VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB REPORT
on

Artificial Intelligence (23CS5PCAIN)

Submitted by

Siri Sathish 1BM22CS280

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

B.M.S. COLLEGE OF ENGINEERING
(Autonomous Institution under VTU)
BENGALURU-560019
Sep-2024 to Jan-2025

B.M.S. College of Engineering,
Bull Temple Road, Bangalore 560019
(Affiliated To Visvesvaraya Technological University, Belgaum)
Department of Computer Science and Engineering

CERTIFICATE

This is to certify that the Lab work entitled "Artificial Intelligence (23CS5PCAIN)" carried out by **Siri Sathish 1BM22CS280,** 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.

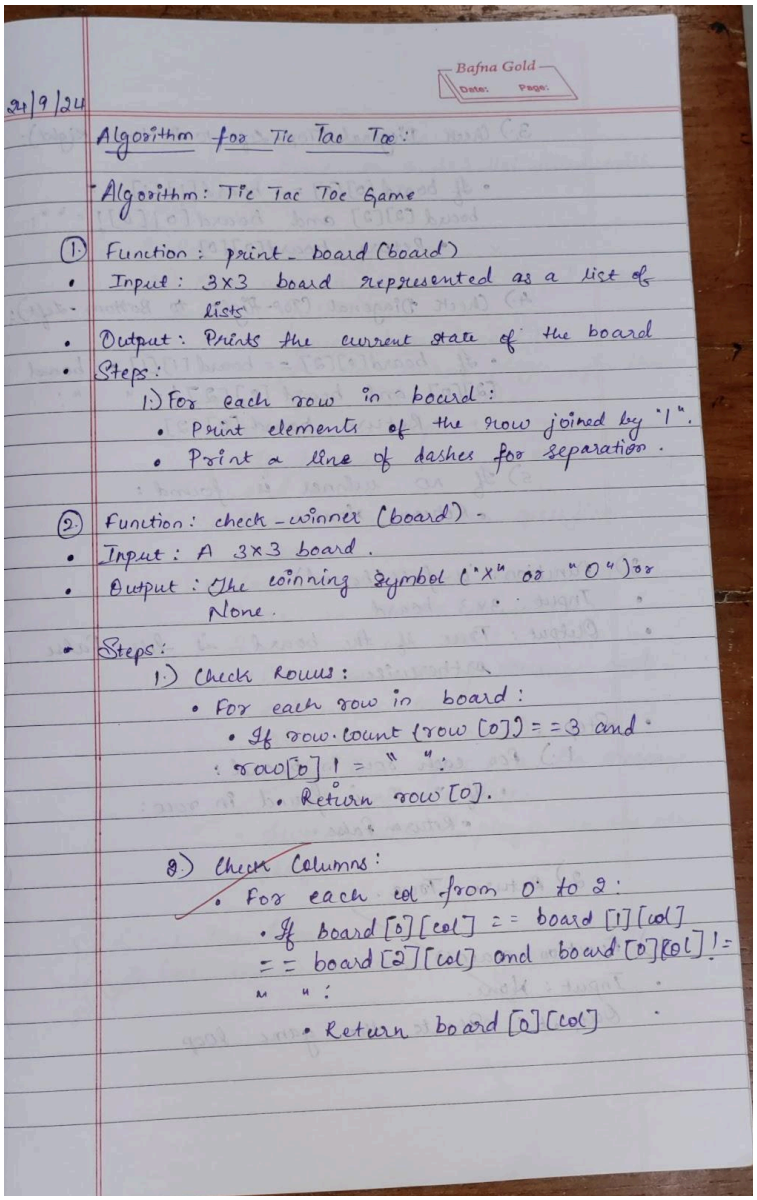
Sunayana S Assistant Professor Department of CSE, BMSCE	Dr. Jyothi S Nayak Professor & HOD Department of CSE, BMSCE

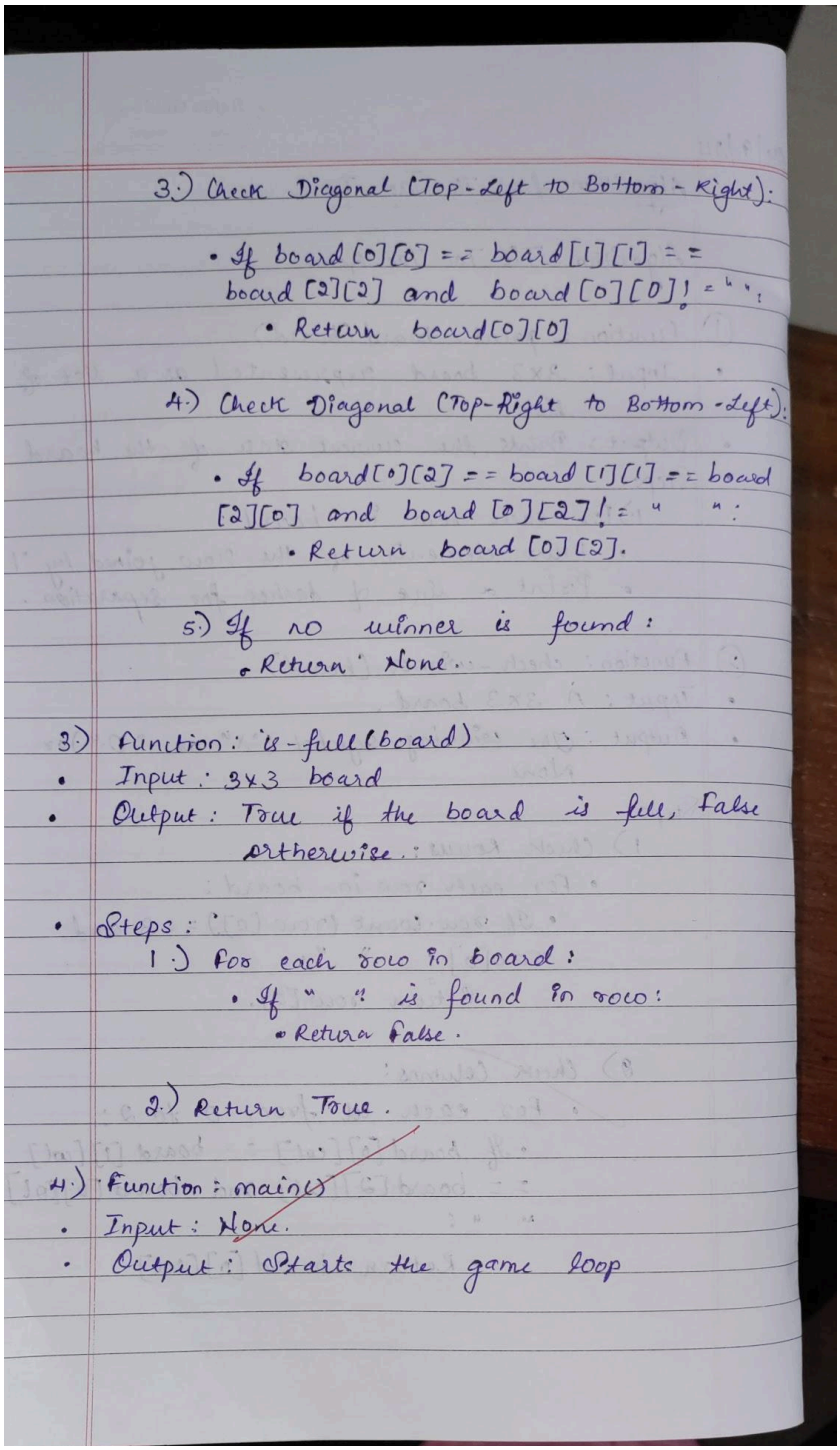
Index

Github Link: https://github.com/SiriSathish600/AI

Program 1:

- **Tic Tac Toe**
- **Vacuum World Agent**





3.) Check Diagonal (Top-Left to Bottom-Right):

- If board[0][0] == board[1][1] == board[2][2] and board[0][0] != " ":
  - Return board[0][0]

4.) Check Diagonal (Top-Right to Bottom-Left):

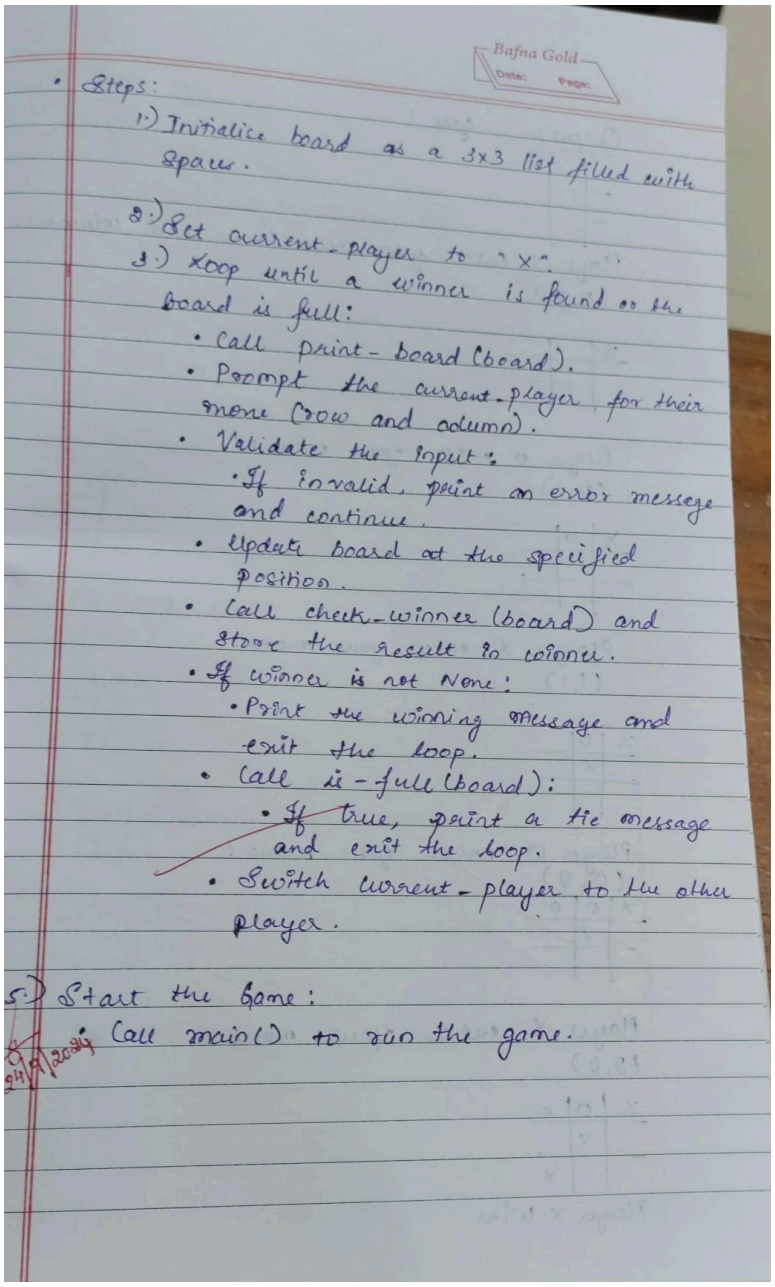
- If board[0][2] == board[1][1] == board[2][0] and board[0][2] != " ":
  - Return board[0][2].

5.) If no winner is found:
- Return None.

3.) Function: is_full(board)
- Input: 3x3 board
- Output: True if the board is full, False otherwise.
- Steps:
  1.) For each row in board:
    - If " " is found in row:
      - Return False.

  2.) Return True.

4.) Function: main()
- Input: None.
- Output: Starts the game loop


Bafna Gold
Date:
Page:
• Steps:
1.) Initialize board as a 3x3 list filled with spaces.
2.) Set current_player to "X".
3.) Loop until a winner is found or the board is full:
• Call print_board(board).
• Prompt the current_player for their move (row and column).
• Validate the input:
• If invalid, print an error message and continue.
• Update board at the specified position.
• Call check_winner(board) and store the result in winner.
• If winner is not None:
• Print the winning message and exit the loop.
• Call is_full(board):
• If true, print a tie message and exit the loop.
• Switch current_player to the other player.
5.) Start the Game:
• Call main() to run the game.

Tic Tac Toe game:

```
# Function to print the Tic Tac Toe board
def print_board(board):
    for row in board:
        print("|".join(row))
        print("-" * 5)

# Function to check if any player has won
def check_winner(board, player):
    # Check rows, columns, and diagonals
    for row in board:
        if all([spot == player for spot in row]):
            return True

    for col in range(3):
        if all([board[row][col] == player for row in range(3)]):
            return True

    if board[0][0] == board[1][1] == board[2][2] == player:
        return True

    if board[0][2] == board[1][1] == board[2][0] == player:
        return True

    return False

# Function to check if the board is full
def is_full(board):
    return all([all([spot != " " for spot in row]) for row in board])

# Main game loop
def tic_tac_toe():
    # Initialize the board
    board = [[" " for _ in range(3)] for _ in range(3)]
    current_player = "X"  # X goes first

    while True:
        print_board(board)
        print(f"Player {current_player}'s turn")

        # Get the player's move
        row = int(input("Enter row (0, 1, 2): "))
        col = int(input("Enter column (0, 1, 2): "))

        # Check if the move is valid
        if board[row][col] == " ":
```

```
            board[row][col] = current_player
        else:
            print("That spot is already taken! Try again.")
            continue

        # Check if the current player has won
        if check_winner(board, current_player):
            print_board(board)
            print(f"Player {current_player} wins!")
            break

        # Check if the game is a tie
        if is_full(board):
            print_board(board)
            print("It's a tie!")
            break

        # Switch player
        current_player = "O" if current_player == "X" else "X"

# Run the game
tic_tac_toe()
```

Output:

```
 | |
-----
 | |
-----
 | |
Player X's turn
Enter row (0, 1, 2): 0
Enter column (0, 1, 2): 0

X| |
-----
 | |
-----
 | |
Player O's turn
Enter row (0, 1, 2): 1
Enter column (0, 1, 2): 1

X| |
-----
 |O|
```

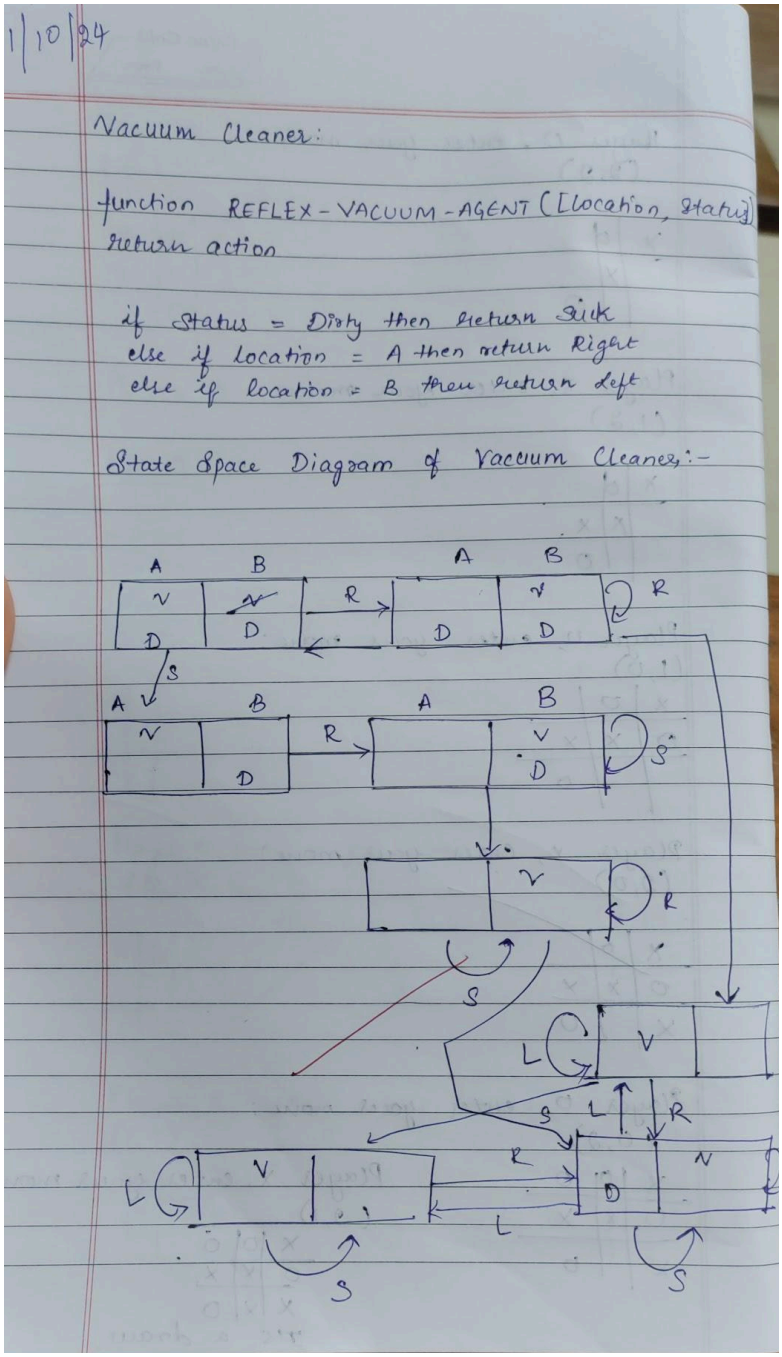
```
-----
 | |
Player X's turn
Enter row (0, 1, 2): 0
Enter column (0, 1, 2): 1

X|X|
-----
 |O|
-----
 | |
Player O's turn
Enter row (0, 1, 2): 0
Enter column (0, 1, 2): 2

X|X|O
-----
 |O|
-----
 | |
Player X's turn
Enter row (0, 1, 2): 2
Enter column (0, 1, 2): 0

X|X|O
-----
 |O|
X| |
Player O's turn
Enter row (0, 1, 2): 2
Enter column (0, 1, 2): 2

X|X|O
-----
 |O|
X| |O
Player X's turn
Enter row (0, 1, 2): 1
Enter column (0, 1, 2): 0

X|X|O
-----
X|O|
X| |O
Player O's turn
Enter row (0, 1, 2): 1
Enter column (0, 1, 2): 2
```

```
X|X|O
-----
X|O|O
X| |O
Player X's turn
Enter row (0, 1, 2): 2
Enter column (0, 1, 2): 1

X|X|O
-----
X|O|O
X|X|O
It's a tie!



 | |
-----
 | |
-----
 | |
Player X's turn
Enter row (0, 1, 2): 0
Enter column (0, 1, 2): 0

X| |
-----
 | |
-----
 | |
Player O's turn
Enter row (0, 1, 2): 1
Enter column (0, 1, 2): 1

X| |
-----
 |O|
-----
 | |
Player X's turn
Enter row (0, 1, 2): 0
Enter column (0, 1, 2): 1

X|X|
-----
 |O|
```

```
-----
 | |
Player O's turn
Enter row (0, 1, 2): 2
Enter column (0, 1, 2): 1

X|X|
-----
 |O|
 |O|
Player X's turn
Enter row (0, 1, 2): 0
Enter column (0, 1, 2): 2

X|X|X
-----
 |O|
 |O|
Player X wins!



 | |
-----
 | |
-----
 | |
Player X's turn
Enter row (0, 1, 2): 0
Enter column (0, 1, 2): 0

X| |
-----
 | |
-----
 | |
Player O's turn
Enter row (0, 1, 2): 0
Enter column (0, 1, 2): 1

X|O|
-----
 | |
-----
 | |
Player X's turn
Enter row (0, 1, 2): 1
```

```
Enter column (0, 1, 2): 0

X|O|
-----
X| |
-----
 | |
Player O's turn
Enter row (0, 1, 2): 1
Enter column (0, 1, 2): 1

X|O|
-----
X|O|
-----
 | |
Player X's turn
Enter row (0, 1, 2): 2
Enter column (0, 1, 2): 0

X|O|
-----
X|O|
X| |
Player O's turn
Enter row (0, 1, 2): 2
Enter column (0, 1, 2): 1

X|O|
-----
X|O|
X|O|
Player O wins!
```

Vacuum Cleaner Agent:



Parameters:

Location
Status of current & next room
Cost

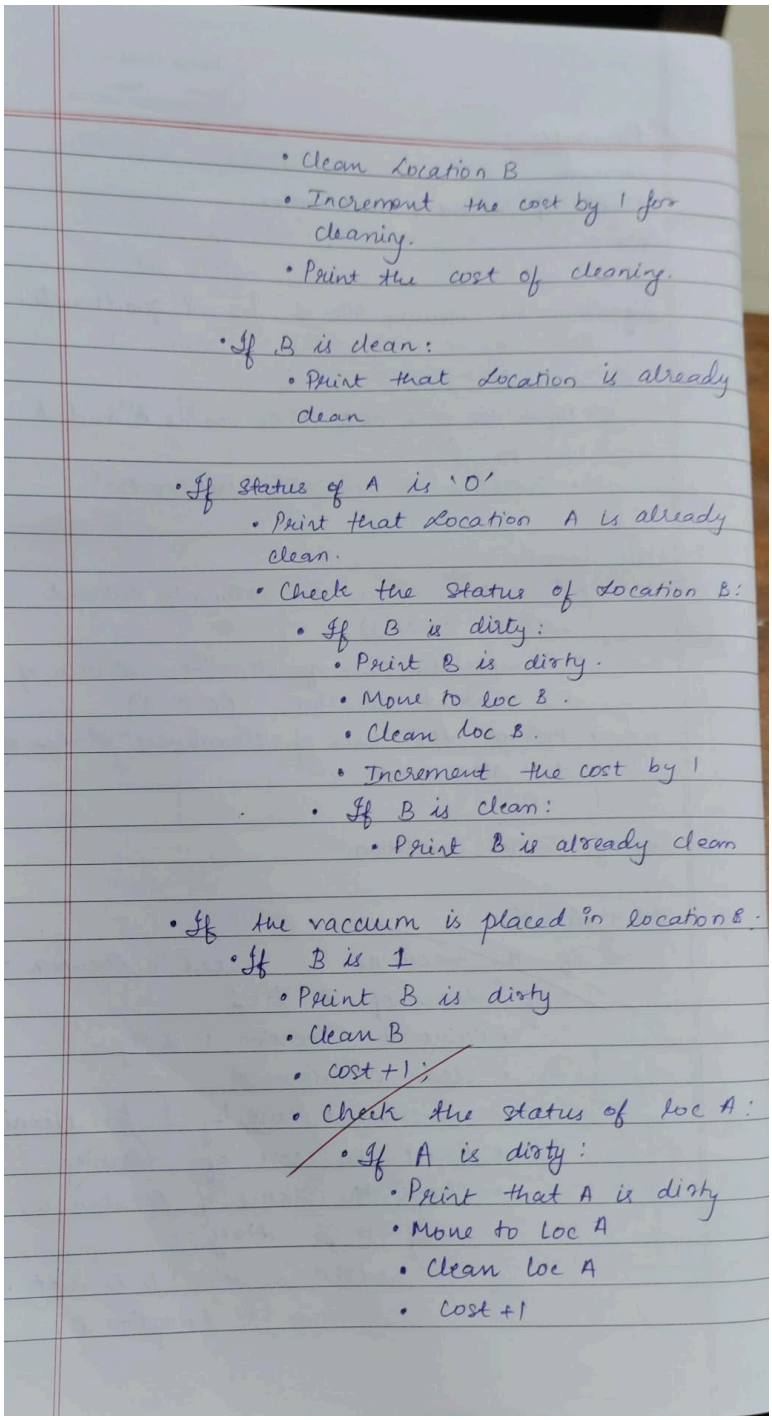
Algorithm for Vacuum World for 2 quadrants:

1. Initialization:
   - Define the goal state with rooms A and B set to '0'
   - Initialize a variable cost to 0.

2. User input:
   - Prompt the user for vacuum's current location (A or B)
   - Prompt the user of cleanliness status of the current location. (0 or 1)
   - Prompt the user of cleanliness status of the other room

3. Display Initial Condition.

4. Check Location:
   - If the vacuum is placed in Location 'A':
     - If Status of A is 1
       - Print the location is dirty.
       - Clean Location A
       - Increment cost by 1 for cleaning.
       - Print the cost of cleaning.
       - Check the status of Location B.
         - If B is dirty:
           - Print Location B is dirty.
           - Move to Location B.

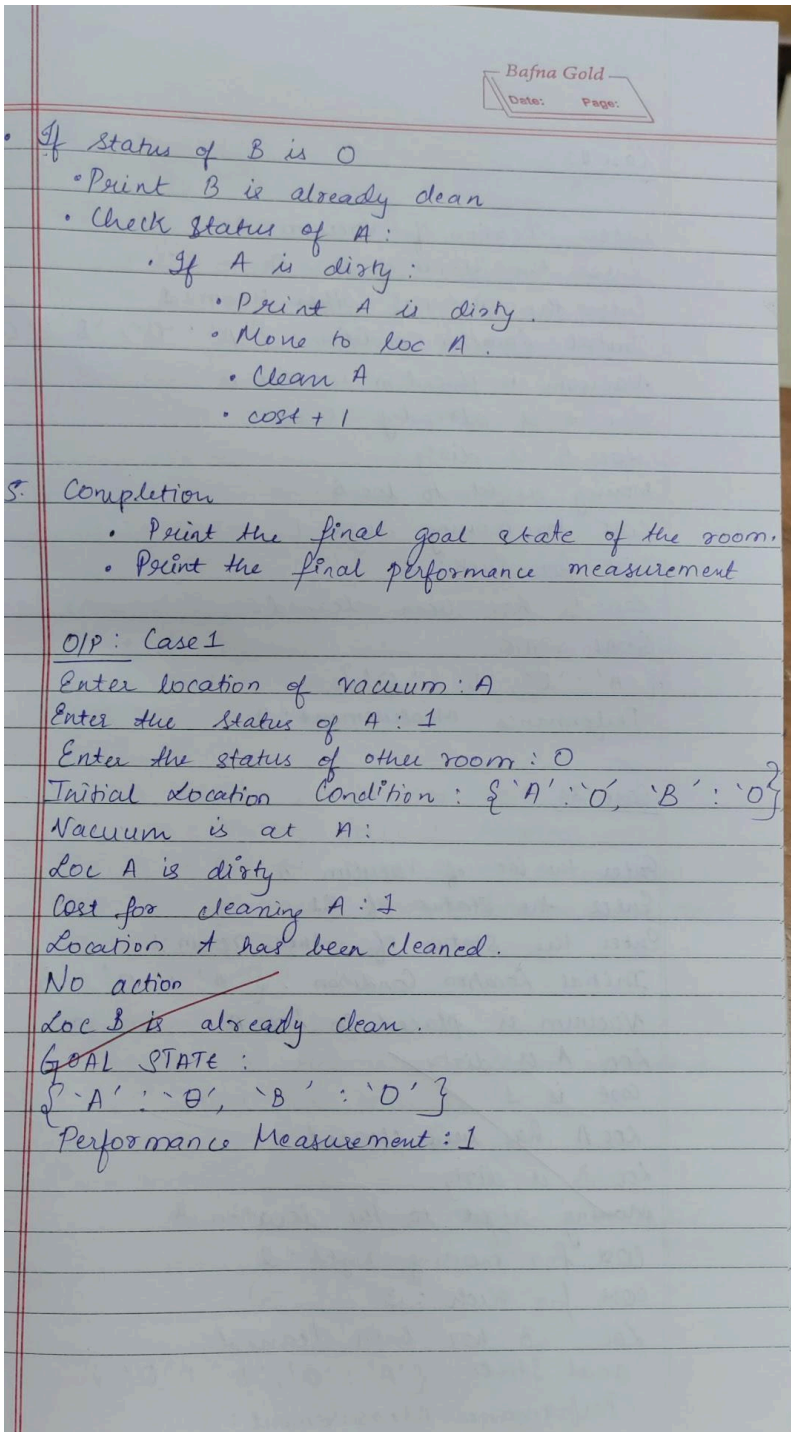


- Clean Location B
- Increment the cost by 1 for cleaning.
- Print the cost of cleaning.

- If B is clean:
  - Print that Location is already clean

- If Status of A is '0'
  - Print that Location A is already clean.
  - Check the Status of Location B:
    - If B is dirty:
      - Print B is dirty.
      - Move to loc B.
      - Clean loc B.
      - Increment the cost by 1
    - If B is clean:
      - Print B is already clean

- If the vacuum is placed in location B:
  - If B is 1
    - Print B is dirty
    - Clean B
    - cost +1;
    - Check the status of loc A:
      - If A is dirty:
        - Print that A is dirty
        - Move to loc A
        - Clean loc A
        - Cost +1

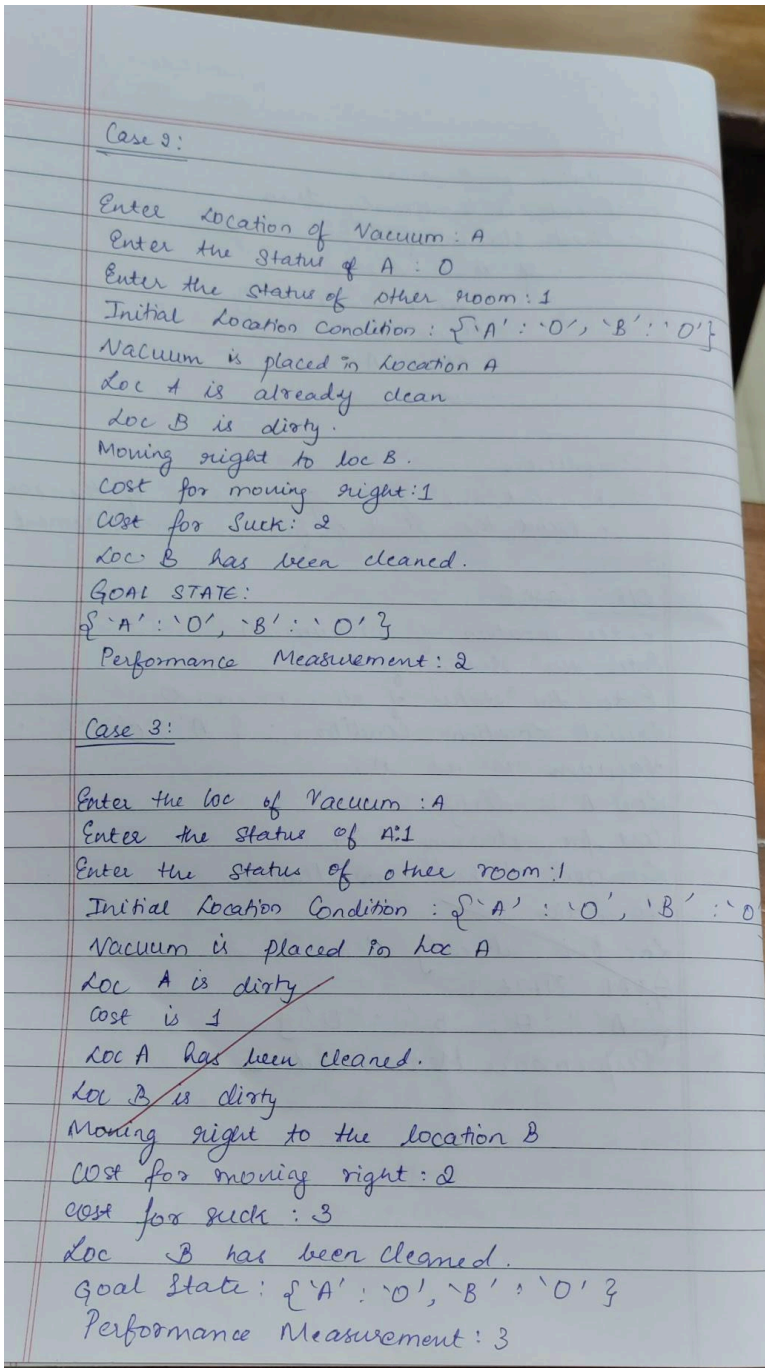


- If Status of B is 0
  - Print B is already clean
  - Check Status of A:
    - If A is dirty:
      - Print A is dirty.
      - Move to Loc A.
      - Clean A
      - cost + 1

5. Completion
   - Print the final goal state of the room.
   - Print the final performance measurement

O/P: Case 1

Enter location of vacuum: A
Enter the Status of A: 1
Enter the status of other room: 0
Initial Location Condition: {'A': '0', 'B': '0'}
Vacuum is at A:
Loc A is dirty
Cost for cleaning A: 1
Location A has been cleaned.
No action
Loc B is already clean.
GOAL STATE:
{'A': '0', 'B': '0'}
Performance Measurement: 1



Case 2:

Enter Location of Vacuum : A
Enter the Status of A : 0
Enter the status of other room : 1
Initial Location condition : {'A' : '0', 'B' : '0'}
Vacuum is placed in Location A
Loc A is already clean
Loc B is dirty.
Moving right to loc B.
Cost for moving right : 1
Cost for Suck : 2
Loc B has been cleaned.
GOAL STATE:
{'A' : '0', 'B' : '0'}
Performance Measurement : 2

Case 3:

Enter the loc of Vacuum : A
Enter the Status of A : 1
Enter the Status of other room : 1
Initial Location Condition : {'A' : '0', 'B' : '0
Vacuum is placed to loc A
Loc A is dirty
Cost is 1
Loc A has been cleaned.
Loc B is dirty
Moving right to the location B
Cost for moving right : 2
Cost for suck : 3
Loc B has been cleaned.
Goal State : {'A' : '0', 'B' : '0'}
Performance Measurement : 3

2 Quadrants:

```
#INSTRUCTIONS
#Enter LOCATION A/B in captial letters
#Enter Status O/1 accordingly where 0 means CLEAN and 1 means DIRTY

def vacuum_world():
      # initializing goal_state
      # 0 indicates Clean and 1 indicates Dirty
   goal_state = {'A': '0', 'B': '0'}
   cost = 0

   location_input = input("Enter Location of Vacuum:") #user_input of location vacuum is placed
   status_input = input("Enter status of: " + location_input) #user_input if location is dirty or clean
   status_input_complement = input("Enter status of other room:")
   print("Initial Location Condition :" + str(goal_state))

   if location_input == 'A':
      # Location A is Dirty.
      print("Vacuum is placed in Location A")
      if status_input == '1':
         print("Location A is Dirty.")
         # suck the dirt  and mark it as clean
         goal_state['A'] = '0'
         cost += 1                    #cost for suck
         print("Cost for CLEANING A " + str(cost))
         print("Location A has been Cleaned.")

         if status_input_complement == '1':
            # if B is Dirty
            print("Location B is Dirty.")
            print("Moving right to the Location B. ")
            cost += 1                     #cost for moving right
            print("COST for moving RIGHT" + str(cost))
            # suck the dirt and mark it as clean
            goal_state['B'] = '0'
            cost += 1                     #cost for suck
            print("COST for SUCK " + str(cost))
            print("Location B has been Cleaned. ")
         else:
            print("No action" + str(cost))
            # suck and mark clean
            print("Location B is already clean.")

      if status_input == '0':
         print("Location A is already clean ")
         if status_input_complement == '1':# if B is Dirty
```

```
            print("Location B is Dirty.")
            print("Moving RIGHT to the Location B. ")
            cost += 1                     #cost for moving right
            print("COST for moving RIGHT " + str(cost))
            # suck the dirt and mark it as clean
            goal_state['B'] = '0'
            cost += 1                     #cost for suck
            print("Cost for SUCK" + str(cost))
            print("Location B has been Cleaned. ")
        else:
            print("No action " + str(cost))
            print(cost)
            # suck and mark clean
            print("Location B is already clean.")

else:
    print("Vacuum is placed in location B")
    # Location B is Dirty.
    if status_input == '1':
        print("Location B is Dirty.")
        # suck the dirt  and mark it as clean
        goal_state['B'] = '0'
        cost += 1  # cost for suck
        print("COST for CLEANING " + str(cost))
        print("Location B has been Cleaned.")

        if status_input_complement == '1':
            # if A is Dirty
            print("Location A is Dirty.")
            print("Moving LEFT to the Location A. ")
            cost += 1  # cost for moving right
            print("COST for moving LEFT" + str(cost))
            # suck the dirt and mark it as clean
            goal_state['A'] = '0'
            cost += 1  # cost for suck
            print("COST for SUCK " + str(cost))
            print("Location A has been Cleaned.")

    else:
        print(cost)
        # suck and mark clean
        print("Location B is already clean.")

        if status_input_complement == '1':  # if A is Dirty
            print("Location A is Dirty.")
            print("Moving LEFT to the Location A. ")
            cost += 1  # cost for moving right
```

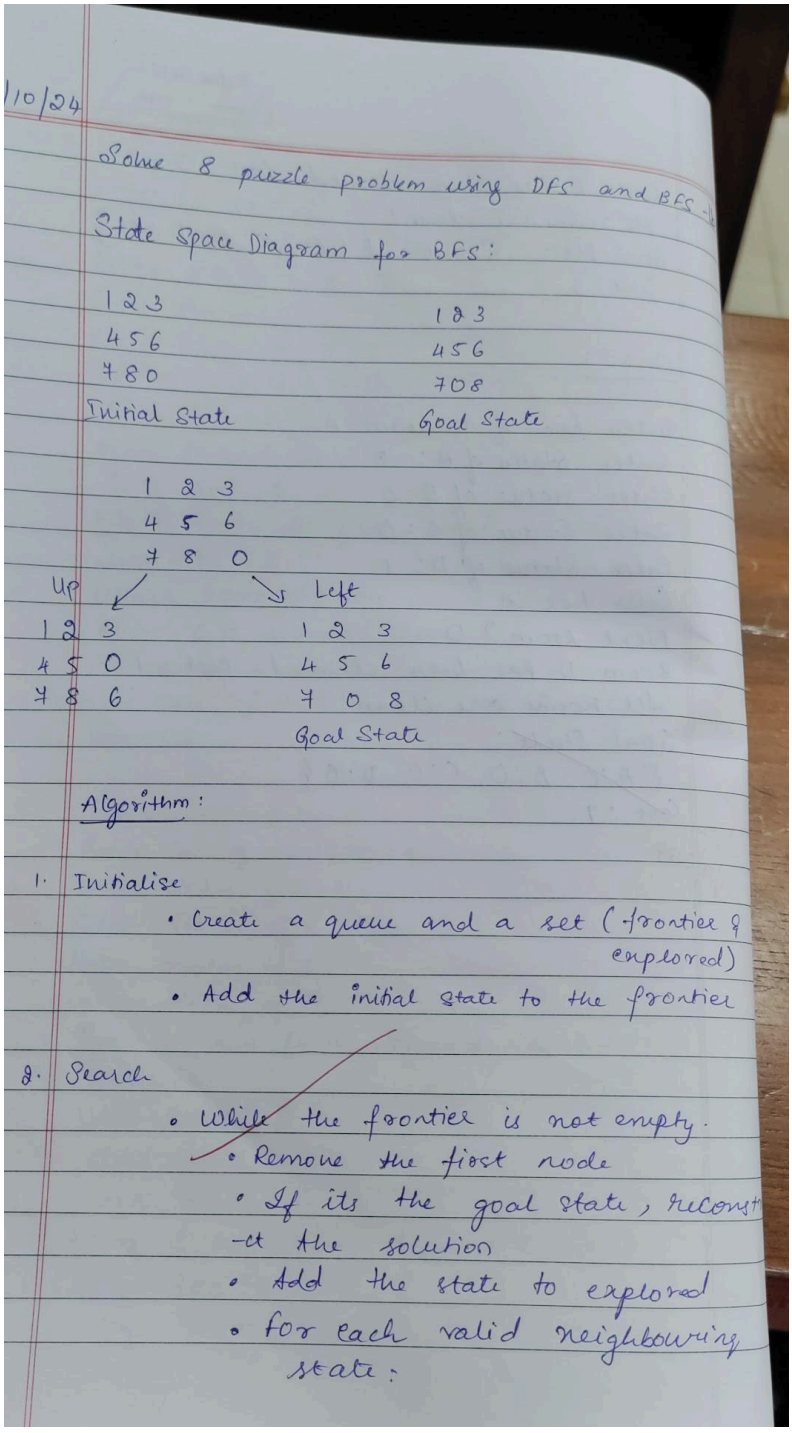
```
            print("COST for moving LEFT " + str(cost))
            # suck the dirt and mark it as clean
            goal_state['A'] = '0'
            cost += 1  # cost for suck
            print("Cost for SUCK " + str(cost))
            print("Location A has been Cleaned. ")
        else:
            print("No action " + str(cost))
            # suck and mark clean
            print("Location A is already clean.")

    # done cleaning
    print("GOAL STATE: ")
    print(goal_state)
    print("Performance Measurement: " + str(cost))


vacuum_world()
```

Output:

```
Enter Location of Vacuum:A
Enter status of: A1
Enter status of other room:1
Initial Location Condition :{'A': '0', 'B': '0'}
Vacuum is placed in Location A
Location A is Dirty.
Cost for CLEANING A 1
Location A has been Cleaned.
Location B is Dirty.
Moving right to the Location B.
COST for moving RIGHT2
COST for SUCK 3
Location B has been Cleaned.
GOAL STATE:
{'A': '0', 'B': '0'}
Performance Measurement: 3
```

Program 2:

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



10/24

Solve 8 puzzle problem using DFS and BFS.

