhattan_heuristic(start.data, goal)
        return start.level + h_val, h_val

    def manhattan_heuristic(self, start_data, goal):
        distance = 0
        for i in range(self.size):
            for j in range(self.size):
                if start_data[i][j] != '_' and start_data[i][j] != goal[i][j]:
                    goal_x, goal_y = self.find_position(goal, start_data[i][j])
                    distance += abs(i - goal_x) + abs(j - goal_y)
        return distance

    def find_position(self, state, value):
        for i in range(self.size):
            for j in range(self.size):
                if state[i][j] == value:
                    return i, j
```

```
    def process(self):
        print("Enter the start state matrix:")
        start_data = self.get_input()
        print("Enter the goal state matrix:")
        goal = self.get_input()

        start_node = Node(start_data, 0, 0)
        start_node.fval, h_val = self.f(start_node, goal)
        self.open.append(start_node)

        while self.open:
            current_node = self.open.pop(0)
            self.display_state(current_node.data, current_node.fval, h_val, current_node.level)

            if self.manhattan_heuristic(current_node.data, goal) == 0:
                print("Goal reached!")
                break

            children = current_node.generate_child()
            for child in children:
                child.fval, h_val = self.f(child, goal)
                self.open.append(child)

            self.closed.append(current_node)
            self.open.sort(key=lambda node: node.fval)

    def display_state(self, data, f_val, h_val, g_val):
        print("\nNext step:")
        for row in data:
            print(" ".join(row))
        print(f"f(x) = {f_val} (g(x) = {g_val}, h(x) = {h_val})")
puz = Puzzle(3)
puz.process()
```

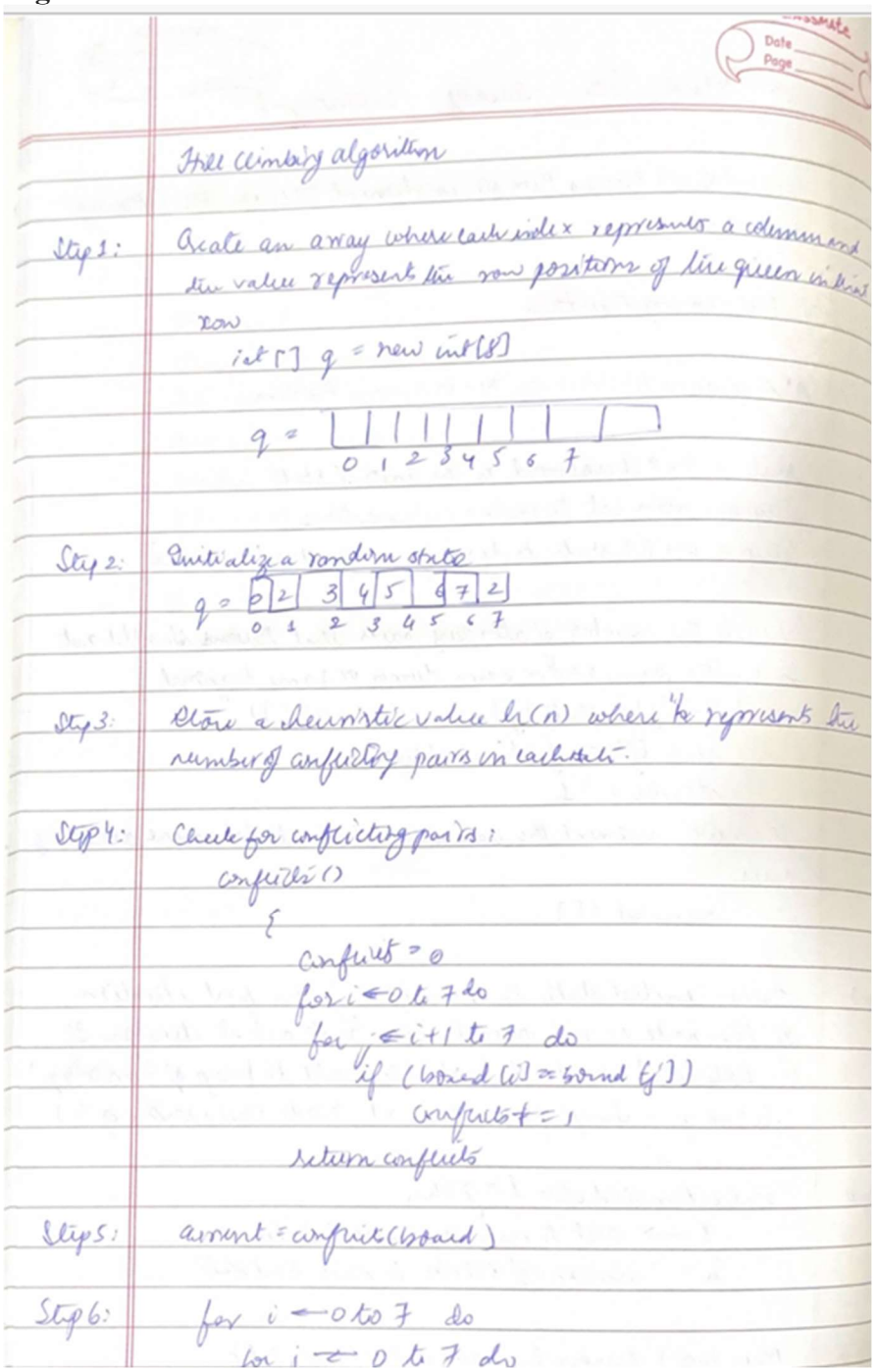
output snapshot:

```
Enter the start state matrix:
2 8 3
1 6 4
_ 7 5
Enter the goal state matrix:
1 2 3
8 _ 4
7 6 5

Next step:
2 8 3
1 6 4
_ 7 5
f(x) = 6 (g(x) = 0, h(x) = 6)

Next step:
2 8 3
1 6 4
7 _ 5
f(x) = 6 (g(x) = 1, h(x) = 7)

Next step:
2 8 3
1 _ 4
7 6 5
f(x) = 6 (g(x) = 2, h(x) = 4)

Next step:
2 _ 3
1 8 4
7 6 5
f(x) = 6 (g(x) = 3, h(x) = 5)

Next step:
_ 2 3
1 8 4
7 6 5
f(x) = 6 (g(x) = 4, h(x) = 4)

Next step:
1 2 3
_ 8 4
7 6 5
f(x) = 6 (g(x) = 5, h(x) = 1)

Next step:
1 2 3
8 _ 4
7 6 5
f(x) = 6 (g(x) = 6, h(x) = 2)
Goal reached!
```

Program4:

Implement Hill Climbing search algorithm to solve N-Queens problem

Algorithm:



Hill climbing algorithm

Step 1: Create an array where each index represents a column and the value represents the row position of the queen in that row

```
int[] q = new int[8]
```

q = | | | | | | | | |
0 1 2 3 4 5 6 7

Step 2: Initialize a random state

q = | 0 | 2 | 3 | 4 | 5 | 6 | 7 | 2 |
0 1 2 3 4 5 6 7

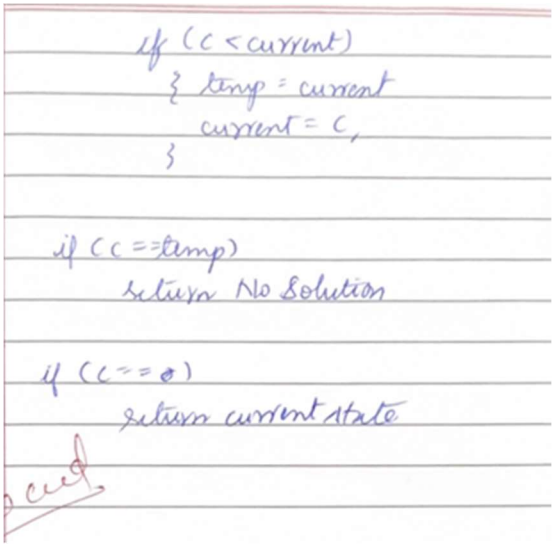
Step 3: Store a heuristic value h(n) where 'h' represents the number of conflicting pairs in each state.

Step 4: Check for conflicting pairs:

```
conflicts()
{
    conflicts = 0
    for i ← 0 to 7 do
        for j ← i+1 to 7 do
            if (board[i] == board[j])
                conflicts += 1
    return conflicts
```

Step 5: current = conflicts(board)

Step 6: for i ← 0 to 7 do
for i ← 0 to 7 do



Code:

```
def calculateCost(state):
    attacking_pairs = 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):
                attacking_pairs += 1
    return attacking_pairs

def getNeighbours(state):
    neighbours = []
    for i in range(N):
        for j in range(i + 1, N):
            new_state = state[:]
            new_state[i], new_state[j] = new_state[j], new_state[i]
            neighbours.append(new_state)
    return neighbours

def hillClimbing(initial_state):
    current_state = initial_state
    current_cost = calculateCost(current_state)

    iteration = 0
    while True:
        print(f"\nIteration {iteration}")
        print(f"Current State: {current_state}, Cost: {current_cost}")

        neighbours = getNeighbours(current_state)
        next_state = current_state
        next_cost = current_cost

        for neighbour in neighbours:
            cost = calculateCost(neighbour)
```

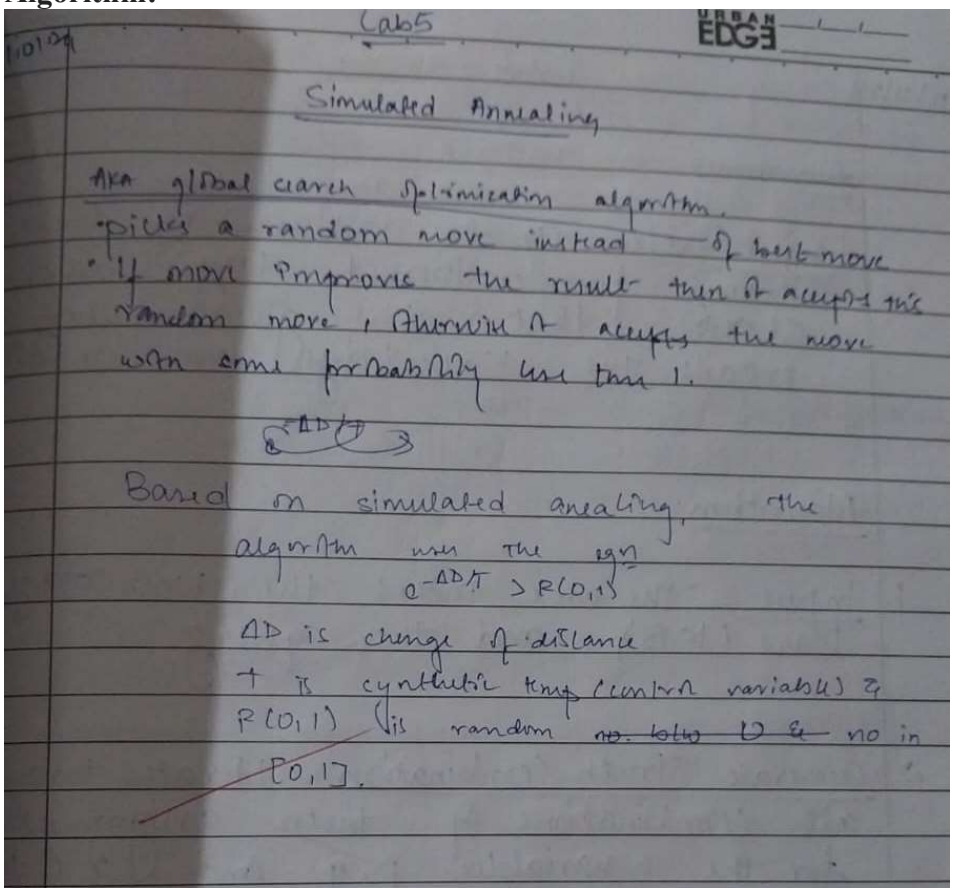
```
            print(f"Neighbour: {neighbour}, Cost: {cost}")
            if cost < next_cost:
                next_state = neighbour
                next_cost = cost

        if next_cost == current_cost:
            break
        else:
            current_state, current_cost = next_state, next_cost

        if current_cost == 0:
            break

        iteration += 1

    return current_state, current_cost

N=4 #Board Size
initial_state = list(map(int, input("Enter initial state as space-separated integers (0-based
indexing): ").split()))
solution_state, solution_cost = hillClimbing(initial_state)

print("\nFinal Results")
print("Initial State:", initial_state)
print("Final State (Solution):", solution_state)
print("Final Cost (Attacking Pairs):", solution_cost)

if solution_cost == 0:
    print("Solution found!")
else:
    print("Local optimum reached, but no solution.")
```

Output Snapshot:

```
Enter initial state as space-separated integers (0-based indexing): 3 1 2 0

Iteration 0
Current State: [3, 1, 2, 0], Cost: 2
Neighbour: [1, 3, 2, 0], Cost: 1
Neighbour: [2, 1, 3, 0], Cost: 1
Neighbour: [0, 1, 2, 3], Cost: 6
Neighbour: [3, 2, 1, 0], Cost: 6
Neighbour: [3, 0, 2, 1], Cost: 1
Neighbour: [3, 1, 0, 2], Cost: 1

Iteration 1
Current State: [1, 3, 2, 0], Cost: 1
Neighbour: [3, 1, 2, 0], Cost: 2
Neighbour: [2, 3, 1, 0], Cost: 2
Neighbour: [0, 3, 2, 1], Cost: 4
Neighbour: [1, 2, 3, 0], Cost: 4
Neighbour: [1, 0, 2, 3], Cost: 2
Neighbour: [1, 3, 0, 2], Cost: 0

Final Results
Initial State: [3, 1, 2, 0]
Final State (Solution): [1, 3, 0, 2]
Final Cost (Attacking Pairs): 0
Solution found!
```

Program5:
Simulated Annealing to Solve 8-Queens problem

Algorithm:



Code:

```
import random
import math

n=8

def no_of_conflicts(board):
    conflicts = 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]) == j - i:
                conflicts += 1
    return conflicts

def initialize(n):
    return [random.randint(0, n - 1) for _ in range(n)]
```

```
def simulated_annealing(n, max_iterations=10000, initial_temp=100, cooling_rate=0.99):
    board = initialize(n)
    current_conflicts = no_of_conflicts(board)
    temperature = initial_temp

    for iteration in range(max_iterations):
        if current_conflicts == 0:
            print(f"Iteration {iteration}: Solution found!")
            print("Board Position:", board)
            return board

        row = random.randint(0, n - 1)
        new_col = random.randint(0, n - 1)
        while new_col == board[row]:
            new_col = random.randint(0, n - 1)

        new_board = board[:]
        new_board[row] = new_col
        new_conflicts = no_of_conflicts(new_board)
        delta_conflicts = new_conflicts - current_conflicts
        if delta_conflicts < 0 or math.exp(-delta_conflicts / temperature) > random.random():
            board, current_conflicts = new_board, new_conflicts
        print(f"Iteration {iteration}: Temperature={temperature:.2f}, Current
Conflicts={current_conflicts}, "
              f"Delta Conflicts={delta_conflicts}, New Position for Row {row} -> Column
{new_col}")
        print("Board Position:", board)
        temperature *= cooling_rate

    print("No solution found within the maximum iterations.")
    return None

solution = simulated_annealing(n)
if solution:
    print("Final Solution:", solution)
else:
    print("No solution found within the maximum iterations.")
```

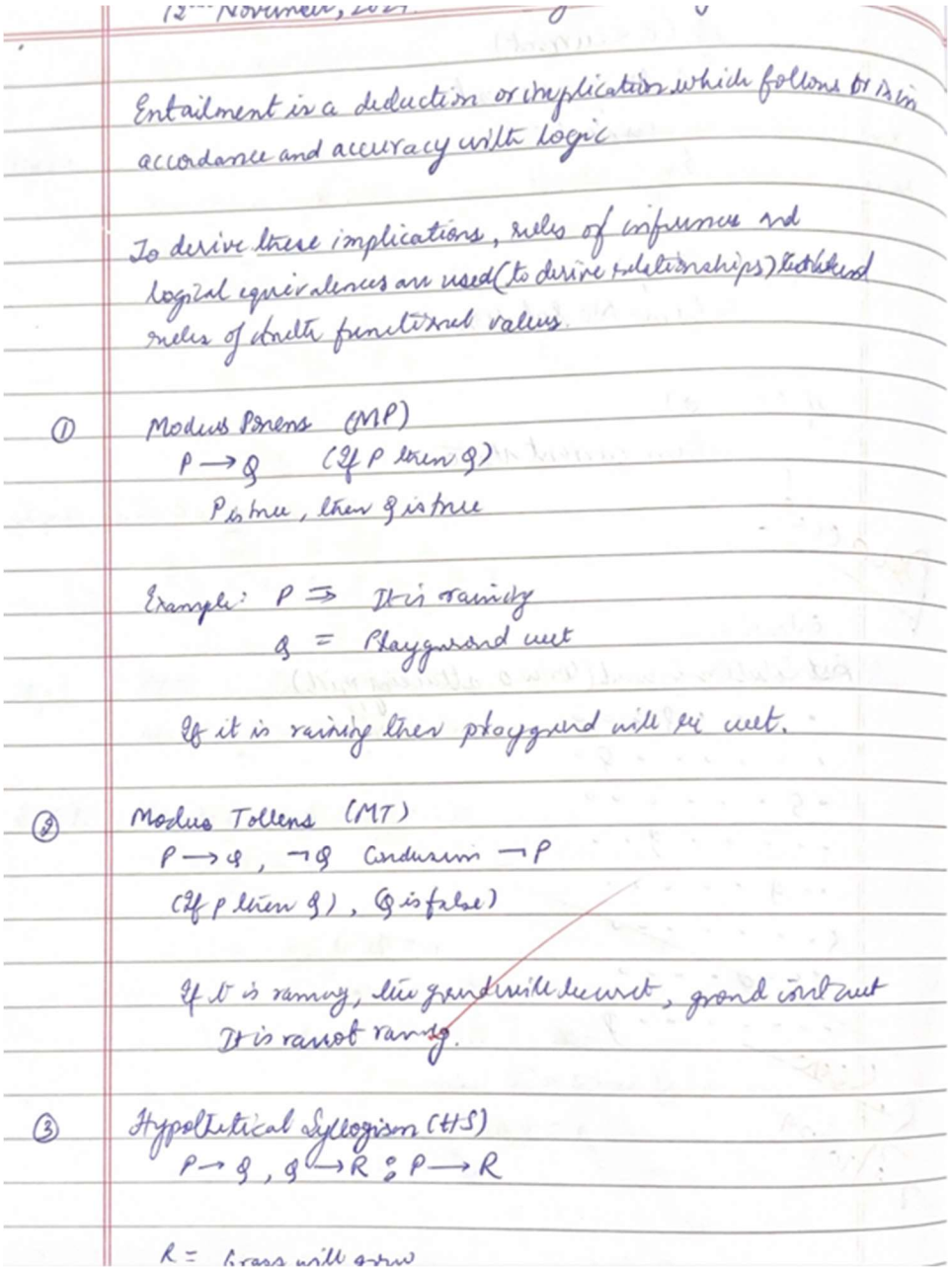
Output Snapshot:

```
Board Position: [4, 6, 3, 5, 7, 1, 4, 2]
Iteration 1227: Temperature=0.00, Current Conflicts=1, Delta Conflicts=3, New Position for Row 4 -> Column 5
Board Position: [4, 6, 3, 5, 7, 1, 4, 2]
Iteration 1228: Temperature=0.00, Current Conflicts=1, Delta Conflicts=1, New Position for Row 0 -> Column 2
Board Position: [4, 6, 3, 5, 7, 1, 4, 2]
Iteration 1229: Temperature=0.00, Current Conflicts=1, Delta Conflicts=3, New Position for Row 7 -> Column 4
Board Position: [4, 6, 3, 5, 7, 1, 4, 2]
Iteration 1230: Temperature=0.00, Current Conflicts=1, Delta Conflicts=1, New Position for Row 0 -> Column 1
Board Position: [4, 6, 3, 5, 7, 1, 4, 2]
Iteration 1231: Temperature=0.00, Current Conflicts=0, Delta Conflicts=-1, New Position for Row 0 -> Column 0
Board Position: [0, 6, 3, 5, 7, 1, 4, 2]
Iteration 1232: Solution found!
Board Position: [0, 6, 3, 5, 7, 1, 4, 2]
Final Solution: [0, 6, 3, 5, 7, 1, 4, 2]
```

Program6:

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

Algorithm:



Entailment is a deduction or implication which follows or is in accordance and accuracy with logic

To derive these implications, rules of inference and logical equivalences are used (to derive relationships) and rules of truth functional values.

① Modus Ponens (MP)
$P \rightarrow Q$ (If P then Q)
P is true, then Q is true

Example: P ⇒ It is raining
Q = Playground wet

If it is raining then playground will be wet.

② Modus Tollens (MT)
$P \rightarrow Q$, $\neg Q$ Conclusion $\neg P$
(If P then Q), Q is false)

If it is raining, the ground will be wet, ground is not wet
It is raining.

③ Hypothetical Syllogism (HS)
$P \rightarrow Q$, $Q \rightarrow R \therefore P \rightarrow R$

R = Grass will grow

Code:

```
def implies(p, q):
    return not p or q

def print_truth_table(KB, alpha, sym):
```

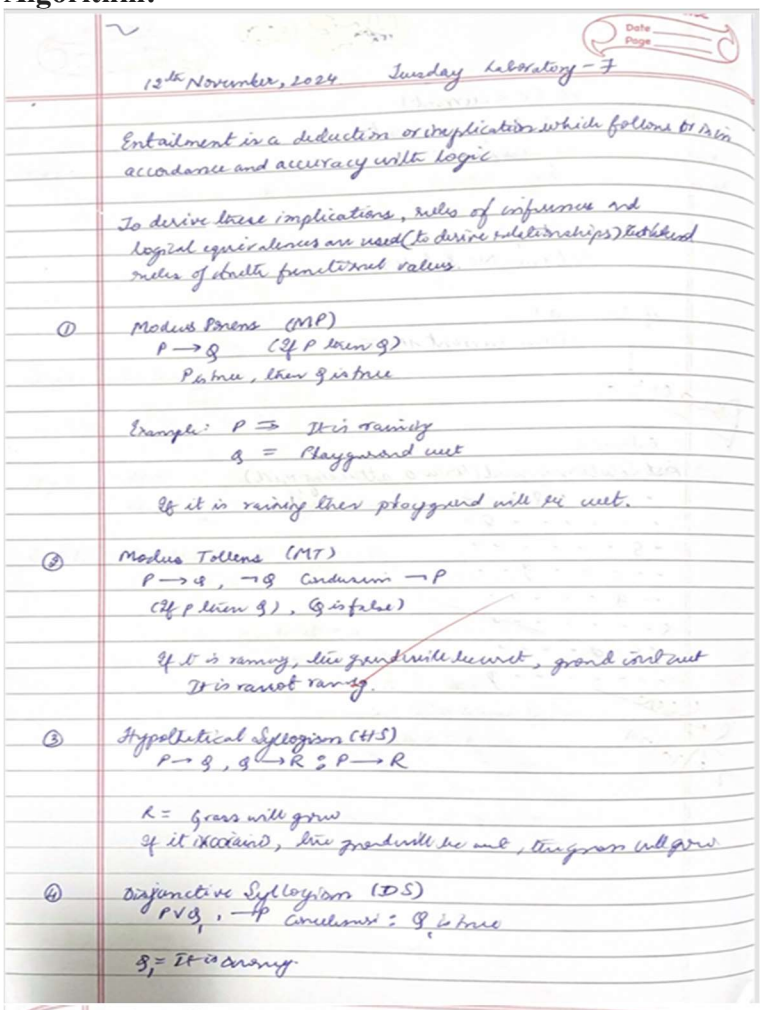
```
    print("Truth Table:")
    header = " | ".join(sym) + " | KB | alpha "
    print(header)
    print("-" * len(header))

    for values in product([True, False], repeat=len(sym)):
        model = dict(zip(sym, values))
        kb_true = all(statement(model) for statement in KB)
        query_true = query(model)

        row = " | ".join(str(val) for val in values) + f" | {kb_true} | {query_true}"
        print(row)

def check_entailment(KB, query, sym):

    for values in product([True, False], repeat=len(sym)):
        model = dict(zip(sym, values))
        kb_true = all(statement(model) for statement in KB)
        query_true = query(model)
        if kb_true and not query_true:
            return False
    return True
sym=['P', 'Q', 'R']
KB = [
    lambda model: implies(model['Q'], model['P']),
    lambda model: implies(model['P'], not model['Q']),
    lambda model: model['Q'] or model['R']
]
alpha = lambda model: model['R']


entails = check_entailment(KB, alpha,sym)
print(f"KB entails alpha: {entails}")
print_truth_table(KB, alpha,sym)
```

Output Snapshot:

```
KB entails alpha: True
Truth Table:
P | Q | R | KB | alpha
----------------------
True | True | True | False | True
True | True | False | False | False
True | False | True | True | True
True | False | False | False | False
False | True | True | False | True
False | True | False | False | False
False | False | True | True | True
False | False | False | False | False
```

Program7:
Implement unification in first order logic

Algorithm:



12th November, 2024 Tuesday Laboratory - 7

Entailment is a deduction or implication which follows logically in accordance and accuracy with logic

To derive these implications, rules of inference and logical equivalences are used (to derive relationships) established rules of truth functional values

① Modus Ponens (MP)
$P \rightarrow Q$ (If P then Q)
P is true, then Q is true

Example: P ⇒ It is raining
Q = Playground wet

If it is raining then playground will be wet.

② Modus Tollens (MT)
$P \rightarrow Q$, $\neg Q$ Conclusion $\neg P$
(If P then Q), Q is false)

If it is raining, the ground will be wet, ground is not wet
It is raining.

③ Hypothetical Syllogism (HS)
$P \rightarrow Q$, $Q \rightarrow R$ ∴ $P \rightarrow R$

R = Grass will grow
If it is raining, the ground will be wet, then grass will grow

④ Disjunctive Syllogism (DS)
$P \vee Q_1$, $\neg P$ Conclusion: Q_1 is true

Q_1 = It is raining.

Either it is raining or sunning
It is raining ✗
Then, it is sunning

⑤ Bi-conditional:
If $P \rightarrow Q$ is true, $P \rightarrow Q$ and $Q \rightarrow P$ are true

Conclusion:
$P \rightarrow Q$

$Q \rightarrow P$

⑥ Contradiction
P leads to a contradiction, $\neg P$ must be true

Raining & ground not wet.
Conclusion: $\neg(P \wedge \neg Q)$

Code:

```
def unify_terms(term_a, term_b, subs=None):
    if subs is None:
        subs = {}

    if term_a == term_b:
        return subs

    if is_variable(term_a):
        return unify_with_var(term_a, term_b, subs)
    if is_variable(term_b):
        return unify_with_var(term_b, term_a, subs)

    if is_compound(term_a) and is_compound(term_b):
        if term_a[0] != term_b[0] or len(term_a[1]) != len(term_b[1]):
            return None
        for subterm_a, subterm_b in zip(term_a[1], term_b[1]):
            subs = unify_terms(subterm_a, subterm_b, subs)
            if subs is None:
                return None
        return subs

    if isinstance(term_a, list) and isinstance(term_b, list):
        if len(term_a) != len(term_b):
            return None
        for element_a, element_b in zip(term_a, term_b):
            subs = unify_terms(element_a, element_b, subs)
            if subs is None:
                return None
        return subs

    return None
def unify_with_var(var, expr, subs):

    if var in subs:
        return unify_terms(subs[var], expr, subs)
    if expr in subs:
        return unify_terms(var, subs[expr], subs)
    if occurs_check(var, expr, subs):
        return None  # Cyclic substitution check failed
    subs[var] = expr
    return subs


def occurs_check(var, expr, subs):

    if var == expr:
```

```
        return True
    if is_compound(expr):
        return any(occurs_check(var, arg, subs) for arg in expr[1])
    if isinstance(expr, list):
        return any(occurs_check(var, item, subs) for item in expr)
    if expr in subs:
        return occurs_check(var, subs[expr], subs)
    return False


def is_variable(item):

    return isinstance(item, str) and item.startswith('?')


def is_compound(item):

    return isinstance(item, tuple) and len(item) == 2 and isinstance(item[1], list)


if __name__ == "__main__":
    print("Enter expressions in the following format:")
    print("Compound terms: ('f', ['a', 'b'])")
    print("Variables: '?x', '?y'")
    print("Lists: ['a', 'b']")
    print("Constants: 'a', 'b', etc.\n")

    term_1 = eval(input("Enter the first expression (Ψ₁): "))
    term_2 = eval(input("Enter the second expression (Ψ₂): "))

    substitution_result = unify_terms(term_1, term_2)
    if substitution_result is None:
        print("Unification failed!")
    else:
        print("Unification successful!")
        print("Substitution Set:", substitution_result)
```

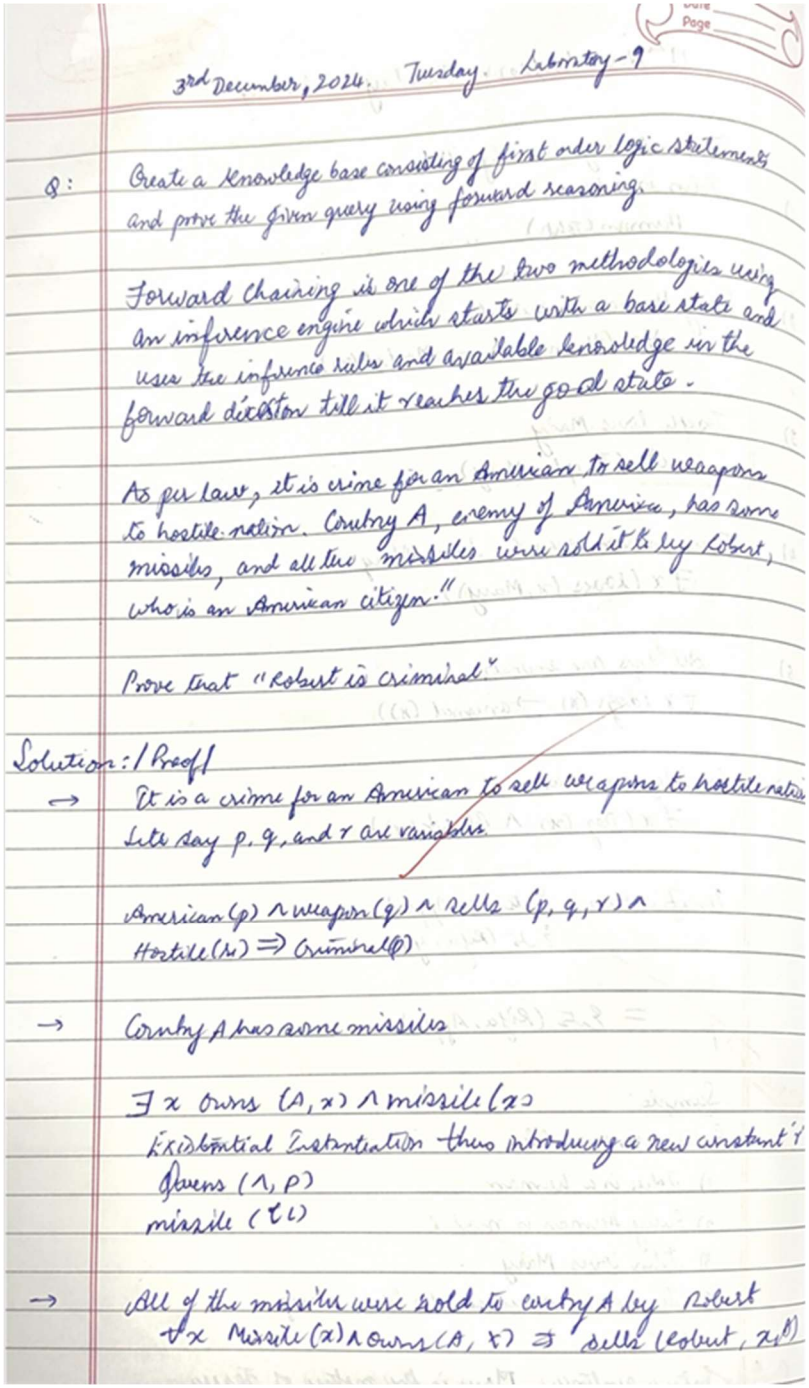
Output Snapshot:

```
Enter expressions in the following format:
Compound terms: ('f', ['a', 'b'])
Variables: '?x', '?y'
Lists: ['a', 'b']
Constants: 'a', 'b', etc.

Enter the first expression (Ψ₁): ('Studies',['Abubakar','?x'])
Enter the second expression (Ψ₂): ('Studies',['?y','AI'])
Unification successful!
Substitution Set: {'?y': 'Abubakar', '?x': 'AI'}
```

Program8:

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

Algorithm:



3rd December, 2024. Tuesday. Laboratory - 9

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

Forward chaining is one of the two methodologies using an inference engine which starts with a base state and uses the inference rules and available knowledge in the forward direction till it reaches the goal state.

As per law, it is crime for an American to sell weapons to hostile nation. Country A, enemy of America, has some missiles, and all the missiles were sold it to by Robert, who is an American citizen."

Prove that "Robert is criminal."

Solution: / Proof /

→ It is a crime for an American to sell weapons to hostile nation. Lets say p, q, and r are variables

American(p) ∧ weapon(q) ∧ sells(p, q, r) ∧ Hostile(r) ⇒ Criminal(p)

→ Country A has some missiles

∃ x owns(A, x) ∧ missile(x)

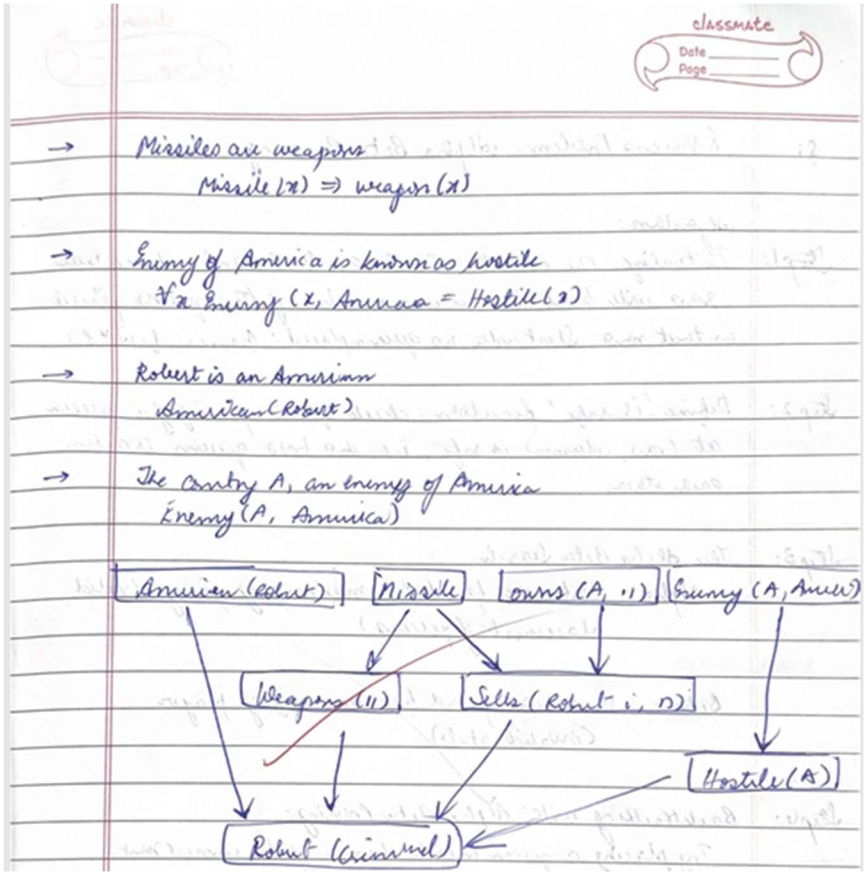
Existential Instantiation thus introducing a new constant T

Owns(A, T)

missile(T)

→ All of the missiles were sold to country A by Robert

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



Code:

```
class KnowledgeBaseSystem:
    def __init__(self):
        self.known_facts = set()
        self.rules_list = []

    def insert_fact(self, fact):
        self.known_facts.add(fact)

    def insert_rule(self, rule):
        self.rules_list.append(rule)

    def deduce(self):
        new_inferences = True
        while new_inferences:
            new_inferences = False
            for rule in self.rules_list:
                if rule.check_and_apply(self.known_facts):
                    new_inferences = True

knowledge_base = KnowledgeBaseSystem()
knowledge_base.insert_fact("American(Robert)")
knowledge_base.insert_fact("Missile(T1)")
knowledge_base.insert_fact("Owns(A, T1)")
```