State Space Diagram for BFS:

1 2 3	1 2 3
4 5 6	4 5 6
7 8 0	7 0 8
Initial State	Goal State

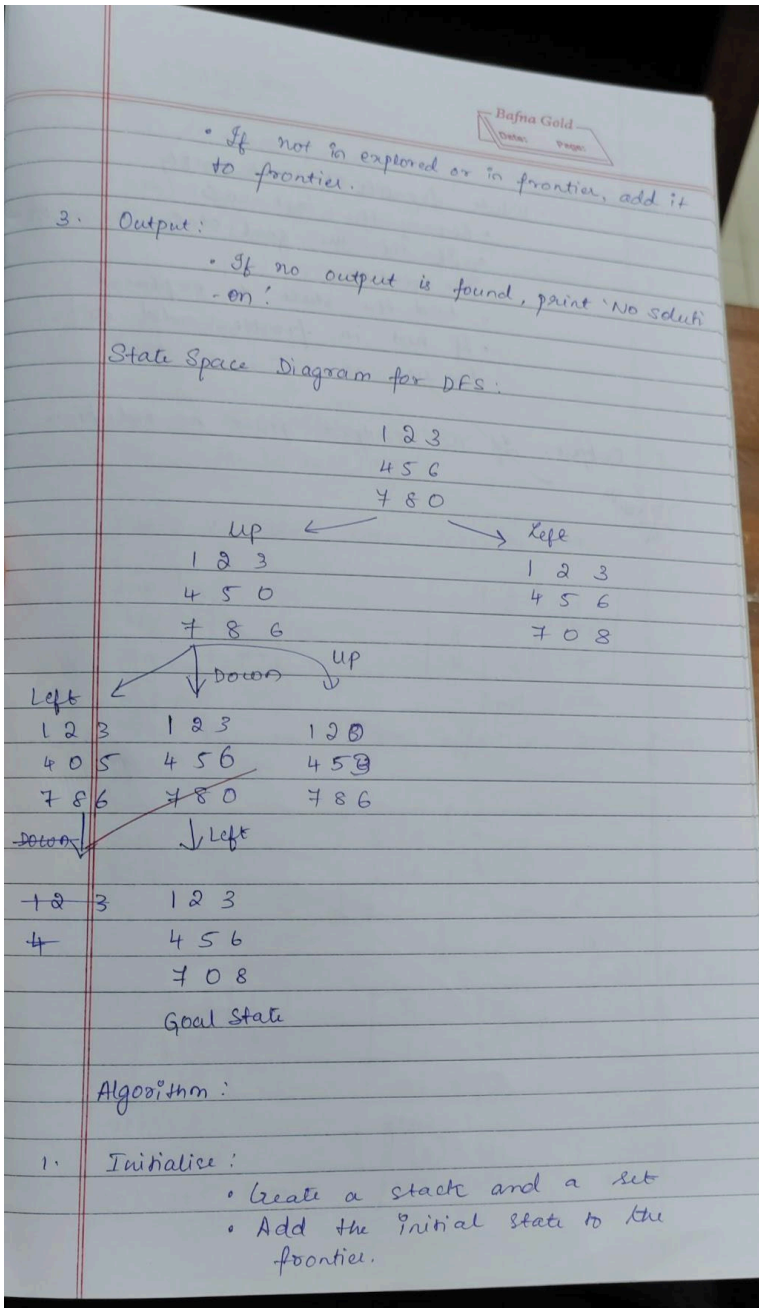
1 2 3
4 5 6
7 8 0

Up ↙ ↘ Left

1 2 3	1 2 3
4 5 0	4 5 6
7 8 6	7 0 8
	Goal State

Algorithm:

1. Initialise
   - Create a queue and a set (frontier & explored)
   - Add the initial state to the frontier
2. Search
   - While the frontier is not empty.
     - Remove the first node
     - If its the goal state, reconst-ct the solution
     - Add the state to explored
     - for each valid neighbouring state:



- If not in explored or in frontier, add it to frontier.

3. Output:
   - If no output is found, print 'No solution'.

State Space Diagram for DFS:

1 2 3
4 5 6
7 8 0

up ← → Left

up: 1 2 3 / 4 5 0 / 7 8 6

Left: 1 2 3 / 4 5 6 / 7 8 0

up: 1 2 0 / 4 5 3 / 7 8 6

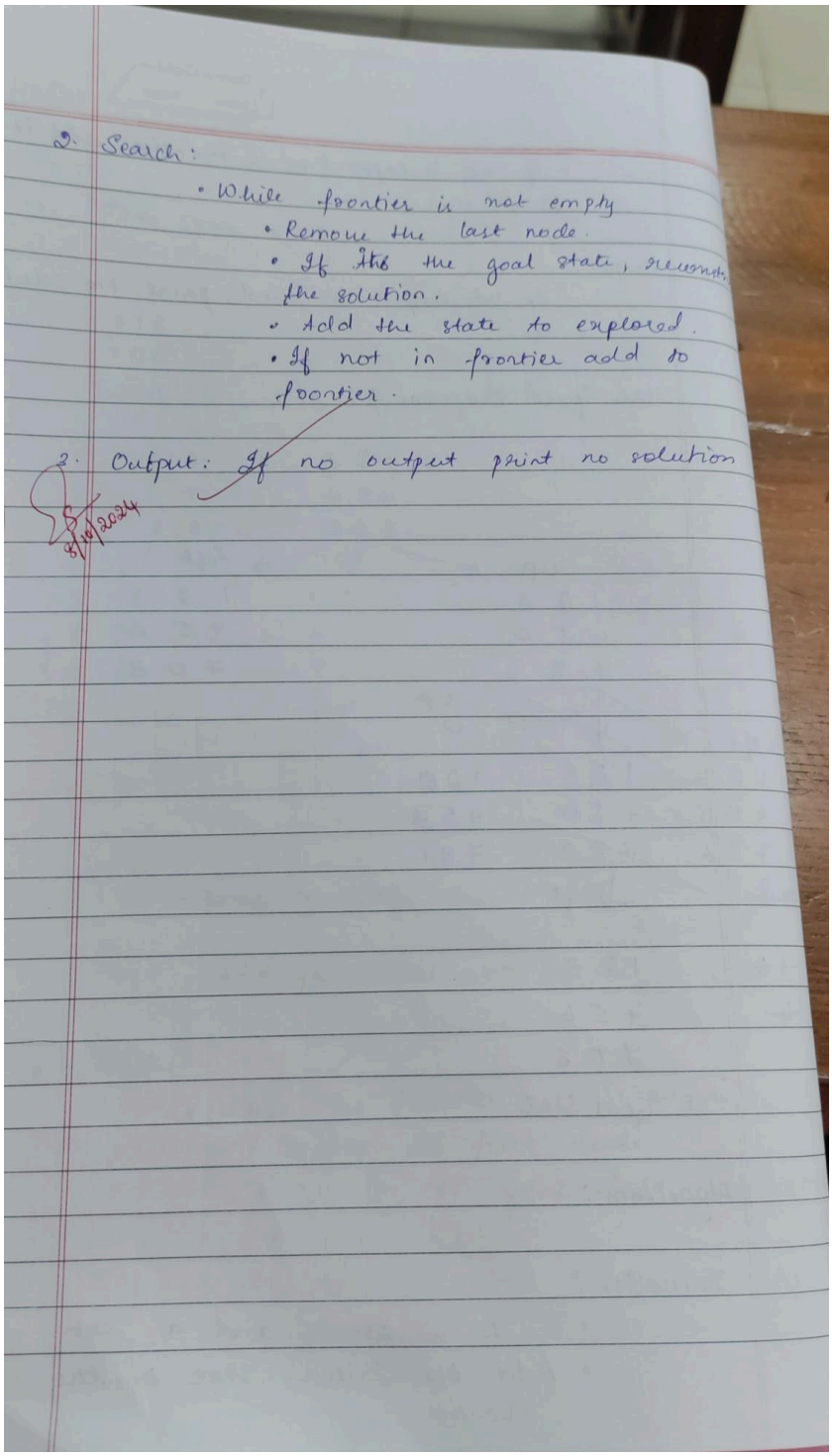
Down: 1 2 3 / 4 5 6 / 7 0 8

Left: 1 2 3 / 4 5 6 / 7 8 0

Goal State

Algorithm:

1. Initialize:
   - Create a stack and a set
   - Add the initial state to the frontier.


2. Search:
• While frontier is not empty
• Remove the last node.
• If its the goal state, reconstruct the solution.
• Add the state to explored.
• If not in frontier add to frontier.
3. Output: If no output print no solution
8/9/2024

DFS:

```
import numpy as np


class Node:
    def __init__(self, state, parent, action):
        self.state = state
        self.parent = parent
        self.action = action


class StackFrontier:
    def __init__(self):
        self.frontier = []

    def add(self, node):
        self.frontier.append(node)

    def contains_state(self, state):
        return any((node.state[0] == state[0]).all() for node in self.frontier)

    def empty(self):
        return len(self.frontier) == 0

    def remove(self):
        if self.empty():
            raise Exception("Empty Frontier")
        else:
            node = self.frontier[-1]  # Remove from the end (LIFO)
            self.frontier = self.frontier[:-1]
            return node


class Puzzle:
    def __init__(self, start, startIndex, goal, goalIndex):
        self.start = [start, startIndex]
        self.goal = [goal, goalIndex]
        self.solution = None

    def neighbors(self, state):
        mat, (row, col) = state
        results = []

        if row > 0:  # Move up
            mat1 = np.copy(mat)
            mat1[row][col] = mat1[row - 1][col]
            mat1[row - 1][col] = 0
```

```
            results.append(('up', [mat1, (row - 1, col)]))
        if col > 0:  # Move left
            mat1 = np.copy(mat)
            mat1[row][col] = mat1[row][col - 1]
            mat1[row][col - 1] = 0
            results.append(('left', [mat1, (row, col - 1)]))
        if row < 2:  # Move down
            mat1 = np.copy(mat)
            mat1[row][col] = mat1[row + 1][col]
            mat1[row + 1][col] = 0
            results.append(('down', [mat1, (row + 1, col)]))
        if col < 2:  # Move right
            mat1 = np.copy(mat)
            mat1[row][col] = mat1[row][col + 1]
            mat1[row][col + 1] = 0
            results.append(('right', [mat1, (row, col + 1)]))

        return results

    def print_solution(self):
        if self.solution is not None:
            print("Start State:\n", self.start[0], "\n")
            print("Goal State:\n", self.goal[0], "\n")
            print("\nStates Explored: ", self.num_explored, "\n")
            for idx, state in enumerate(self.explored):
                print(f"{idx + 1}. \n{state[0]}\n")
            print("Solution Steps:")
            for action, cell in zip(self.solution[0], self.solution[1]):
                print(f"Move {action}:\n{cell[0]}\n")
            print("Goal Reached!!")
        else:
            print("No solution found.")

    def does_not_contain_state(self, state):
        for st in self.explored:
            if (st[0] == state[0]).all():
                return False
        return True

    def solve(self):
        self.num_explored = 0
        self.explored = []  # Initialize explored states

        start = Node(state=self.start, parent=None, action=None)
        frontier = StackFrontier()  # Use StackFrontier for DFS
        frontier.add(start)
```

```
        while True:
            if frontier.empty():
                raise Exception("No solution")

            node = frontier.remove()
            self.num_explored += 1
            self.explored.append(node.state)  # Add current state to explored

            if (node.state[0] == self.goal[0]).all():
                actions = []
                cells = []
                while node.parent is not None:
                    actions.append(node.action)
                    cells.append(node.state)
                    node = node.parent
                actions.reverse()
                cells.reverse()
                self.solution = (actions, cells)
                return

            for action, state in self.neighbors(node.state):
                if not frontier.contains_state(state) and self.does_not_contain_state(state):
                    child = Node(state=state, parent=node, action=action)
                    frontier.add(child)


def input_puzzle():
    print("Enter the initial state (3x3 matrix, use spaces to separate numbers):")
    start = np.array([list(map(int, input().split())) for _ in range(3)])

    print("Enter the goal state (3x3 matrix, use spaces to separate numbers):")
    goal = np.array([list(map(int, input().split())) for _ in range(3)])

    startIndex = tuple(map(int, np.argwhere(start == 0)[0]))  # Find position of 0 in start
    goalIndex = tuple(map(int, np.argwhere(goal == 0)[0]))  # Find position of 0 in goal

    return start, startIndex, goal, goalIndex


if __name__ == "__main__":
    start, startIndex, goal, goalIndex = input_puzzle()
    p = Puzzle(start, startIndex, goal, goalIndex)
    p.solve()
    p.print_solution()
```

Output:

Enter the initial state (3x3 matrix, use spaces to separate numbers):
1 2 3
4 5 6
7 8 0
Enter the goal state (3x3 matrix, use spaces to separate numbers):
1 2 3
4 5 6
7 0 8
Start State:
[[1 2 3]
[4 5 6]
[7 8 0]]

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

States Explored: 2

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

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

Solution Steps:
Move left:
[[1 2 3]
[4 5 6]
[7 0 8]]

Goal Reached!!

BFS:

```
import sys
import numpy as np


class Node:
    def __init__(self, state, parent, action):
        self.state = state
        self.parent = parent
        self.action = action


class QueueFrontier:
    def __init__(self):
        self.frontier = []

    def add(self, node):
        self.frontier.append(node)

    def contains_state(self, state):
        return any((node.state[0] == state[0]).all() for node in self.frontier)

    def empty(self):
        return len(self.frontier) == 0

    def remove(self):
        if self.empty():
            raise Exception("Empty Frontier")
        else:
            node = self.frontier[0]  # Remove from front
            self.frontier = self.frontier[1:]
            return node


class Puzzle:
    def __init__(self, start, startIndex, goal, goalIndex):
        self.start = [start, startIndex]
        self.goal = [goal, goalIndex]
        self.solution = None

    def neighbors(self, state):
        mat, (row, col) = state
        results = []

        if row > 0:  # Move up
            mat1 = np.copy(mat)
```

```
            mat1[row][col] = mat1[row - 1][col]
            mat1[row - 1][col] = 0
            results.append(('up', [mat1, (row - 1, col)]))
        if col > 0:  # Move left
            mat1 = np.copy(mat)
            mat1[row][col] = mat1[row][col - 1]
            mat1[row][col - 1] = 0
            results.append(('left', [mat1, (row, col - 1)]))
        if row < 2:  # Move down
            mat1 = np.copy(mat)
            mat1[row][col] = mat1[row + 1][col]
            mat1[row + 1][col] = 0
            results.append(('down', [mat1, (row + 1, col)]))
        if col < 2:  # Move right
            mat1 = np.copy(mat)
            mat1[row][col] = mat1[row][col + 1]
            mat1[row][col + 1] = 0
            results.append(('right', [mat1, (row, col + 1)]))

        return results

    def print_explored_states(self):
        print("All Explored States:")
        for idx, state in enumerate(self.explored):
            print(f"{idx + 1}. \n{state[0]}\n")

    def print_solution(self):
        if self.solution is not None:
            print("Solution Steps:")
            for action, cell in zip(self.solution[0], self.solution[1]):
                print(f"Move {action}:")
                print(f"{cell[0]}\n")
            print("Goal Reached!!")
        else:
            print("No solution found.")

    def does_not_contain_state(self, state):
        for st in self.explored:
            if (st[0] == state[0]).all():
                return False
        return True

    def solve(self):
        self.num_explored = 0

        start = Node(state=self.start, parent=None, action=None)
        frontier = QueueFrontier()
```

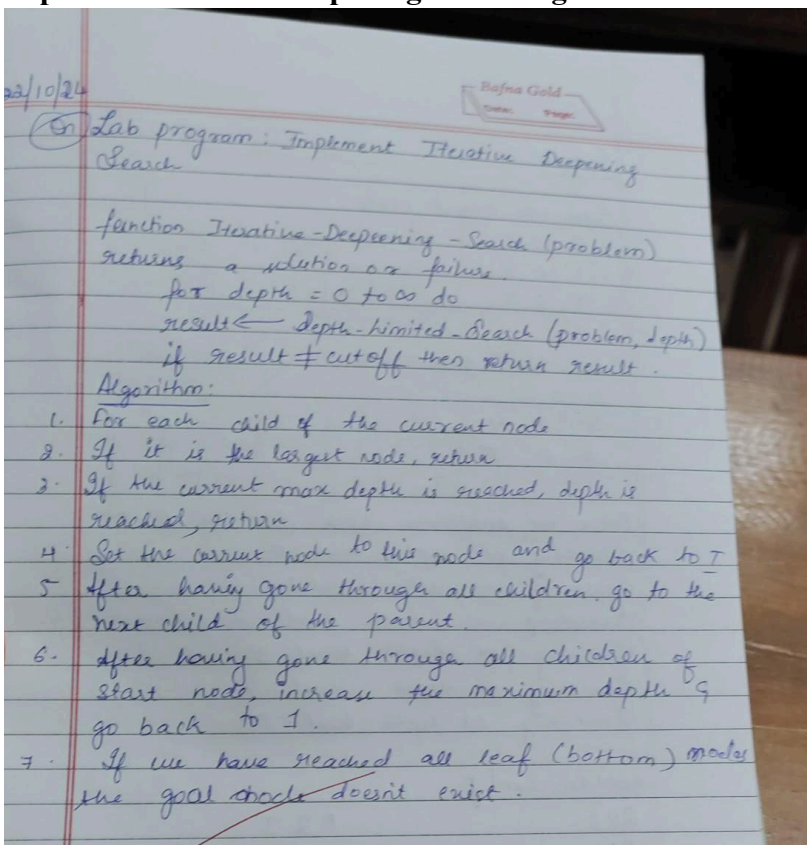
```
        frontier.add(start)

        self.explored = []

        while True:
            if frontier.empty():
                raise Exception("No solution")

            node = frontier.remove()
            self.num_explored += 1

            if (node.state[0] == self.goal[0]).all():
                actions = []
                cells = []
                while node.parent is not None:
                    actions.append(node.action)
                    cells.append(node.state)
                    node = node.parent
                actions.reverse()
                cells.reverse()
                self.solution = (actions, cells)
                return

            self.explored.append(node.state)

            for action, state in self.neighbors(node.state):
                if not frontier.contains_state(state) and self.does_not_contain_state(state):
                    child = Node(state=state, parent=node, action=action)
                    frontier.add(child)


def input_puzzle():
    print("Enter the initial state (3x3 matrix, use spaces to separate numbers):")
    start = np.array([list(map(int, input().split())) for _ in range(3)])

    print("Enter the goal state (3x3 matrix, use spaces to separate numbers):")
    goal = np.array([list(map(int, input().split())) for _ in range(3)])

    startIndex = tuple(map(int, np.argwhere(start == 0)[0]))  # Find position of 0 in start
    goalIndex = tuple(map(int, np.argwhere(goal == 0)[0]))  # Find position of 0 in goal

    return start, startIndex, goal, goalIndex


if __name__ == "__main__":
    start, startIndex, goal, goalIndex = input_puzzle()
    p = Puzzle(start, startIndex, goal, goalIndex)
```

```
    p.solve()
    p.print_explored_states()
    p.print_solution()
```

Output:

```
Enter the initial state (3x3 matrix, use spaces to separate numbers):
1 2 3
4 5 6
7 8 0
Enter the goal state (3x3 matrix, use spaces to separate numbers):
1 2 3
4 5 6
7 0 8
All Explored States:
1.
[[1 2 3]
 [4 5 6]
 [7 8 0]]

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

Solution Steps:
Move left:
[[1 2 3]
 [4 5 6]
 [7 0 8]]

Goal Reached!!
```

Implement Iterative deepening search algorithm:



22/10/24

6) Lab program: Implement Iterative Deepening Search

function Iterative-Deepening-Search (problem)
returns a solution or failure.
for depth = 0 to ∞ do
result ← Depth-limited-Search (problem, depth)
if result ≠ cutoff then return result.