```
knowledge_base.insert_fact("Enemy(A, America)")


class InferenceRule:
    def __init__(self, conditions, result):
        self.conditions = conditions  # List of conditions
        self.result = result  # The result to derive if conditions are met

    def check_and_apply(self, facts):
        if all(condition in facts for condition in self.conditions):
            if self.result not in facts:
                facts.add(self.result)
                print(f"Derived: {self.result}")
                return True
        return False



knowledge_base.insert_rule(InferenceRule(["Missile(T1)"],"Weapon(T1)"))
knowledge_base.insert_rule(InferenceRule(["Enemy(A, America)"],"Hostile(A)"))
knowledge_base.insert_rule(InferenceRule(["Missile(T1)", "Owns(A, T1)"],"Sells(Robert,
T1, A)"))
knowledge_base.insert_rule(InferenceRule(
    ["American(Robert)", "Weapon(T1)", "Sells(Robert, T1, A)",
"Hostile(A)"],"Criminal(Robert)"))
knowledge_base.deduce()

if "Criminal(Robert)" in knowledge_base.known_facts:
    print("Conclusion: Robert is a criminal.")
else:
    print("Conclusion: Unable to prove Robert is a criminal.")
```

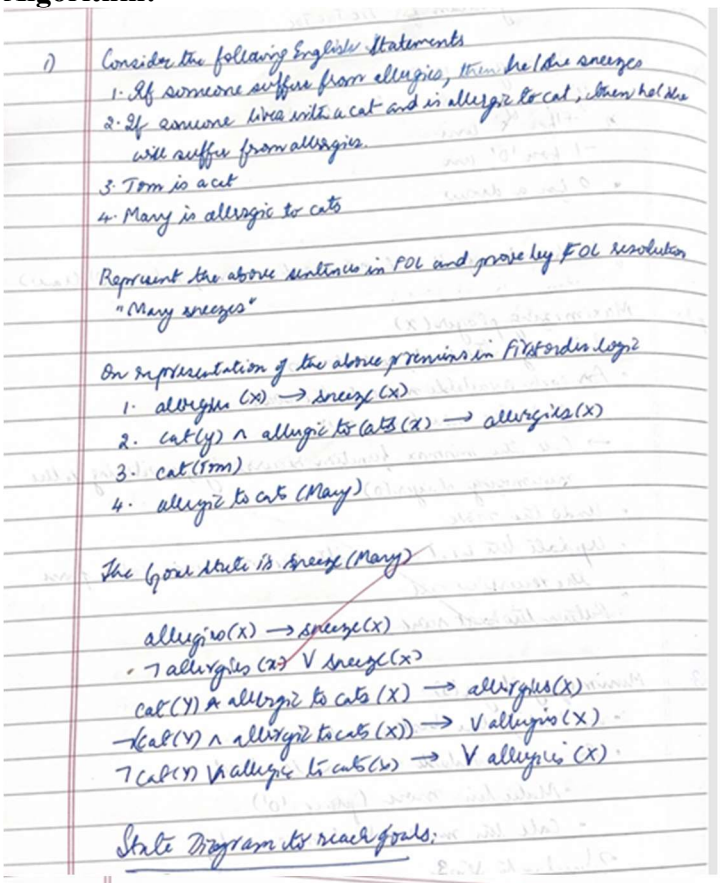
Output Snapshot:

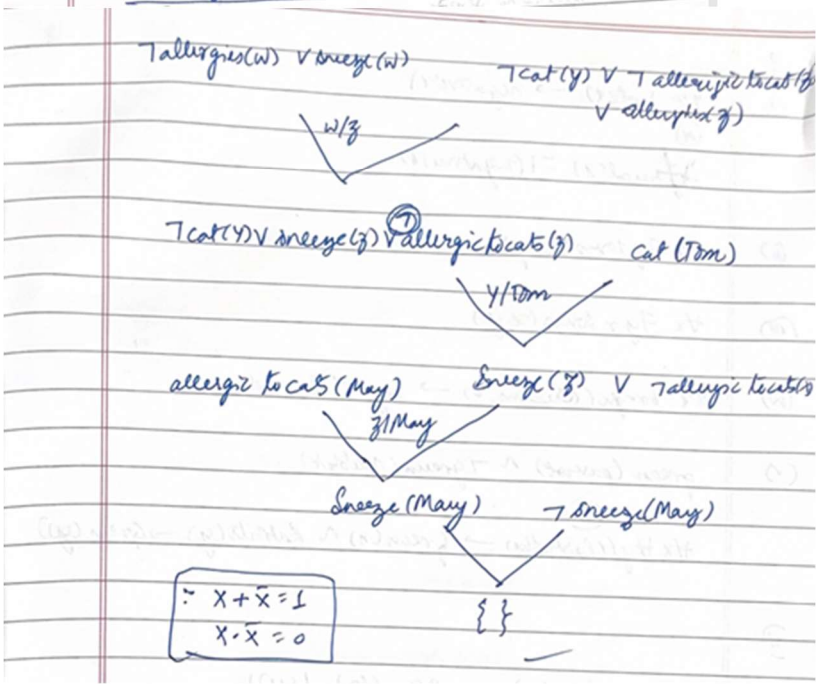
```
Derived: Weapon(T1)
Derived: Hostile(A)
Derived: Sells(Robert, T1, A)
Derived: Criminal(Robert)
Conclusion: Robert is a criminal.
```

Program9:

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

Algorithm:





Code:

```
from itertools import combinations

def unify_sentences_var(var, x, theta):
    if var in theta:
        return unify_sentences(theta[var], x, theta)
    elif x in theta:
        return unify_sentences(var, theta[x], theta)
    else:
        theta[var] = x
        return theta

def resolve(sentence1, sentence2):
    resolvents = []
    for predicate1 in sentence1:
        for predicate2 in sentence2:
            theta = unify_sentences(predicate1, negate(predicate2))
            if theta is not None:
                new_sentence = (substitute(sentence1, theta) | substitute(sentence2, theta)) -
{predicate1, predicate2}
                resolvents.append(frozenset(new_sentence))
    return resolvents

def unify_sentences(x, y, theta={}):
    if theta is None:
        return None
    elif x == y:
        return theta
    elif isinstance(x, str) and x.islower():
        return unify_sentences_var(x, y, theta)
    elif isinstance(y, str) and y.islower():
        return unify_sentences_var(y, x, theta)
    elif isinstance(x, tuple) and isinstance(y, tuple) and len(x) == len(y):
        return unify_sentences(x[1:], y[1:], unify_sentences(x[0], y[0], theta))
    else:
        return None

def negate(predicate):
    return ('not', predicate) if isinstance(predicate, str) else predicate[1]

def substitute(sentence, theta):
    return {substitute_predicate(p, theta) for p in sentence}

def substitute_predicate(predicate, theta):
    if isinstance(predicate, str):
        return theta.get(predicate, predicate)
```

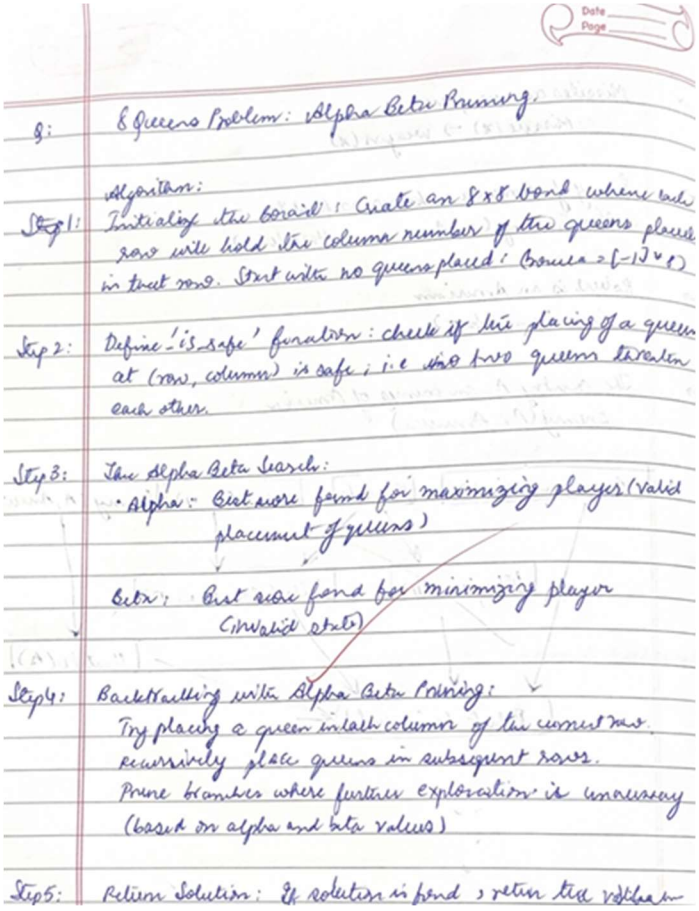
```
    else:
        return (predicate[0],) + tuple(theta.get(arg, arg) for arg in predicate[1:])

def proof_by_resolution(Knowledge_Base, query):
    negated_query = frozenset({negate(query)})
    sentences = Knowledge_Base | {negated_query}
    new_sentences = set()

    while True:
        for sentence1, sentence2 in combinations(sentences, 2):
            resolvents = resolve(sentence1, sentence2)
            if frozenset() in resolvents:
                return True
            new_sentences.update(resolvents)
        if new_sentences.issubset(sentences):
            return False
        sentences |= new_sentences

Knowledge_Base = {
    frozenset({('Mother', 'Leela', 'Oshin')}),
    frozenset({('Alive', 'Leela')}),
    frozenset({('not','Mother', 'x','y')}),
    frozenset({('Parent','x','y')}),
    frozenset({('not','Parent', 'w', 'z')}),
    frozenset({('not','Alive','w','z')}),
    frozenset({('Older','w','z')}),
}

query = ('Older', 'Leela', 'Older')
result = proof_by_resolution(Knowledge_Base, query)
if result:
    print("Leela is older than Oshin.\nProved by resolution.")
else:
    print("Cannot prove. Leela is not older than Oshin.")
```

Output Snapshot:

```
Leela is older than Oshin.
Proved by resolution.
```

Program10:
Implement Alpha-Beta Pruning.

Algorithm:



Code:

```
class Node:
    def __init__(self, value=None, children=None):
        self.value = value
        self.children = children if children else []

def alpha_beta_pruning(node, depth, alpha, beta, maximizing_player):
    if not node.children or depth == 0:
        return node.value

    if maximizing_player:
        max_eval = float('-inf')
        for child in node.children:
            eval = alpha_beta_pruning(child, depth - 1, alpha, beta, False)
            max_eval = max(max_eval, eval)
            alpha = max(alpha, eval)
            if beta <= alpha:
```

```
                print(f"Pruned at MAX node with alpha={alpha}, beta={beta}")
                break
        node.value = max_eval
        return max_eval
    else:
        min_eval = float('inf')
        for child in node.children:
            eval = alpha_beta_pruning(child, depth - 1, alpha, beta, True)
            min_eval = min(min_eval, eval)
            beta = min(beta, eval)
            if beta <= alpha:
                print(f"Pruned at MIN node with alpha={alpha}, beta={beta}")
                break
        node.value = min_eval
        return min_eval


def print_tree(node, level=0):
    print("  " * level *2 + f"Value of Node: {node.value}")
    for child in node.children:
        print_tree(child, level + 1)


if __name__ == "__main__":
    tree = Node(None, [
        Node(None, [
            Node(None,[Node(10),Node(9)]),
            Node(None, [Node(14),Node(18)])
        ]),
        Node(None, [
            Node(None, [Node(5),Node(4)]),
            Node(None, [Node(50),Node(3)])
        ])
    ])

    print("Game Tree Before Alpha-Beta Pruning:")
    print_tree(tree)
    final_value = alpha_beta_pruning(tree, depth=3, alpha=float('-inf'), beta=float('inf'),
maximizing_player=True)
    print("\nGame Tree After Alpha-Beta Pruning:")
    print_tree(tree)
    print("\nFinal Value at MAX node:", final_value)
```

Output Snapshot:

```
Game Tree Before Alpha-Beta Pruning:
Value of Node: None
    Value of Node: None
        Value of Node: None
            Value of Node: 10
            Value of Node: 9
        Value of Node: None
            Value of Node: 14
            Value of Node: 18
    Value of Node: None
        Value of Node: None
            Value of Node: 5
            Value of Node: 4
        Value of Node: None
            Value of Node: 50
            Value of Node: 3
Pruned at MAX node with alpha=14, beta=10
Pruned at MIN node with alpha=10, beta=5

Game Tree After Alpha-Beta Pruning:
Value of Node: 10
    Value of Node: 10
        Value of Node: 10
            Value of Node: 10
            Value of Node: 9
        Value of Node: 14
            Value of Node: 14
            Value of Node: 18
    Value of Node: 5
        Value of Node: 5
            Value of Node: 5
            Value of Node: 4
        Value of Node: None
            Value of Node: 50
            Value of Node: 3

Final Value at MAX node: 10
```