Algorithm:

1. For each child of the current node
2. If it is the largest node, return
3. If the current max depth is reached, depth is reached, return
4. Set the current node to this node and go back to 1
5. After having gone through all children, go to the next child of the parent.
6. After having gone through all children of start node, increase the maximum depth by 1 go back to 1.
7. If we have reached all leaf (bottom) nodes the goal node doesn't exist.

```
from copy import deepcopy

DIRECTIONS = [(-1, 0), (1, 0), (0, -1), (0, 1)]

class PuzzleState:
    def __init__(self, board, parent=None, move=""):
        self.board = board
        self.parent = parent
        self.move = move

    def get_blank_position(self): # Indented this line to align with the class definition
        for i in range(3):
            for j in range(3):
                if self.board[i][j] == 0:
                    return i, j
```

```
    def generate_successors(self): # Indented this line to align with the class definition
        successors = []
        x, y = self.get_blank_position()

        for dx, dy in DIRECTIONS:
            new_x, new_y = x + dx, y + dy

            if 0 <= new_x < 3 and 0 <= new_y < 3:
                new_board = deepcopy(self.board)
                new_board[x][y], new_board[new_x][new_y] = new_board[new_x][new_y],
new_board[x][y]
                successors.append(PuzzleState(new_board, parent=self))

        return successors

    def is_goal(self, goal_state): # Indented this line to align with the class definition
        return self.board == goal_state

    def __str__(self): # Indented this line to align with the class definition
        return "\n".join([" ".join(map(str, row)) for row in self.board])

def depth_limited_search(current_state, goal_state, depth):
    if depth == 0 and current_state.is_goal(goal_state):
        return current_state

    if depth > 0:
        for successor in current_state.generate_successors():
            found = depth_limited_search(successor, goal_state, depth - 1)
            if found:
                return found
    return None

def iterative_deepening_search(start_state, goal_state):
    depth = 0
    while True:
        print(f"\nSearching at depth level: {depth}")
        result = depth_limited_search(start_state, goal_state, depth)
        if result:
            return result
        depth += 1
def get_user_input():
    print("Enter the start state (use 0 for the blank):")
    start_state = []
    for _ in range(3):
        row = list(map(int, input().split()))
        start_state.append(row)
```

```
    print("Enter the goal state (use 0 for the blank):")
    goal_state = []
    for _ in range(3):
        row = list(map(int, input().split()))
        goal_state.append(row)

    return start_state, goal_state

def main():
    start_board, goal_board = get_user_input()
    start_state = PuzzleState(start_board)
    goal_state = goal_board

    result = iterative_deepening_search(start_state, goal_state)


    if result:
        print("\nGoal reached!")
        path = []
        while result:
            path.append(result)
            result = result.parent
        path.reverse()
        for state in path:
            print(state, "\n")
    else:
        print("Goal state not found.")

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

Output:

```
Enter the start state (use 0 for the blank):
1 2 3
4 0 5
6 7 8
Enter the goal state (use 0 for the blank):
1 2 0
3 4 5
6 7 8



Searching at depth level: 0

Searching at depth level: 1
```

Searching at depth level: 2

Searching at depth level: 3

Searching at depth level: 4

Searching at depth level: 5

Searching at depth level: 6

Searching at depth level: 7

Searching at depth level: 8

Searching at depth level: 9

Searching at depth level: 10

Searching at depth level: 11

Searching at depth level: 12

Goal reached!
1 2 3
4 0 5
6 7 8

1 2 3
0 4 5
6 7 8

0 2 3
1 4 5
6 7 8

2 0 3
1 4 5
6 7 8

2 3 0
1 4 5
6 7 8

2 3 5
1 4 0
6 7 8

2 3 5
1 0 4
6 7 8

2 0 5
1 3 4
6 7 8

0 2 5
1 3 4
6 7 8

1 2 5
0 3 4
6 7 8

1 2 5
3 0 4
6 7 8

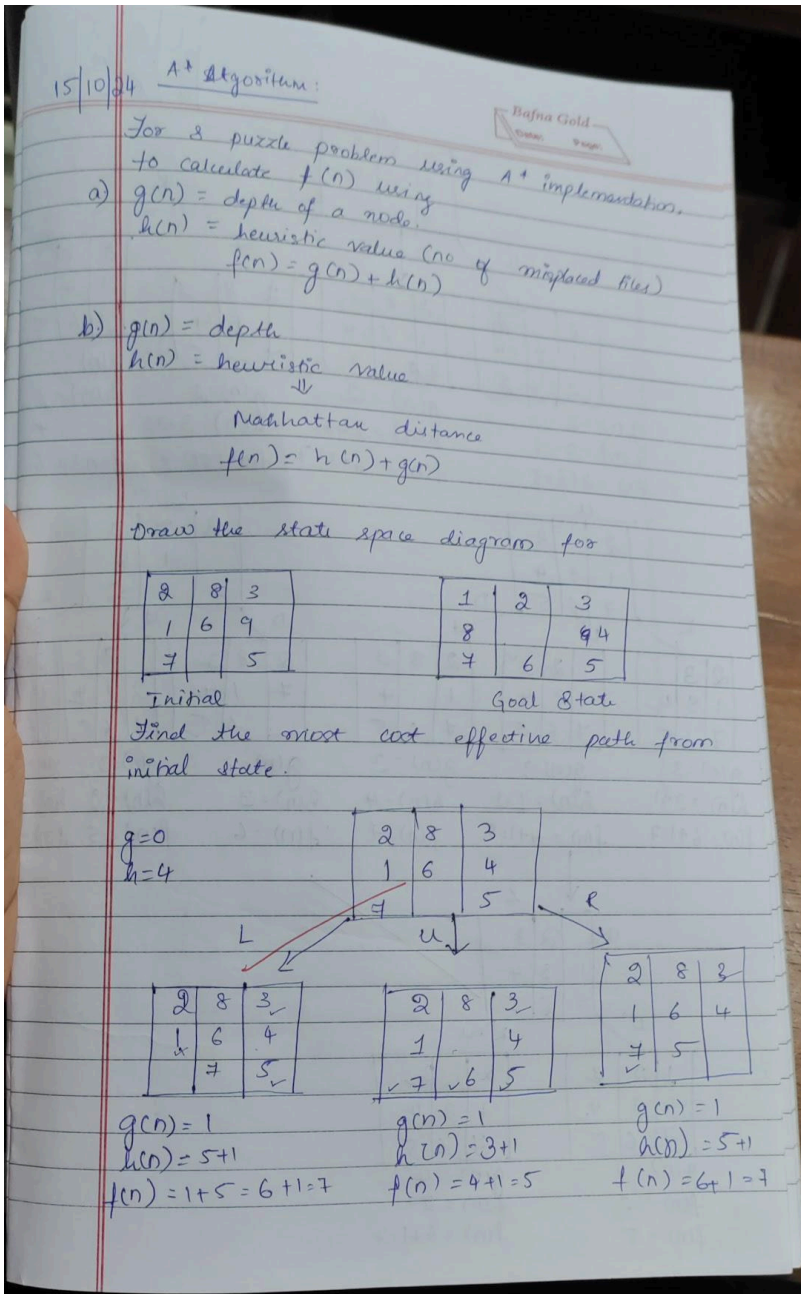
1 2 5
3 4 0
6 7 8

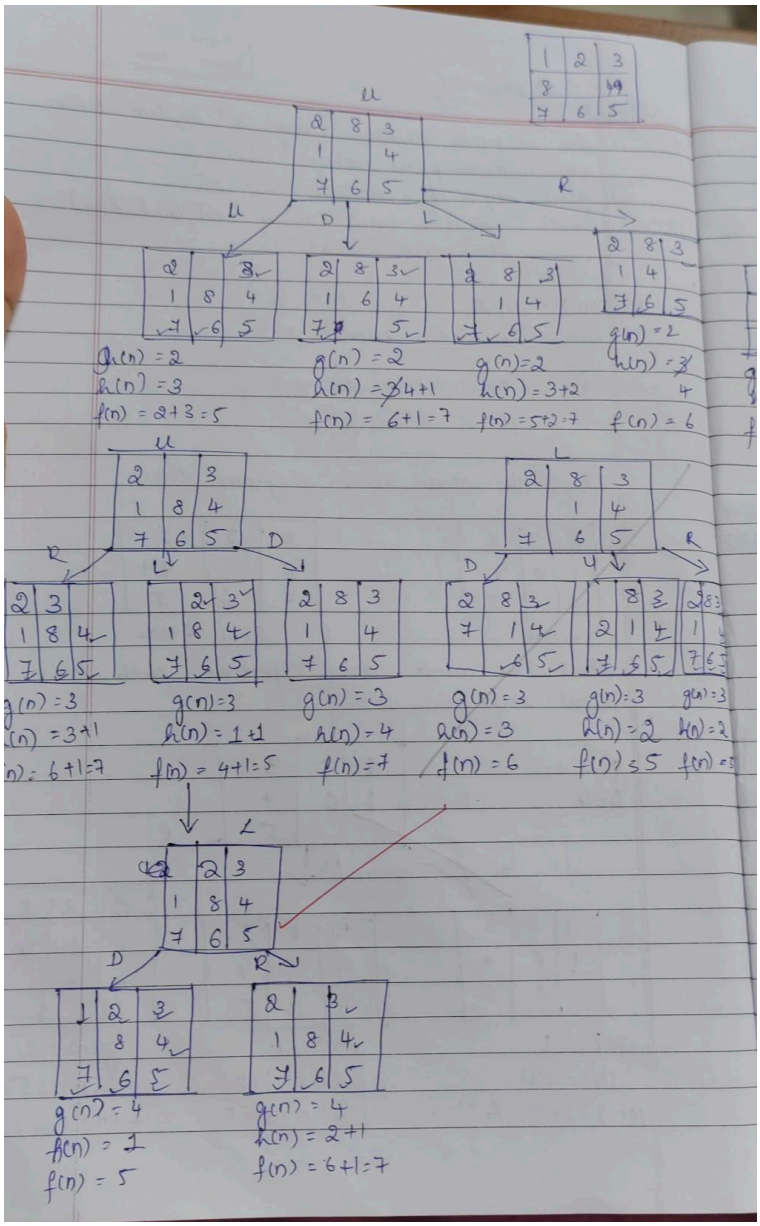
1 2 0
3 4 5
6 7 8

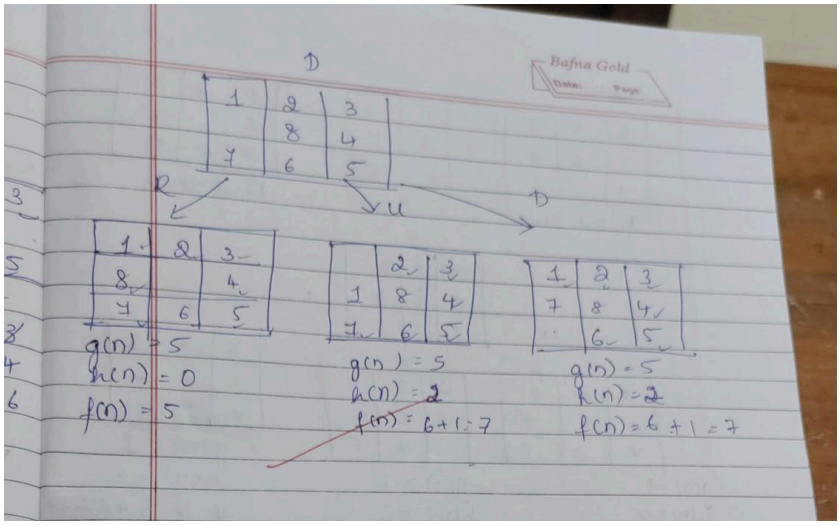
Program 3:

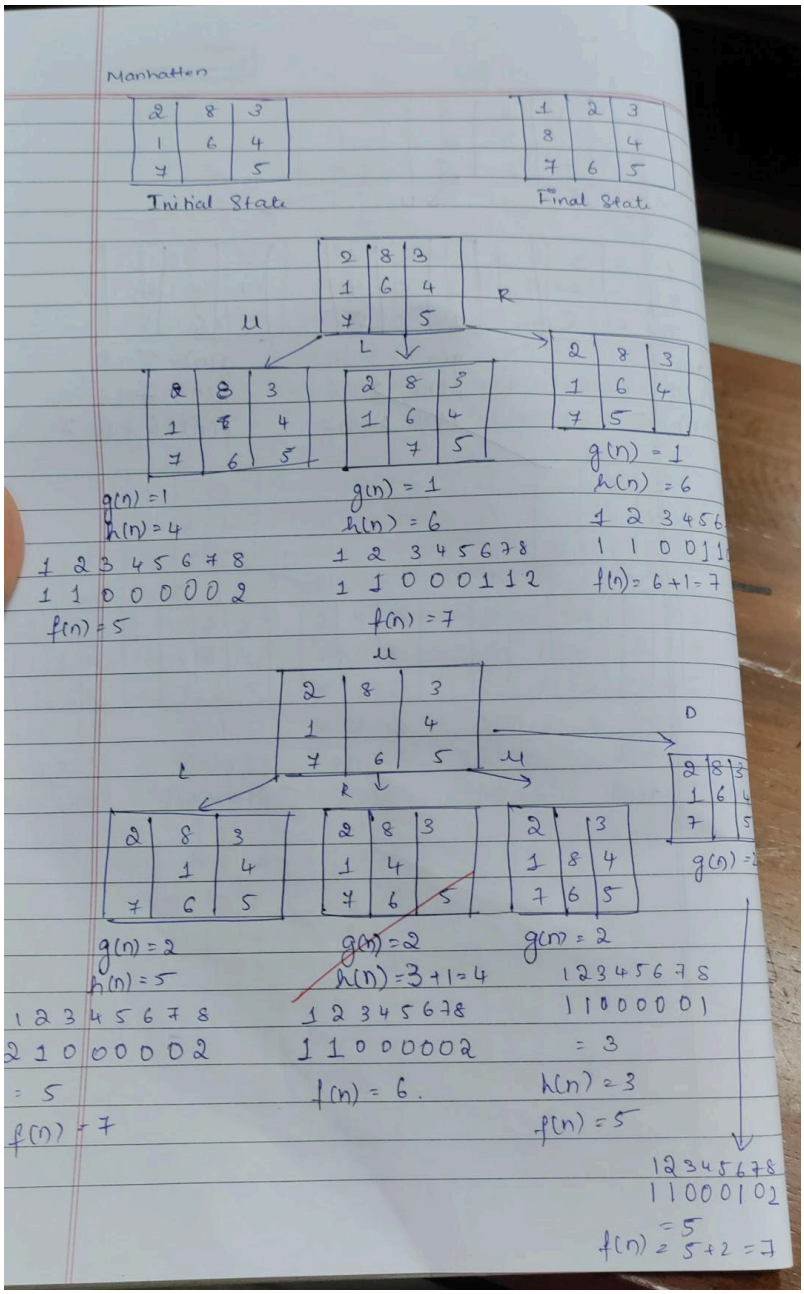
Implement A* search algorithm:

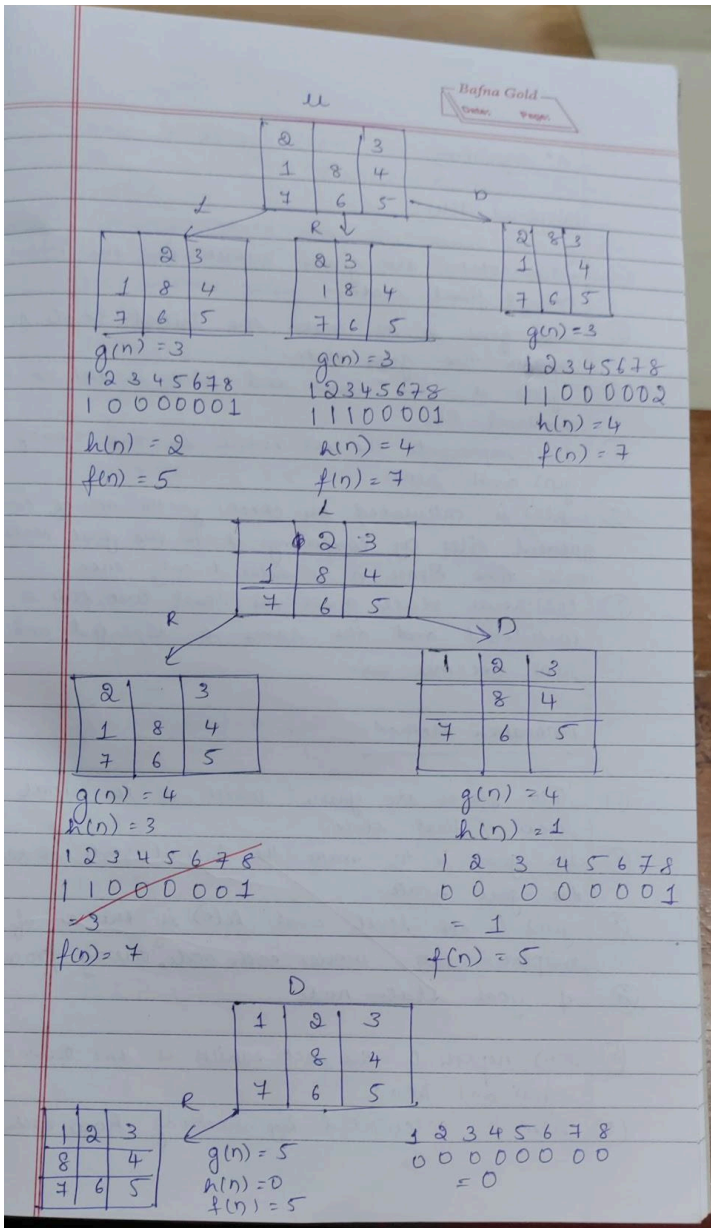
A* Manhattan Distance:

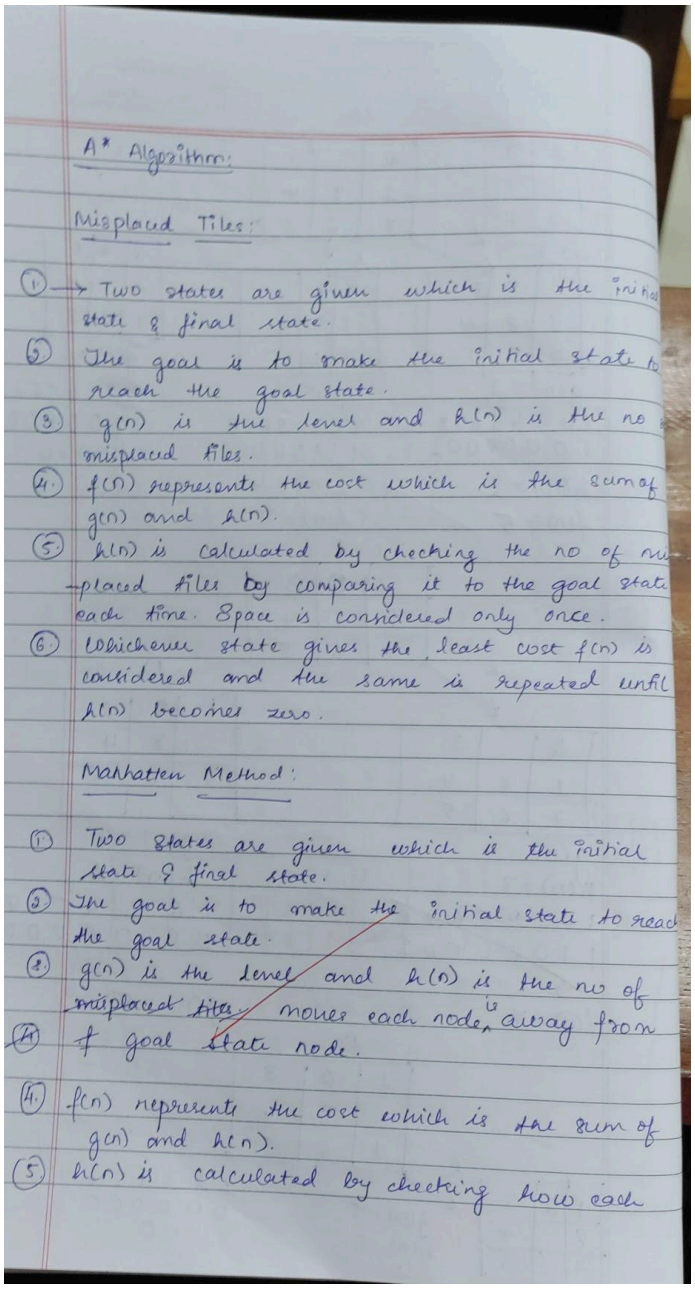



g(n) = 2
h(n) = 3
f(n) = 2+3 = 5
g(n) = 2
h(n) = 4+1
f(n) = 6+1 = 7
g(n) = 2
h(n) = 3+2
f(n) = 5+2 = 7
g(n) = 2
h(n) = 4
f(n) = 6
g(n) = 3
h(n) = 3+1
f(n) = 6+1 = 7
g(n) = 3
h(n) = 1+1
f(n) = 4+1 = 5
g(n) = 3
h(n) = 4
f(n) = 7
g(n) = 3
h(n) = 3
f(n) = 6
g(n) = 3
h(n) = 2
f(n) = 5
g(n) = 4
h(n) = 1
f(n) = 5
g(n) = 4
h(n) = 2+1
f(n) = 6+1 = 7


Bafna Gold
g(n) = 5
h(n) = 0
f(n) = 5
g(n) = 5
h(n) = 2
f(n) = 6+1 = 7
g(n) = 5
h(n) = 2
f(n) = 6+1 = 7


Manhatten
Initial State
Final State
g(n) = 1
h(n) = 4
1 2 3 4 5 6 7 8
1 1 0 0 0 0 0 2
f(n) = 5
g(n) = 1
h(n) = 6
1 2 3 4 5 6 7 8
1 1 0 0 0 1 1 2
f(n) = 7
g(n) = 1
h(n) = 6
f(n) = 6+1 = 7
g(n) = 2
h(n) = 5
1 2 3 4 5 6 7 8
2 1 0 0 0 0 0 2
= 5
f(n) = 7
g(n) = 2
h(n) = 3+1 = 4
1 2 3 4 5 6 7 8
1 1 0 0 0 0 0 2
f(n) = 6.
g(n) = 2
1 2 3 4 5 6 7 8
1 1 0 0 0 0 0 1
= 3
h(n) = 3
f(n) = 5
1 2 3 4 5 6 7 8
1 1 0 0 0 1 0 2
= 5
f(n) = 5+2 = 7


Bafna Gold
L
R
D
g(n) = 3
1 2 3 4 5 6 7 8
1 0 0 0 0 0 0 1
h(n) = 2
f(n) = 5
g(n) = 3
1 2 3 4 5 6 7 8
1 1 1 0 0 0 0 1
h(n) = 4
f(n) = 7
g(n) = 3
1 2 3 4 5 6 7 8
1 1 0 0 0 0 0 2
h(n) = 4
f(n) = 7
g(n) = 4
h(n) = 3
1 2 3 4 5 6 7 8
1 1 0 0 0 0 0 1
= 3
f(n) = 7
g(n) = 4
h(n) = 1
1 2 3 4 5 6 7 8
0 0 0 0 0 0 0 1
= 1
f(n) = 5
g(n) = 5
h(n) = 0
f(n) = 5
1 2 3 4 5 6 7 8
0 0 0 0 0 0 0 0
= 0



A* Algorithm:

Misplaced Tiles:

1. Two states are given which is the initial state & final state.
2. The goal is to make the initial state to reach the goal state.
3. g(n) is the level and h(n) is the no of misplaced tiles.
4. f(n) represents the cost which is the sum of g(n) and h(n).
5. h(n) is calculated by checking the no of misplaced tiles by comparing it to the goal state each time. Space is considered only once.
6. Whichever state gives the least cost f(n) is considered and the same is repeated until h(n) becomes zero.

Manhatten Method:

1. Two states are given which is the initial state & final state.
2. The goal is to make the initial state to reach the goal state.
3. g(n) is the level and h(n) is the no of misplaced tiles, moves each node away from
4. f goal state node.
4. f(n) represents the cost which is the sum of g(n) and h(n).
5. h(n) is calculated by checking how each

node is away from the goal state nodes. The movement is represented by a numerical value.

(6) Whichever state gives the least cost of f(n) is considered & the same is repeated until h(n) becomes zero.

Output:

Misplaced tiles:

Initial State:

2 8 3 1 6 4 7 0 5

Goal State:

1 2 3 8 0 4 7 6 5

Level 0 (Initial State):

2 8 3
1 6 4
7 0 5

g(n) = 0 h(n) = 4 f(n) = 4

Level 1:

2 8 3
1 0 4
7 6 5

g(n) = 1, h(n) = 3, f(n) = 4

2 8 3
1 6 4
0 7 5

g(n) = 1 h(n) = 5, f(n) = 6

```
import heapq

# Function to check if the puzzle is in the goal state
def is_goal(state, goal_state):
    return state == goal_state

# Function to calculate the Manhattan distance heuristic
def manhattan_distance(state, goal_state):
    distance = 0
    for i in range(1, 9):  # Skip the blank tile (0)
        current_index = state.index(i)
        goal_index = goal_state.index(i)
        current_row, current_col = divmod(current_index, 3)
        goal_row, goal_col = divmod(goal_index, 3)
        distance += abs(current_row - goal_row) + abs(current_col - goal_col)
    return distance

# Function to find possible moves (successors) from the current state
def get_successors(state):
    successors = []
    blank_idx = state.index(0)  # Find the index of the blank (0)

    # Possible moves: up, down, left, right
    moves = {
        "up": -3, "down": 3, "left": -1, "right": 1
    }

    for direction, move in moves.items():
        new_idx = blank_idx + move
        if 0 <= new_idx < 9:  # Check if the move is within bounds
            if direction == "left" and blank_idx % 3 == 0:
                continue  # Skip invalid move to the left
            if direction == "right" and blank_idx % 3 == 2:
                continue  # Skip invalid move to the right
            new_state = state[:]
            new_state[blank_idx], new_state[new_idx] = new_state[new_idx],
new_state[blank_idx]
            successors.append(new_state)

    return successors

# A* Algorithm using Manhattan Distance heuristic
def astar_manhattan_distance(start_state, goal_state):
    open_list = []
    closed_list = set()

    # Push the initial state into the priority queue (heap), with f = g + h
```

```
    initial_h = manhattan_distance(start_state, goal_state)
    heapq.heappush(open_list, (initial_h, 0, start_state, []))

    print(f"\nLevel 0 (Initial State):")
    display_states_side_by_side([start_state])  # Display the start state
    print(f"g(n) = 0, h(n) = {initial_h}, f(n) = {initial_h}\n")

    level = 1  # Track levels for display

    while open_list:
        f, g, current_state, path = heapq.heappop(open_list)

        if is_goal(current_state, goal_state):
            return path + [current_state]  # Return the path when goal is reached

        if tuple(current_state) in closed_list:
            continue

        closed_list.add(tuple(current_state))

        # Expand the current node (find successors)
        level_states = []
        for successor in get_successors(current_state):
            if tuple(successor) not in closed_list:
                new_g = g + 1  # Increment the cost to reach the successor
                new_h = manhattan_distance(successor, goal_state)
                new_f = new_g + new_h
                heapq.heappush(open_list, (new_f, new_g, successor, path + [current_state]))
                level_states.append((successor, new_g, new_h, new_f))

        if level_states:
            print(f"\nLevel {level}:")
            for state_info in level_states:
                state, new_g, new_h, new_f = state_info
                display_states_side_by_side([state])
                print(f"g(n) = {new_g}, h(n) = {new_h}, f(n) = {new_f}\n")
            level += 1

    return None  # No solution found

# Function to display the 8-puzzle in a readable format, multiple states side by side
def display_states_side_by_side(states):
    lines = [""] * 3  # Each puzzle has 3 lines

    for state in states:
        for i in range(0, 9, 3):
            lines[i // 3] += f"{state[i:i+3]}    "
```

```
    for line in lines:
        print(line)

# Main function to take input and run the A* algorithm
def main():
    print("Enter the start state of the 8-puzzle (9 numbers, use 0 for the blank tile):")
    start_state = list(map(int, input().split()))

    print("Enter the goal state of the 8-puzzle (9 numbers, use 0 for the blank tile):")
    goal_state = list(map(int, input().split()))

    print("\nSolving the 8-puzzle...\n")

    solution = astar_manhattan_distance(start_state, goal_state)

    if solution:
        print("\nSolution Path Found!")
        for step, state in enumerate(solution):
            print(f"Step {step}:")
            display_states_side_by_side([state])
    else:
        print("No solution found.")

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

Enter the start state of the 8-puzzle (9 numbers, use 0 for the blank tile):
2 8 3 1 6 4 7 0 5
Enter the goal state of the 8-puzzle (9 numbers, use 0 for the blank tile):
1 2 3 8 0 4 7 6 5

Solving the 8-puzzle...

Output:

Level 0 (Initial State):
[2, 8, 3]
[1, 6, 4]
[7, 0, 5]
g(n) = 0, h(n) = 5, f(n) = 5

Level 1:
[2, 8, 3]
[1, 0, 4]

[7, 6, 5]
g(n) = 1, h(n) = 4, f(n) = 5

[2, 8, 3]
[1, 6, 4]
[0, 7, 5]
g(n) = 1, h(n) = 6, f(n) = 7

[2, 8, 3]
[1, 6, 4]
[7, 5, 0]
g(n) = 1, h(n) = 6, f(n) = 7

Level 2:
[2, 0, 3]
[1, 8, 4]
[7, 6, 5]
g(n) = 2, h(n) = 3, f(n) = 5

[2, 8, 3]
[0, 1, 4]
[7, 6, 5]
g(n) = 2, h(n) = 5, f(n) = 7

[2, 8, 3]
[1, 4, 0]
[7, 6, 5]
g(n) = 2, h(n) = 5, f(n) = 7

Level 3:
[0, 2, 3]
[1, 8, 4]
[7, 6, 5]
g(n) = 3, h(n) = 2, f(n) = 5

[2, 3, 0]
[1, 8, 4]
[7, 6, 5]
g(n) = 3, h(n) = 4, f(n) = 7

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

g(n) = 4, h(n) = 1, f(n) = 5

Level 5:
[1, 2, 3]
[7, 8, 4]
[0, 6, 5]
g(n) = 5, h(n) = 2, f(n) = 7

[1, 2, 3]
[8, 0, 4]
[7, 6, 5]
g(n) = 5, h(n) = 0, f(n) = 5

Solution Path Found!
Step 0:
[2, 8, 3]
[1, 6, 4]
[7, 0, 5]
Step 1:
[2, 8, 3]
[1, 0, 4]
[7, 6, 5]
Step 2:
[2, 0, 3]
[1, 8, 4]
[7, 6, 5]
Step 3:
[0, 2, 3]
[1, 8, 4]
[7, 6, 5]
Step 4:
[1, 2, 3]
[0, 8, 4]
[7, 6, 5]
Step 5:
[1, 2, 3]
[8, 0, 4]
[7, 6, 5]

A* Misplaced Tiles:

```
import heapq

# Function to check if the puzzle is in the goal state
def is_goal(state, goal_state):
    return state == goal_state

# Function to calculate the Misplaced Tiles heuristic
def misplaced_tiles(state, goal_state):
    count = 0
    for i in range(len(state)):
        if state[i] != goal_state[i] and state[i] != 0:  # Exclude the blank tile (0)
            count += 1
    return count

# Function to find possible moves (successors) from the current state
def get_successors(state):
    successors = []
    blank_idx = state.index(0)  # Find the index of the blank (0)

    # Possible moves: up, down, left, right
    moves = {
        "up": -3, "down": 3, "left": -1, "right": 1
    }

    for direction, move in moves.items():
        new_idx = blank_idx + move
        if 0 <= new_idx < 9:  # Check if the move is within bounds
            if direction == "left" and blank_idx % 3 == 0:
                continue  # Skip invalid move to the left
            if direction == "right" and blank_idx % 3 == 2:
                continue  # Skip invalid move to the right
            new_state = state[:]
            new_state[blank_idx], new_state[new_idx] = new_state[new_idx],
new_state[blank_idx]
            successors.append(new_state)

    return successors

# A* Algorithm using Misplaced Tiles heuristic
def astar_misplaced_tiles(start_state, goal_state):
    open_list = []
    closed_list = set()
```

```
    # Push the initial state into the priority queue (heap), with f = g + h
    initial_h = misplaced_tiles(start_state, goal_state)
    heapq.heappush(open_list, (initial_h, 0, start_state, []))

    print(f"\nLevel 0 (Initial State):")
    display_states_side_by_side([start_state])  # Display the start state
    print(f"g(n) = 0, h(n) = {initial_h}, f(n) = {initial_h}\n")

    level = 1  # Track levels for display

    while open_list:
        f, g, current_state, path = heapq.heappop(open_list)

        if is_goal(current_state, goal_state):
            return path + [current_state]  # Return the path when goal is reached

        if tuple(current_state) in closed_list:
            continue

        closed_list.add(tuple(current_state))

        # Expand the current node (find successors)
        level_states = []
        for successor in get_successors(current_state):
            if tuple(successor) not in closed_list:
                new_g = g + 1  # Increment the cost to reach the successor
                new_h = misplaced_tiles(successor, goal_state)
                new_f = new_g + new_h
                heapq.heappush(open_list, (new_f, new_g, successor, path + [current_state]))
                level_states.append((successor, new_g, new_h, new_f))

        if level_states:
            print(f"\nLevel {level}:")
            for state_info in level_states:
                state, new_g, new_h, new_f = state_info
                display_states_side_by_side([state])
                print(f"g(n) = {new_g}, h(n) = {new_h}, f(n) = {new_f}\n")
            level += 1

    return None  # No solution found

# Function to display the 8-puzzle in a readable format, multiple states side by side
def display_states_side_by_side(states):
    lines = [""] * 3  # Each puzzle has 3 lines

    for state in states:
        for i in range(0, 9, 3):
```

```
            lines[i // 3] += f"{state[i:i+3]}     "

    for line in lines:
        print(line)

# Main function to take input and run the A* algorithm
def main():
    print("Enter the start state of the 8-puzzle (9 numbers, use 0 for the blank tile):")
    start_state = list(map(int, input().split()))

    print("Enter the goal state of the 8-puzzle (9 numbers, use 0 for the blank tile):")
    goal_state = list(map(int, input().split()))

    print("\nSolving the 8-puzzle...\n")

    solution = astar_misplaced_tiles(start_state, goal_state)

    if solution:
        print("\nSolution Path Found!")
        for step, state in enumerate(solution):
            print(f"Step {step}:")
            display_states_side_by_side([state])
    else:
        print("No solution found.")

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

```
Enter the start state of the 8-puzzle (9 numbers, use 0 for the blank tile):
2 8 3 1 6 4 7 0 5
Enter the goal state of the 8-puzzle (9 numbers, use 0 for the blank tile):
1 2 3 8 0 4 7 6 5

Solving the 8-puzzle...
```

Output:

```
Level 0 (Initial State):
[2, 8, 3]
[1, 6, 4]
[7, 0, 5]
g(n) = 0, h(n) = 4, f(n) = 4


Level 1:
[2, 8, 3]
```

[1, 0, 4]
[7, 6, 5]
g(n) = 1, h(n) = 3, f(n) = 4

[2, 8, 3]
[1, 6, 4]
[0, 7, 5]
g(n) = 1, h(n) = 5, f(n) = 6

[2, 8, 3]
[1, 6, 4]
[7, 5, 0]
g(n) = 1, h(n) = 5, f(n) = 6

Level 2:
[2, 0, 3]
[1, 8, 4]
[7, 6, 5]
g(n) = 2, h(n) = 3, f(n) = 5

[2, 8, 3]
[0, 1, 4]
[7, 6, 5]
g(n) = 2, h(n) = 3, f(n) = 5

[2, 8, 3]
[1, 4, 0]
[7, 6, 5]
g(n) = 2, h(n) = 4, f(n) = 6

Level 3:
[0, 2, 3]
[1, 8, 4]
[7, 6, 5]
g(n) = 3, h(n) = 2, f(n) = 5

[2, 3, 0]
[1, 8, 4]
[7, 6, 5]
g(n) = 3, h(n) = 4, f(n) = 7

Level 4:
[0, 8, 3]
[2, 1, 4]

[7, 6, 5]
g(n) = 3, h(n) = 3, f(n) = 6

[2, 8, 3]
[7, 1, 4]
[0, 6, 5]
g(n) = 3, h(n) = 4, f(n) = 7

Level 5:
[1, 2, 3]
[0, 8, 4]
[7, 6, 5]
g(n) = 4, h(n) = 1, f(n) = 5

Level 6:
[1, 2, 3]
[7, 8, 4]
[0, 6, 5]
g(n) = 5, h(n) = 2, f(n) = 7

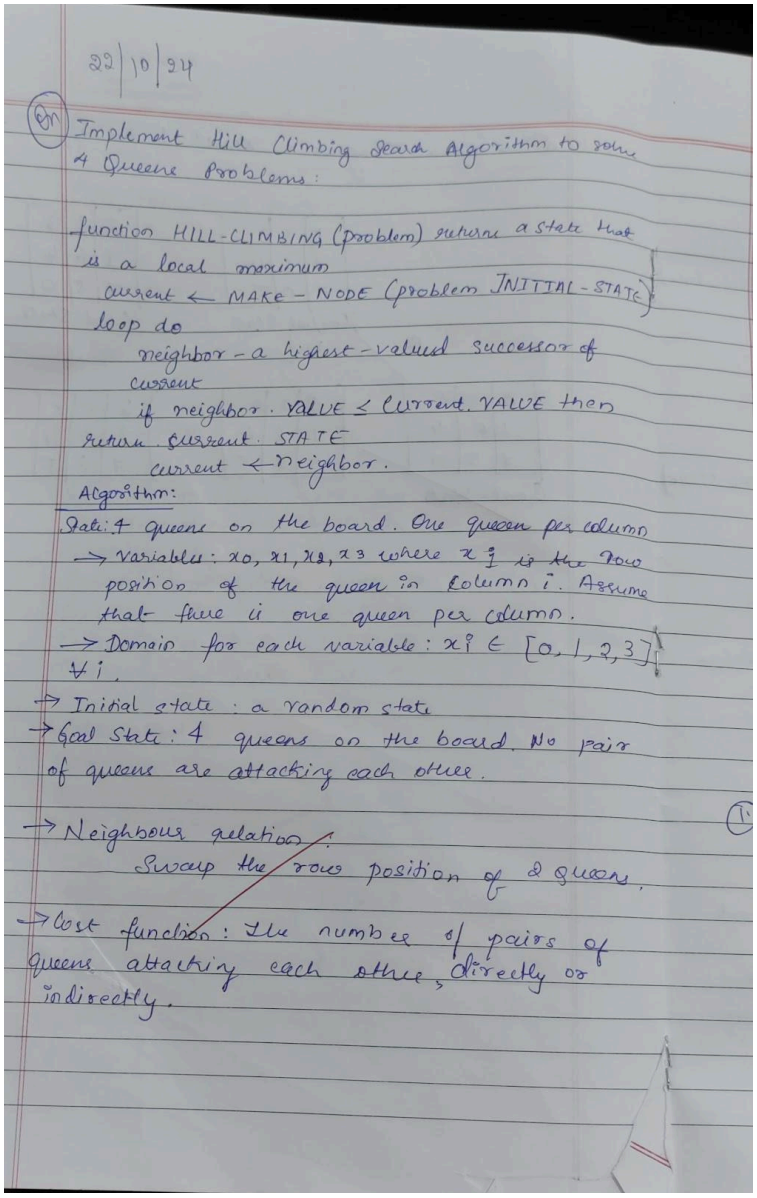
[1, 2, 3]
[8, 0, 4]
[7, 6, 5]
g(n) = 5, h(n) = 0, f(n) = 5

Solution Path Found!
Step 0:
[2, 8, 3]
[1, 6, 4]
[7, 0, 5]
Step 1:
[2, 8, 3]
[1, 0, 4]
[7, 6, 5]
Step 2:
[2, 0, 3]
[1, 8, 4]
[7, 6, 5]
Step 3:
[0, 2, 3]
[1, 8, 4]
[7, 6, 5]
Step 4:
[1, 2, 3]

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

Program 4:

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

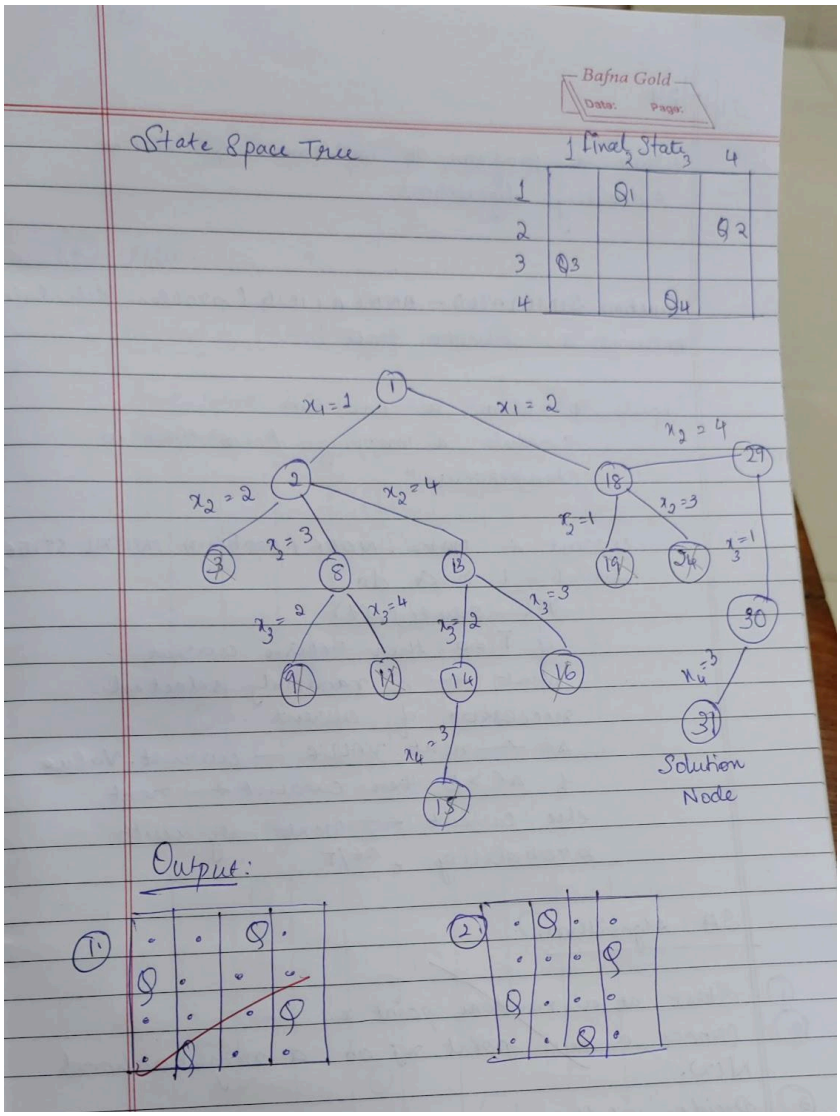


22/10/24

8M Implement Hill Climbing Search Algorithm to solve 4 Queens Problems:

function HILL-CLIMBING (problem) returns a state that is a local maximum
current ← MAKE-NODE (problem.INITIAL-STATE)
loop do
neighbor - a highest-valued successor of current
if neighbor.VALUE ≤ current.VALUE then return current.STATE
current ← neighbor.

Algorithm:
State: 4 queens on the board. One queen per column
→ Variables: x_0, x_1, x_2, x_3 where x_i ~~is the~~ row position of the queen in column i. Assume that there is one queen per column.
→ Domain for each variable: $x_i \in [0, 1, 2, 3]$ $\forall i$.
→ Initial state: a random state
→ Goal State: 4 queens on the board. No pair of queens are attacking each other.
→ Neighbours relation:
Swap the rows position of 2 queens.
→ Cost function: The number of pairs of queens attacking each other, directly or indirectly.




```
import random

def get_attacking_pairs(state):
    """Calculates the number of attacking pairs of queens."""
    attacks = 0
    n = len(state)
    for i in range(n):
        for j in range(i + 1, n):
            if state[i] == state[j]:
                attacks += 1
            if abs(state[i] - state[j]) == abs(i - j):
                attacks += 1
    return attacks
def generate_successors(state):
    """Generates all possible successors by moving each queen to every other column in
```

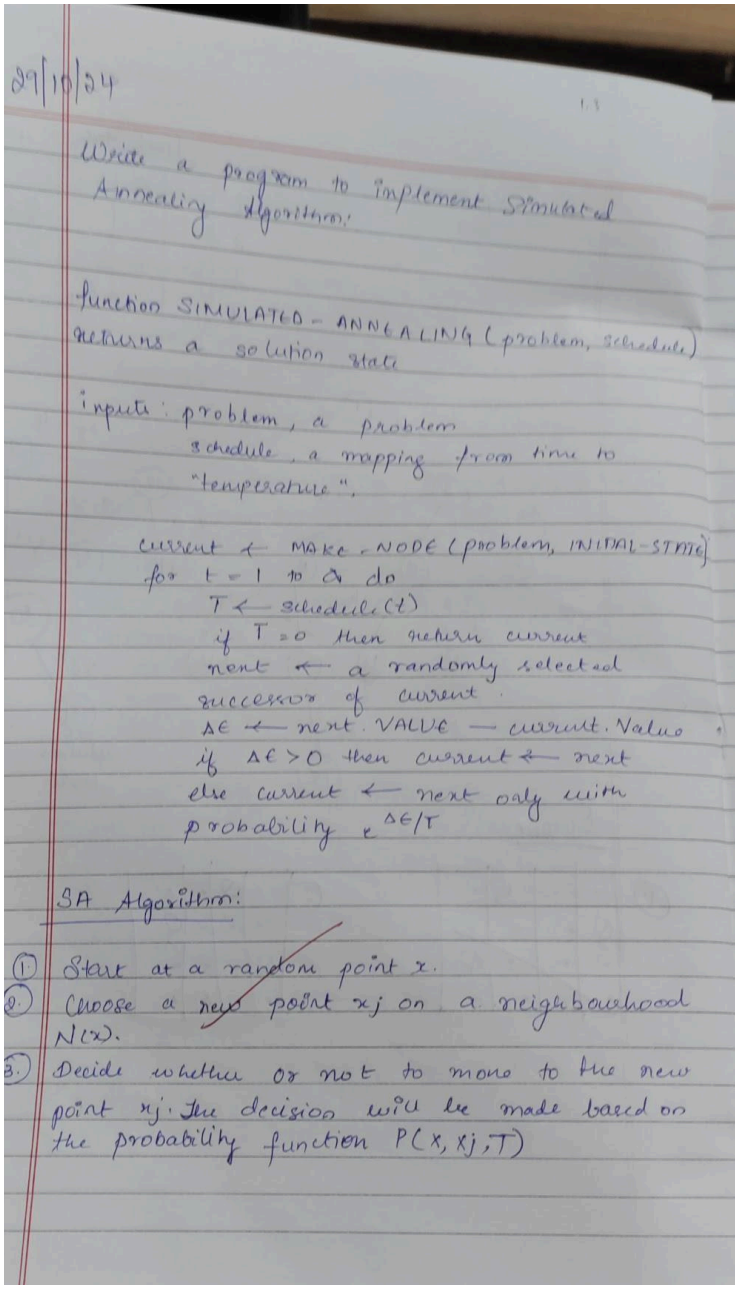
```
its row."""
    n = len(state)
    successors = []
    for row in range(n):
        for col in range(n):
            if col != state[row]:
                new_state = state[:]
                new_state[row] = col
                successors.append(new_state)
    return successors

def hill_climbing(n):
    """Hill climbing algorithm for n-queens problem."""
    current = [random.randint(0, n - 1) for _ in range(n)]
    while True:
        current_attacks = get_attacking_pairs(current)
        successors = generate_successors(current)
        neighbor = min(successors, key=get_attacking_pairs)
        neighbor_attacks = get_attacking_pairs(neighbor)
        if neighbor_attacks >= current_attacks:
            return current, current_attacks
        current = neighbor

def print_board(state):
    """Prints the board with queens placed."""
    n = len(state)
    board = [["." for _ in range(n)] for _ in range(n)]
    for row in range(n):
        board[row][state[row]] = "Q"
    for row in board:
        print(" ".join(row))
    print("\n")

n = 4
solution, attacks = hill_climbing(n)
print("Final State (Solution):", solution)
print("Number of Attacking Pairs:", attacks)
print_board(solution)
```

Output:

Final State (Solution): [1, 3, 0, 2]
Number of Attacking Pairs: 0
. Q . .
. . . Q
Q . . .
. . Q .

Program 5:

Simulated Annealing to Solve 8-Queens problem:



29/10/24

Write a program to implement Simulated Annealing Algorithm.

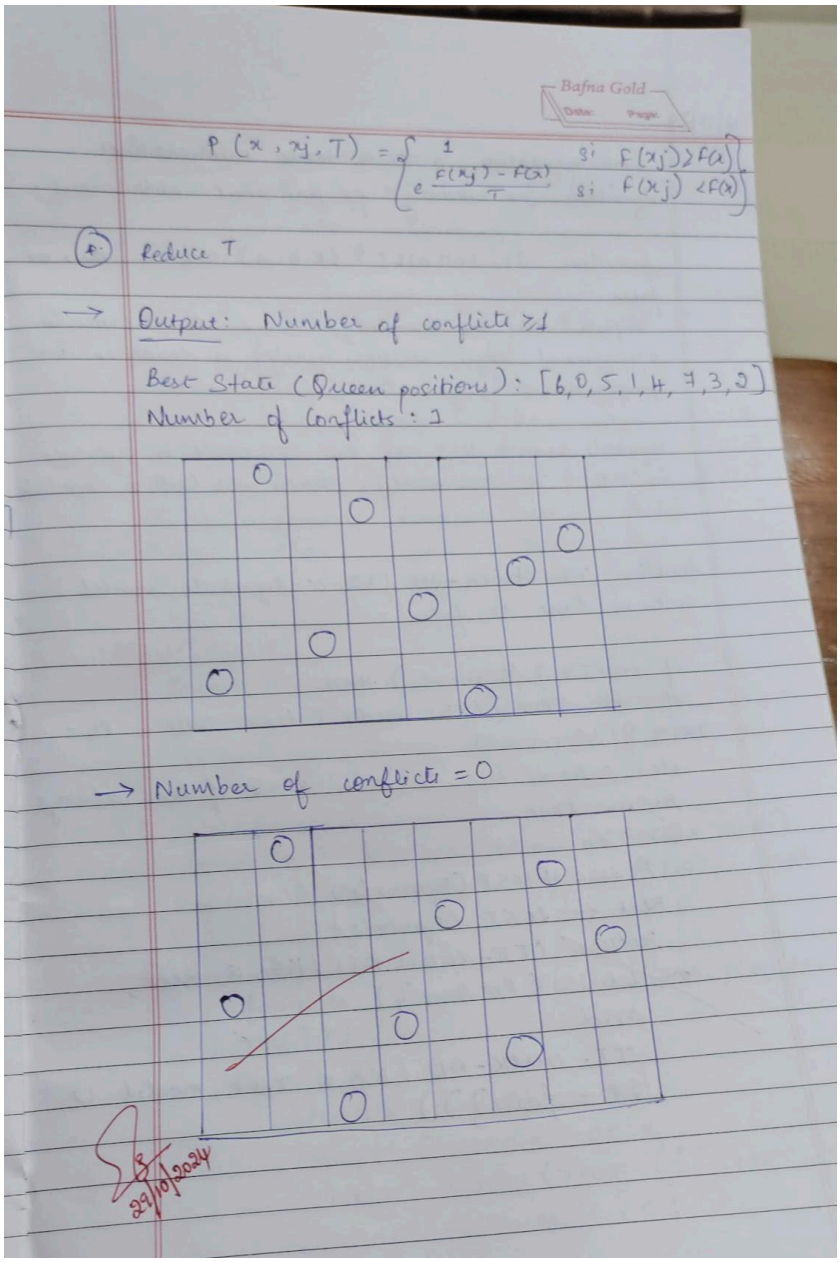
function SIMULATED-ANNEALING (problem, schedule) returns a solution state

inputs: problem, a problem
schedule, a mapping from time to "temperature".

current ← MAKE-NODE (problem, INITIAL-STATE)
for t = 1 to ∞ do
T ← schedule(t)
if T = 0 then return current
next ← a randomly selected successor of current.
ΔE ← next.VALUE — current.Value
if ΔE > 0 then current ← next
else current ← next only with probability $e^{\Delta E/T}$

SA Algorithm:

1. Start at a random point x.
2. Choose a new point xj on a neighbourhood N(x).
3. Decide whether or not to move to the new point xj. The decision will be made based on the probability function P(x, xj, T)



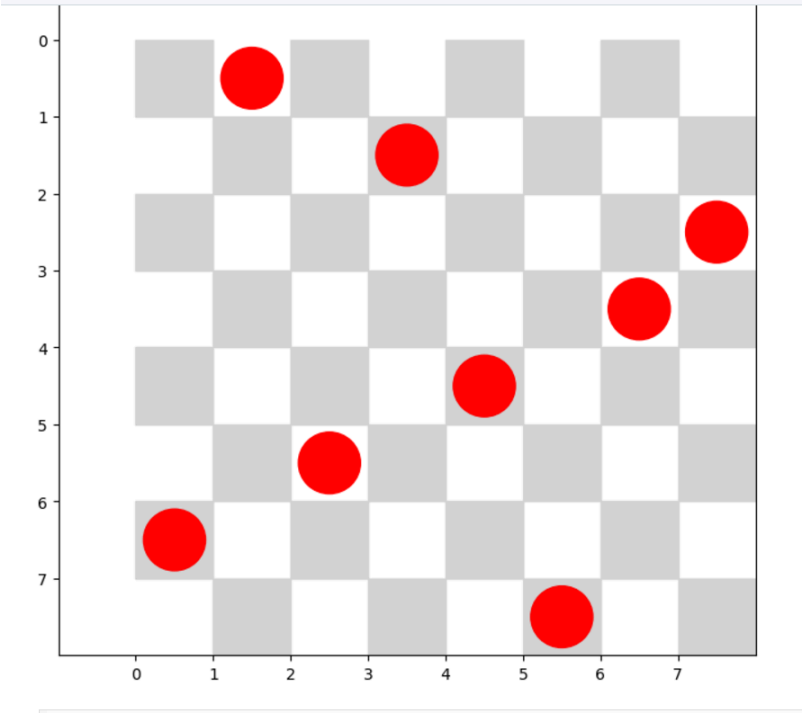

```
import random
import math
import matplotlib.pyplot as plt

def create_initial_solution(n):
    return random.sample(range(n), n)

def calculate_fitness(state):
    n = len(state)
    row_conflicts = sum([state.count(i) - 1 for i in state])
    diagonal_conflicts = 0
```

```
    for i in range(n):
        for j in range(i + 1, n):
            if abs(state[i] - state[j]) == abs(i - j):
                diagonal_conflicts += 1

    return row_conflicts + diagonal_conflicts

def random_neighbor(state):
    neighbor = state[:]
    i = random.randint(0, len(state) - 1)
    neighbor[i] = random.randint(0, len(state) - 1)
    return neighbor

def simulated_annealing(n, initial_temp=1000, cooling_rate=0.95, max_iterations=1000):
    current_solution = create_initial_solution(n)
    current_fitness = calculate_fitness(current_solution)
    best_solution = current_solution
    best_fitness = current_fitness
    temperature = initial_temp

    for iteration in range(max_iterations):
        neighbor = random_neighbor(current_solution)
        neighbor_fitness = calculate_fitness(neighbor)

        fitness_diff = neighbor_fitness - current_fitness

        if fitness_diff < 0 or random.uniform(0, 1) < math.exp(-fitness_diff / temperature):
            current_solution = neighbor
            current_fitness = neighbor_fitness

            if current_fitness < best_fitness:
                best_solution = current_solution
                best_fitness = current_fitness

        temperature *= cooling_rate

    return best_solution, best_fitness

def plot_solution(solution):
    n = len(solution)
    plt.figure(figsize=(n, n))
    plt.xlim(-1, n)
    plt.ylim(-1, n)

    # Draw the chessboard
    for i in range(n):
        for j in range(n):
```

```
            if (i + j) % 2 == 0:
                plt.gca().add_patch(plt.Rectangle((j, i), 1, 1, color='lightgrey'))

    # Place the queens
    for col, row in enumerate(solution):
        plt.gca().add_patch(plt.Circle((col + 0.5, row + 0.5), 0.4, color='red'))

    plt.xticks(range(n))
    plt.yticks(range(n))
    plt.gca().invert_yaxis()
    plt.grid(False)
    plt.show()

# Parameters
n = 8  # Number of queens
best_solution, best_fitness = simulated_annealing(n)

# Output results
print(f"Best state (Queen positions): {best_solution}, Number of conflicts: {best_fitness}")

# Plot the solution
plot_solution(best_solution)
```

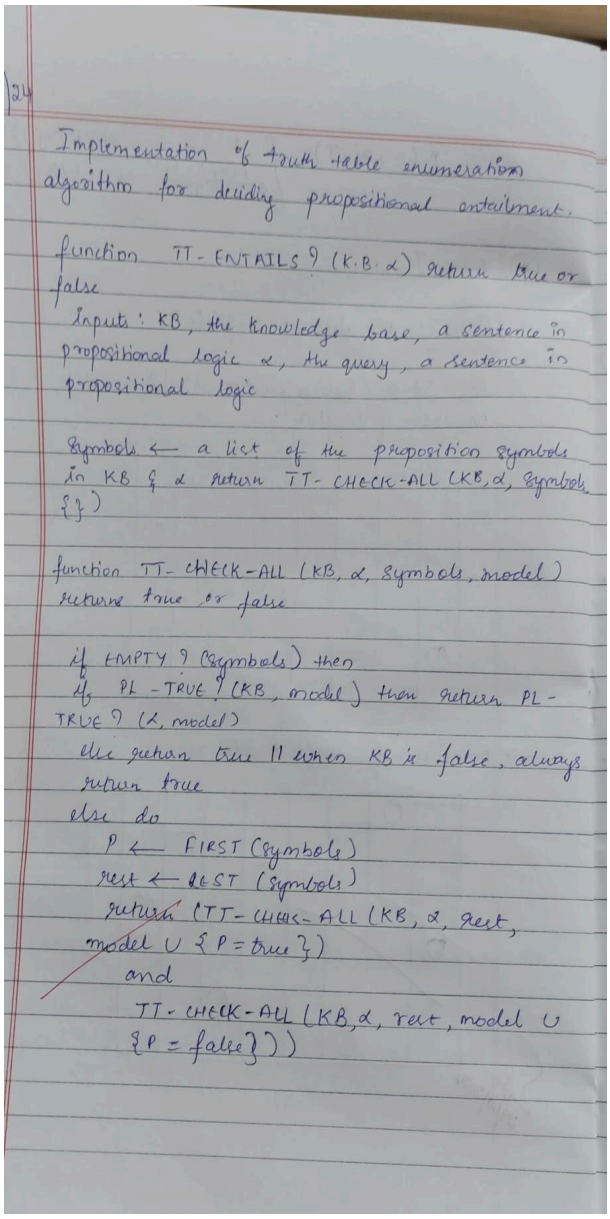
Output:

Best state (Queen positions): [6, 0, 5, 1, 4, 7, 3, 2], Number of conflicts: 1



Program 6:

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



Implementation of truth table enumeration algorithm for deciding propositional entailment.

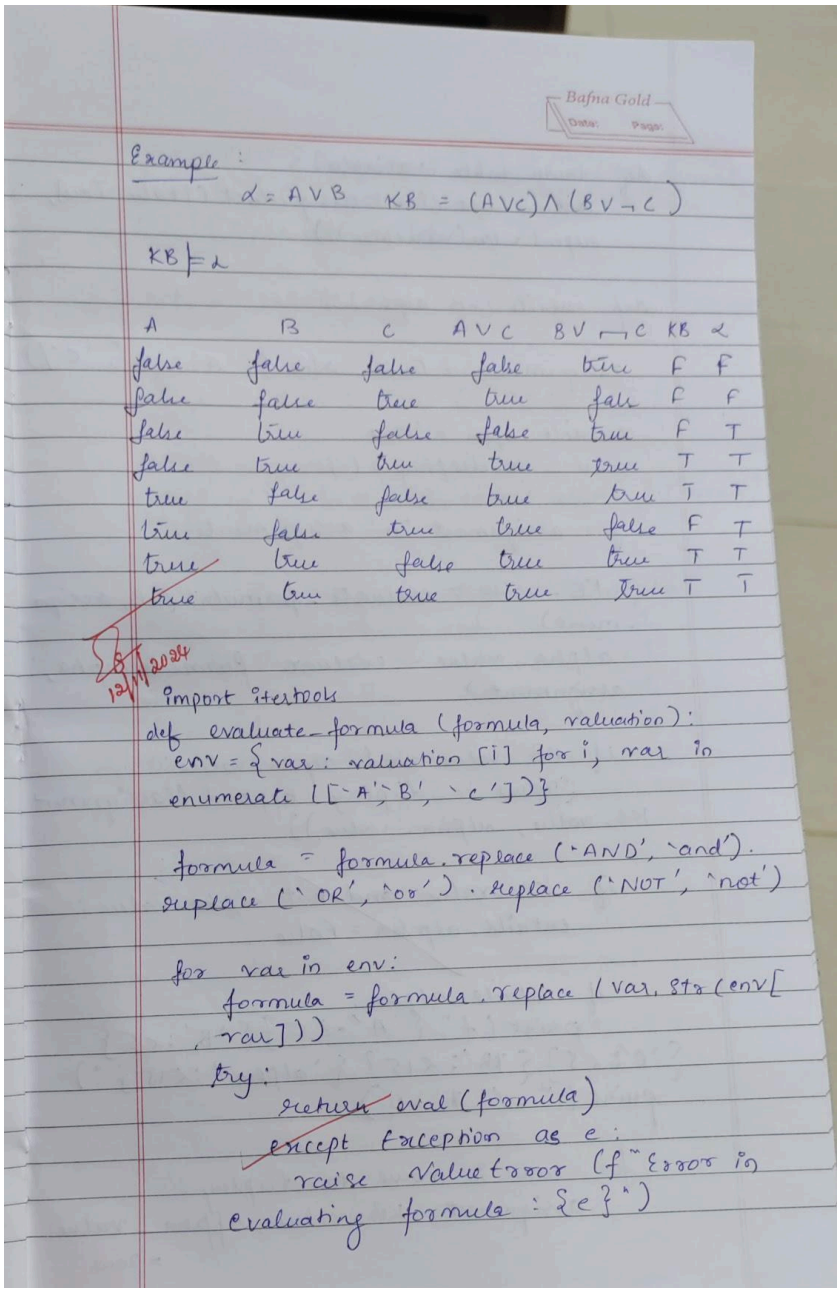
function TT-ENTAILS? (KB, α) return true or false

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

Symbols ← a list of the proposition symbols in KB & α return TT-CHECK-ALL (KB, α, Symbols, {})

function TT-CHECK-ALL (KB, α, symbols, model) returns true or false

if EMPTY? (Symbols) then
if PL-TRUE? (KB, model) then return PL-TRUE? (α, model)
else return true // when KB is false, always return true
else do
P ← FIRST (Symbols)
rest ← REST (Symbols)
return (TT-CHECK-ALL (KB, α, rest, model ∪ {P = true}) and TT-CHECK-ALL (KB, α, rest, model ∪ {P = false}))



import itertools

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

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

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

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

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

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

    # To keep track of whether KB entails alpha
    entails_alpha = True
    rows_to_display = []  # List to collect rows where KB entails alpha

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

        # Collect only the rows where KB and alpha are both true
        if KB_value and alpha_value:
            rows_to_display.append((assignment, KB_value, alpha_value))

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

    # If KB entails alpha, display the rows where KB and alpha are true
    if entails_alpha:
        # Print the header for the truth table (only for the rows where KB entails alpha)
```

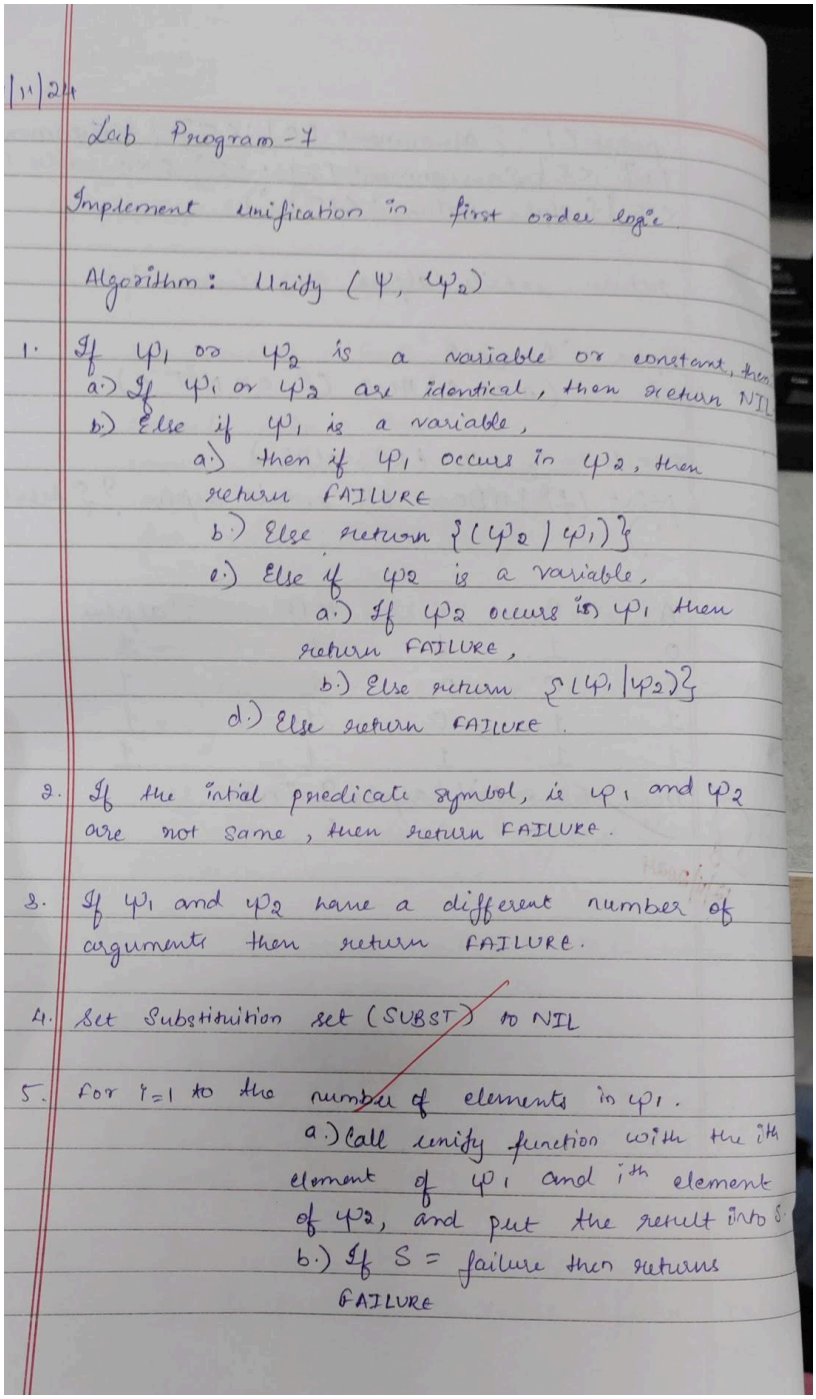
```
        print(f"{'A':<5}{'B':<5}{'C':<5}{'KB':<15}{'alpha':<15}")
        print("-" * 40)  # Separator for readability

        # Print the rows where KB entails alpha
        for row in rows_to_display:
            assignment, KB_value, alpha_value = row

print(f"{assignment[0]:<5}{assignment[1]:<5}{assignment[2]:<5}{KB_value:<15}{alpha_value:<15}")

    return entails_alpha

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

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

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

Output:

```
A    B    C    KB             alpha
----------------------------------------
0    1    1    1              1
1    0    0    1              1
1    1    0    1              1
1    1    1    1              1

Does KB entail alpha? True
```

Program 7:

Implement unification in first order logic:



Lab Program - 7

Implement unification in first order logic

Algorithm: Unify (ψ_1, ψ_2)

1. If ψ_1 or ψ_2 is a variable or constant, then
   a.) If ψ_1 or ψ_2 are identical, then return NIL
   b.) Else if ψ_1 is a variable,
      a.) then if ψ_1 occurs in ψ_2, then return FAILURE
      b.) Else return $\{(\psi_2 / \psi_1)\}$
   c.) Else if ψ_2 is a variable,
      a.) If ψ_2 occurs in ψ_1 then return FAILURE,
      b.) Else return $\{(\psi_1 / \psi_2)\}$
   d.) Else return FAILURE.

2. If the initial predicate symbol, is ψ_1 and ψ_2 are not same, then return FAILURE.

3. If ψ_1 and ψ_2 have a different number of arguments then return FAILURE.

4. Set Substitution set (SUBST) to NIL

5. For i=1 to the number of elements in ψ_1.
   a.) Call unify function with the ith element of ψ_1 and ith element of ψ_2, and put the result into S
   b.) If S = failure then returns FAILURE

c.) If S ≠ NIL then do,

a.) Apply S to the remainder of both L1 and L2.

b.) SUBST = APPEND (S, SUBST)

4.) Return SUBST

Eg(1):

$P(x, F(y)) \rightarrow (i)$

$P(a, F(g(x))) \rightarrow (ii)$

(i) & (ii) are identical if x is replaced with a in (ii)

a/x

$P(a, F(y))^{a/x} \rightarrow (1)$

in (1) replace y with g(x)

$P(a, F(g(x)))^{g(x)/y} \rightarrow (1)$.

Now (1) & (2) are same

Eg 2: $Q(a, g(x,a), f(y)) \rightarrow (1)$

$Q(a, g(f(h), a), x) \rightarrow (2)$

in (1) & (2) predicate are same & no of arguments are same.

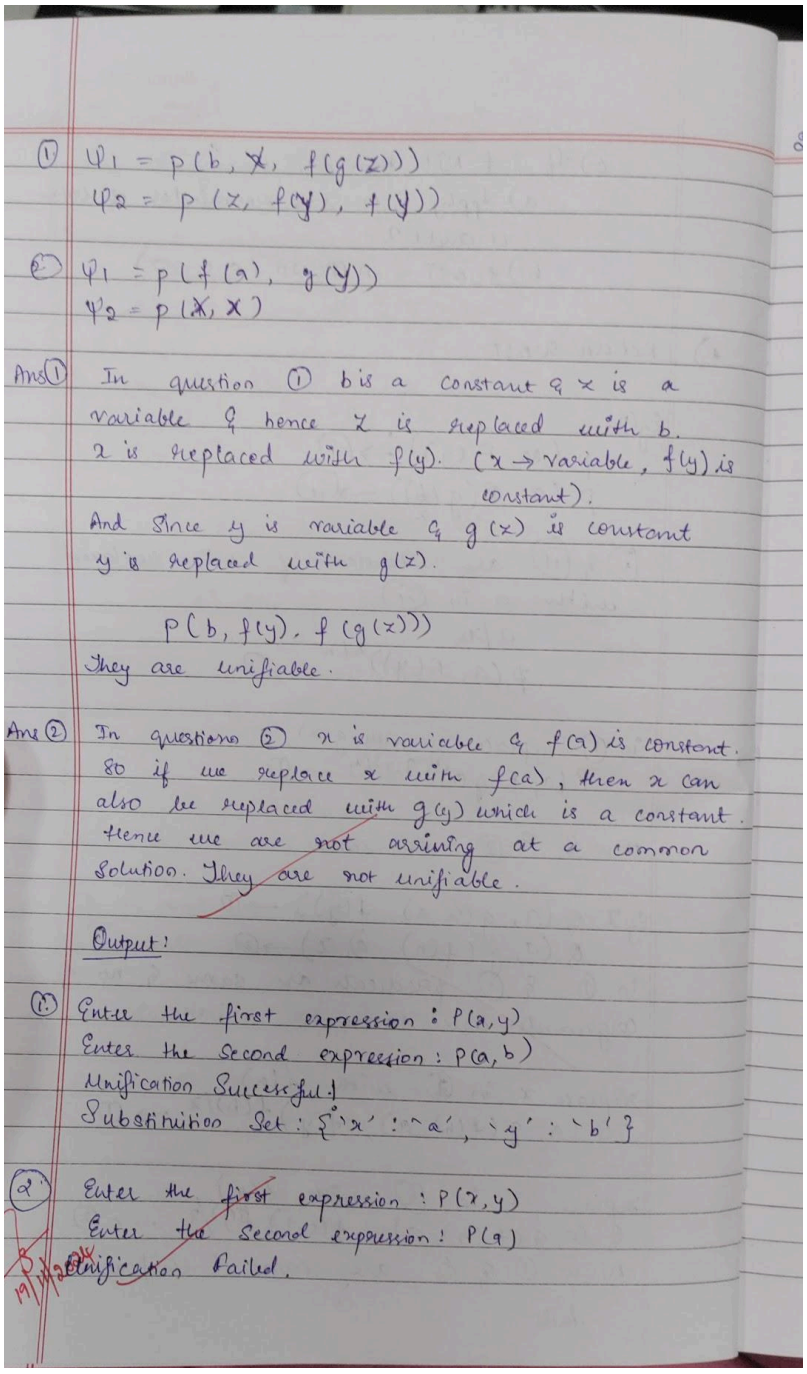
replace x in (2) with f(h)

$Q(a, g(f(h), a), f(y))^{f(h)/x} \rightarrow (1)$

replace x in (2) with f(y)

$Q(a, g(f(h), a), f(y))^{f(y)/x} \rightarrow (2)$

Now (1) & (2) are ~~not~~ same. & unification fails




```
def unify(expr1, expr2, subst={}):
    """
    A function to unify two expressions using the unification algorithm.
    :param expr1: First expression (string, constant, variable, or predicate).
    :param expr2: Second expression.
    :param subst: Current substitution dictionary.
    :return: Substitution set or None if unification fails.
    """
```

```
    if subst is None:
        return None  # Unification failed
    elif expr1 == expr2:
        return subst  # Expressions are already the same
    elif is_variable(expr1):
        return unify_variable(expr1, expr2, subst)
    elif is_variable(expr2):
        return unify_variable(expr2, expr1, subst)
    elif is_compound(expr1) and is_compound(expr2):
        if get_predicate(expr1) != get_predicate(expr2):
            return None  # Different predicates, cannot unify
        return unify(get_arguments(expr1), get_arguments(expr2), subst)
    elif isinstance(expr1, list) and isinstance(expr2, list):
        if len(expr1) != len(expr2):
            return None  # Different number of arguments
        if not expr1:  # Both lists are empty
            return subst
        return unify(expr1[1:], expr2[1:], unify(expr1[0], expr2[0], subst))
    else:
        return None  # No unification possible


def unify_variable(var, x, subst):
    """
    Handles variable unification.
    """
    if var in subst:
        return unify(subst[var], x, subst)  # Recursive unification
    elif x in subst:
        return unify(var, subst[x], subst)
    elif occurs_check(var, x):
        return None  # Failure: cyclic dependency
    else:
        subst[var] = x
        return subst


def is_variable(x):
    """
    Checks if x is a variable.
    Variables are usually single lowercase letters.
    """
    return isinstance(x, str) and x.islower()


def is_compound(x):
    """
```

```
    Checks if x is a compound expression (predicate with arguments).
    """
    return isinstance(x, str) and "(" in x and ")" in x


def get_predicate(expr):
    """
    Extracts the predicate from a compound expression.
    """
    return expr.split("(")[0]


def get_arguments(expr):
    """
    Extracts the arguments from a compound expression as a list.
    """
    args = expr[expr.index("(") + 1:expr.rindex(")")]
    return [arg.strip() for arg in args.split(",")]


def occurs_check(var, x):
    """
    Checks if var occurs in x (prevents infinite recursion).
    """
    if var == x:
        return True
    elif is_compound(x):
        return var in get_arguments(x)
    elif isinstance(x, list):
        return any(occurs_check(var, arg) for arg in x)
    return False


def user_input_unification():
    """
    Function to get user input and perform unification.
    """
    print("=== Unification Algorithm ===")
    expr1 = input("Enter the first expression (e.g., P(x, y)): ")
    expr2 = input("Enter the second expression (e.g., P(a, b)): ")

    # Perform unification
    result = unify(expr1, expr2)

    # Print the result
    if result is None:
        print("Unification Failed")
```

```
    else:
        print("Unification Successful!")
        print("Substitution Set:", result)


# Run the program
if __name__ == "__main__":
    user_input_unification()
```

=== Unification Algorithm ===

Output:

Enter the first expression (e.g., P(x, y)): P(x,y)
Enter the second expression (e.g., P(a, b)): P(a,b)
Unification Successful!
Substitution Set: {'x': 'a', 'y': 'b'}

```
def unify(expr1, expr2, subst={}):
    """
    A function to unify two expressions using the unification algorithm.
    :param expr1: First expression (string, constant, variable, or predicate).
    :param expr2: Second expression.
    :param subst: Current substitution dictionary.
    :return: Substitution set or None if unification fails.
    """
    if subst is None:
        return None  # Unification failed
    elif expr1 == expr2:
        return subst  # Expressions are already the same
    elif is_variable(expr1):
        return unify_variable(expr1, expr2, subst)
    elif is_variable(expr2):
        return unify_variable(expr2, expr1, subst)
    elif is_compound(expr1) and is_compound(expr2):
        if get_predicate(expr1) != get_predicate(expr2):
            return None  # Different predicates, cannot unify
        return unify(get_arguments(expr1), get_arguments(expr2), subst)
    elif isinstance(expr1, list) and isinstance(expr2, list):
        if len(expr1) != len(expr2):
            return None  # Different number of arguments
        if not expr1:  # Both lists are empty
            return subst
        return unify(expr1[1:], expr2[1:], unify(expr1[0], expr2[0], subst))
    else:
```

```
        return None  # No unification possible


def unify_variable(var, x, subst):
    """
    Handles variable unification.
    """
    if var in subst:
        return unify(subst[var], x, subst)  # Recursive unification
    elif x in subst:
        return unify(var, subst[x], subst)
    elif occurs_check(var, x):
        return None  # Failure: cyclic dependency
    else:
        subst[var] = x
        return subst


def is_variable(x):
    """
    Checks if x is a variable.
    Variables are usually single lowercase letters.
    """
    return isinstance(x, str) and x.islower()


def is_compound(x):
    """
    Checks if x is a compound expression (predicate with arguments).
    """
    return isinstance(x, str) and "(" in x and ")" in x


def get_predicate(expr):
    """
    Extracts the predicate from a compound expression.
    """
    return expr.split("(")[0]


def get_arguments(expr):
    """
    Extracts the arguments from a compound expression as a list.
    """
    args = expr[expr.index("(") + 1:expr.rindex(")")]
    return [arg.strip() for arg in args.split(",")]
```

```
def occurs_check(var, x):
    """
    Checks if var occurs in x (prevents infinite recursion).
    """
    if var == x:
        return True
    elif is_compound(x):
        return var in get_arguments(x)
    elif isinstance(x, list):
        return any(occurs_check(var, arg) for arg in x)
    return False


def user_input_unification():
    """
    Function to get user input and perform unification.
    """
    print("=== Unification Algorithm ===")
    expr1 = input("Enter the first expression (e.g., P(x, y)): ")
    expr2 = input("Enter the second expression (e.g., P(a, b)): ")

    # Perform unification
    result = unify(expr1, expr2)

    # Print the result
    if result is None:
        print("Unification Failed")
    else:
        print("Unification Successful!")
        print("Substitution Set:", result)


# Run the program
if __name__ == "__main__":
    user_input_unification()
```

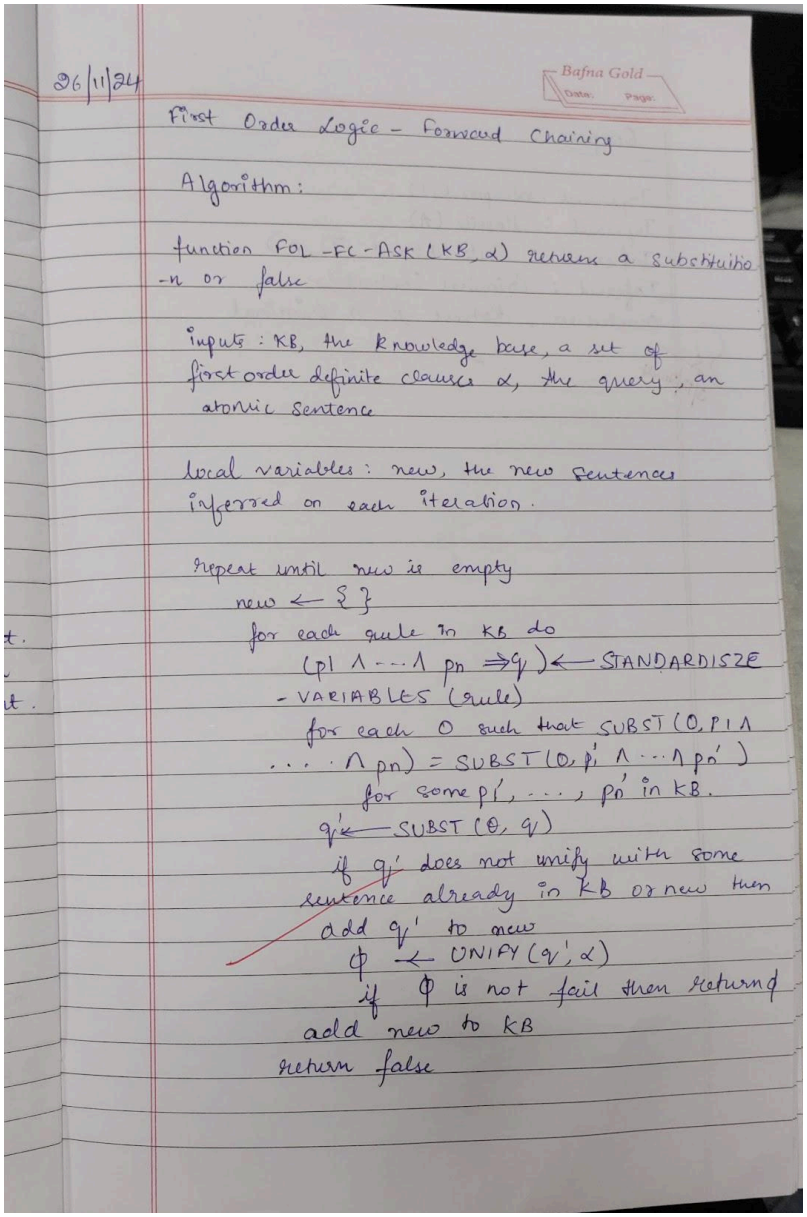
=== Unification Algorithm ===

Output:

Enter the first expression (e.g., P(x, y)): P(x,y)
Enter the second expression (e.g., P(a, b)): P(a)
Unification Failed

Program 8:

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



26/11/24

Bafna Gold

First Order Logic - Forward Chaining

Algorithm:

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

inputs: KB, the knowledge base, a set of first order definite clauses α, the query, an atomic sentence

local variables: new, the new sentences inferred on each iteration.

repeat until new is empty
new ← { }
for each rule in KB do
(p1 ∧ ... ∧ pn ⇒ q) ← STANDARDIZE-VARIABLES (rule)
for each θ such that SUBST(θ, p1 ∧ ... ∧ pn) = SUBST(θ, p1' ∧ ... ∧ pn')
for some p1', ..., pn' 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 q
add new to KB
return false

 # Define the knowledge base as a list of rules and facts

```
class KnowledgeBase:
    def __init__(self):
        self.facts = set()   # Set of known facts
        self.rules = []      # List of rules
```

```
    def add_fact(self, fact):
        self.facts.add(fact)

    def add_rule(self, rule):
        self.rules.append(rule)

    def infer(self):
        inferred = True
        while inferred:
            inferred = False
            for rule in self.rules:
                if rule.apply(self.facts):
                    inferred = True

# Define the Rule class
class Rule:
    def __init__(self, premises, conclusion):
        self.premises = premises  # List of conditions
        self.conclusion = conclusion  # Conclusion to add if premises are met

    def apply(self, facts):
        if all(premise in facts for premise in self.premises):
            if self.conclusion not in facts:
                facts.add(self.conclusion)
                print(f"Inferred: {self.conclusion}")
                return True
        return False

# Initialize the knowledge base
kb = KnowledgeBase()

# Facts in the problem
kb.add_fact("American(Robert)")
kb.add_fact("Missile(T1)")
kb.add_fact("Owns(A, T1)")
kb.add_fact("Enemy(A, America)")

# Rules based on the problem
# 1. Missile(x) implies Weapon(x)
kb.add_rule(Rule(["Missile(T1)"], "Weapon(T1)"))

# 2. Enemy(x, America) implies Hostile(x)
kb.add_rule(Rule(["Enemy(A, America)"], "Hostile(A)"))

# 3. Missile(x) and Owns(A, x) imply Sells(Robert, x, A)
kb.add_rule(Rule(["Missile(T1)", "Owns(A, T1)"], "Sells(Robert, T1, A)"))
```

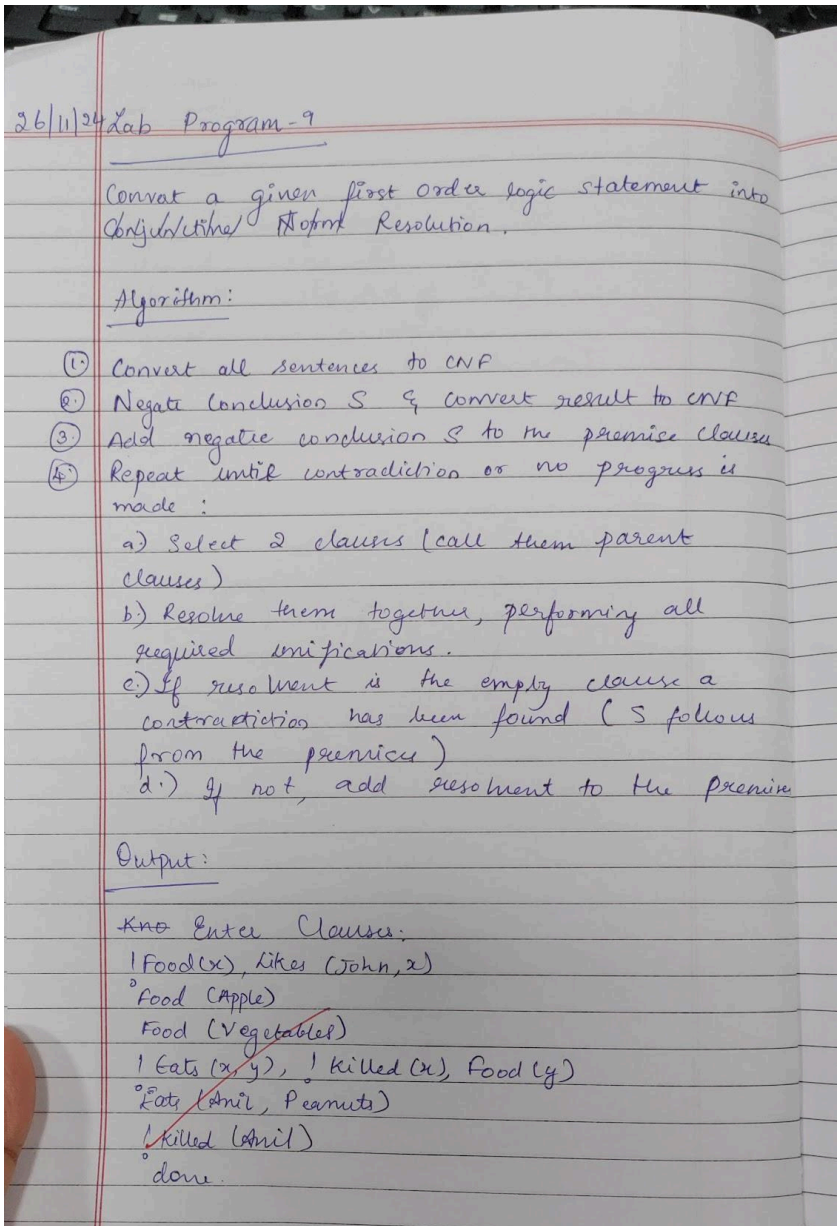
```
# 4. American(p) and Weapon(q) and Sells(p, q, r) and Hostile(r) imply
Criminal(p)
kb.add_rule(Rule(["American(Robert)", "Weapon(T1)", "Sells(Robert, T1, A)",
"Hostile(A)"], "Criminal(Robert)"))

# Infer new facts based on the rules
kb.infer()

# Check if Robert is a criminal
if "Criminal(Robert)" in kb.facts:
    print("Conclusion: Robert is a criminal.")
else:
    print("Conclusion: Unable to prove Robert is a criminal.")
```

Output:

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

Program 9:

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



26/11/24 Lab Program-9

Convert a given first order logic statement into Conjunctive Normal Form Resolution.

Algorithm:

1. Convert all sentences to CNF
2. Negate conclusion S & convert result to CNF
3. Add negate conclusion S to the premise clauses
4. Repeat until contradiction or no progress is made :
   a) Select 2 clauses (call them parent clauses)
   b) Resolve them together, performing all required unifications.
   c) If resolvent is the empty clause a contradiction has been found (S follows from the premises)
   d) If not, add resolvent to the premise

Output:

Enter Clauses:
!Food(x), Likes(John, x)
Food(Apple)
Food(Vegetables)
!Eats(x, y), !Killed(x), Food(y)
Eats(Anil, Peanuts)
!Killed(Anil)
done.

```
# Step 1: Helper function to parse user inputs into clauses (with '!' for negation)
def parse_clause(clause_input):
    """
    Parses a user input string into a tuple of literals for the clause.
    Replaces '!' with '¬' for negation handling.
    Example: "!Food(x), Likes(John, x)" -> ("¬Food(x)", "Likes(John, x)")
    """
    return tuple(literal.strip().replace("!", "¬") for literal in clause_input.split(","))
```

```
# Step 2: Collect knowledge base (KB) from user
def get_knowledge_base():
    print("Enter the premises of the knowledge base, one by one.")
    print("Use ',' to separate literals in a clause. Use '!' for negation.")
    print("Example: !Food(x), Likes(John, x)")
    print("Type 'done' when finished.")

    kb = []
    while True:
        clause_input = input("Enter a clause (or 'done' to finish): ").strip()
        if clause_input.lower() == "done":
            break
        kb.append(parse_clause(clause_input))

    return kb

# Step 3: Add negated conclusion
def get_negated_conclusion():
    print("\nEnter the conclusion to be proved.")
    print("It will be negated automatically for proof by contradiction.")
    conclusion = input("Enter the conclusion (e.g., Likes(John, Peanuts)): ").strip()
    negated = f"!{conclusion}" if not conclusion.startswith("!") else conclusion[1:]
    return (negated.replace("!", "¬"),)  # Convert '!' to '¬' for consistency

# Step 4: Resolution algorithm
def resolve(clause1, clause2):
    """
    Resolves two clauses and returns the resolvent (new clause).
    If no resolvable literals exist, returns an empty set.
    """
    resolved = set()
    for literal in clause1:
        if "¬" + literal in clause2:
            temp1 = set(clause1)
            temp2 = set(clause2)
            temp1.remove(literal)
            temp2.remove("¬" + literal)
            resolved = temp1.union(temp2)
        elif literal.startswith("¬") and literal[1:] in clause2:
            temp1 = set(clause1)
            temp2 = set(clause2)
            temp1.remove(literal)
            temp2.remove(literal[1:])
            resolved = temp1.union(temp2)
    return tuple(resolved)
```

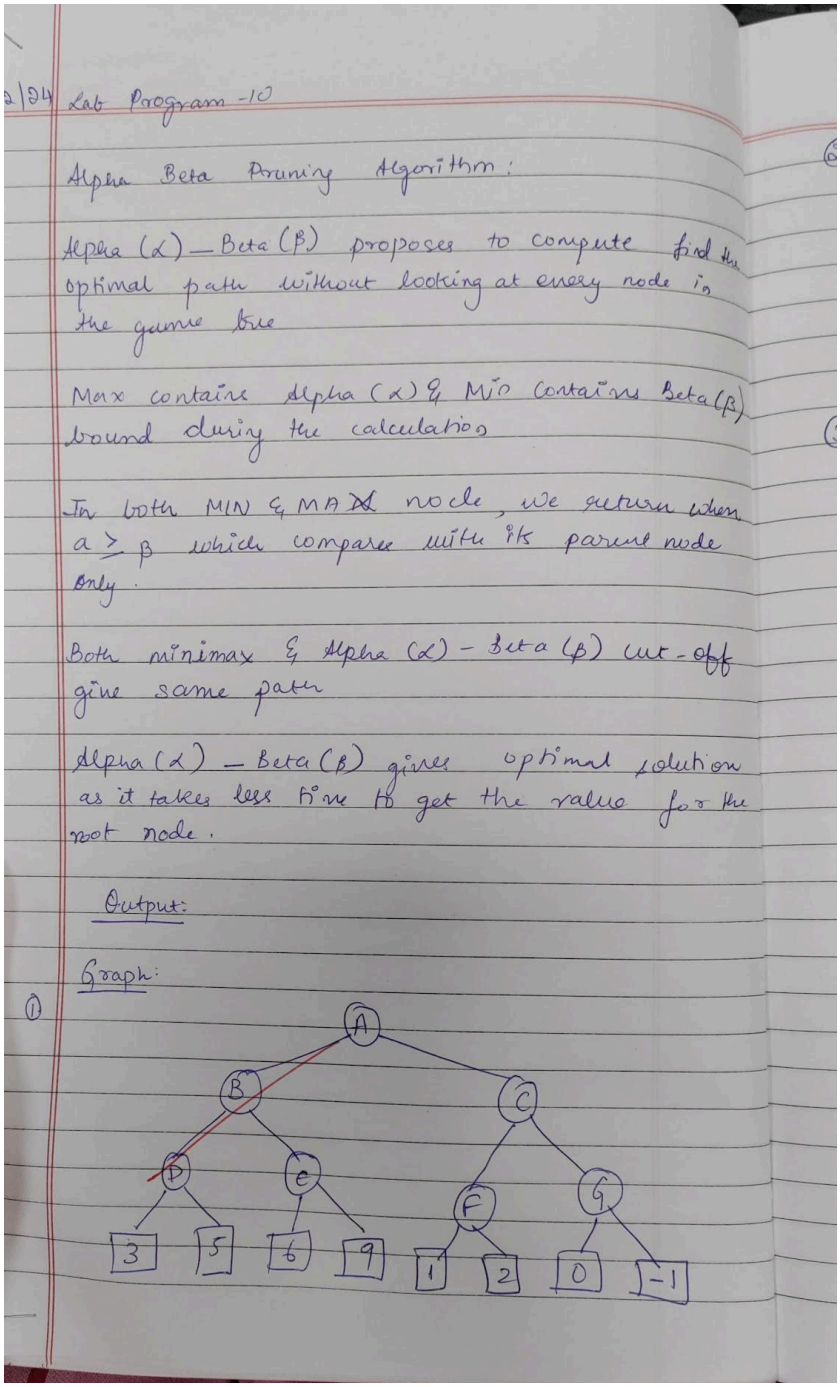
```
def resolution(kb):
    """
    Runs the resolution algorithm on the knowledge base (KB).
    Returns True if an empty clause is derived (proving the conclusion),
    or False if resolution fails.
    """
    clauses = set(kb)
    new = set()

    while True:
        # Generate all pairs of clauses
        pairs = [(ci, cj) for ci in clauses for cj in clauses if ci != cj]

        for (ci, cj) in pairs:
            resolvent = resolve(ci, cj)
            if not resolvent:  # Empty clause found
                return True
            new.add(resolvent)

        if new.issubset(clauses):  # No new clauses
            return False
        clauses = clauses.union(new)

# Step 5: Main execution
if __name__ == "__main__":
    print("=== Resolution Proof System ===")
    print("Provide the knowledge base and conclusion to prove.")

    # Collect user inputs
    kb = get_knowledge_base()
    negated_conclusion = get_negated_conclusion()
    kb.append(negated_conclusion)

    # Show the knowledge base
    print("\nKnowledge Base (KB):")
    for clause in kb:
        print("  ", " ∨ ".join(clause))  # Join literals with OR for readability

    # Perform resolution
    print("\nStarting resolution process...")
    result = resolution(kb)
    if result:
        print("\nProof complete: The conclusion is TRUE.")
    else:
        print("\nResolution failed: The conclusion could not be proved.")
```

Output:

```
=== Resolution Proof System ===
Provide the knowledge base and conclusion to prove.
Enter the premises of the knowledge base, one by one.
Use ',' to separate literals in a clause. Use '!' for negation.
Example: !Food(x), Likes(John, x)
Type 'done' when finished.
Enter a clause (or 'done' to finish): !Food(x), Likes(John,x)
Enter a clause (or 'done' to finish): Food(Apple)
Enter a clause (or 'done' to finish): Food(Vegetables)
Enter a clause (or 'done' to finish): !Eats(x,y), !Killed(x), Food(y)
Enter a clause (or 'done' to finish): Eats(Anil, Peanuts)
Enter a clause (or 'done' to finish): !Killed(Anil)
Enter a clause (or 'done' to finish): done

Enter the conclusion to be proved.
It will be negated automatically for proof by contradiction.
Enter the conclusion (e.g., Likes(John, Peanuts)): Likes(John, Peanuts)

Knowledge Base (KB):
  ¬Food(x) ∨ Likes(John ∨ x)
  Food(Apple)
  Food(Vegetables)
  ¬Eats(x ∨ y) ∨ ¬Killed(x) ∨ Food(y)
  Eats(Anil ∨ Peanuts)
  ¬Killed(Anil)
  ¬Likes(John, Peanuts)

Starting resolution process...

Proof complete: The conclusion is TRUE.
```

Program 10:

Implement Alpha-Beta Pruning:



 # Python3 program to demonstrate
 # working of Alpha-Beta Pruning

 # Initial values of Alpha and Beta
MAX, MIN = 1000, -1000

```
# Returns optimal value for current player
#(Initially called for root and maximizer)
def minimax(depth, nodeIndex, maximizingPlayer,
                    values, alpha, beta):

    # Terminating condition. i.e
    # leaf node is reached
    if depth == 3:
        return values[nodeIndex]

    if maximizingPlayer:

        best = MIN

        # Recur for left and right children
        for i in range(0, 2):

            val = minimax(depth + 1, nodeIndex * 2 + i,
                                False, values, alpha, beta)
            best = max(best, val)
            alpha = max(alpha, best)

            # Alpha Beta Pruning
            if beta <= alpha:
                break

        return best

    else:
        best = MAX

        # Recur for left and
        # right children
        for i in range(0, 2):

            val = minimax(depth + 1, nodeIndex * 2 + i,
                                        True, values, alpha, beta)
            best = min(best, val)
            beta = min(beta, best)

            # Alpha Beta Pruning
            if beta <= alpha:
                break

        return best
```

```
# Driver Code
if __name__ == "__main__":

    values = [3, 5, 6, 9, 1, 2, 0, -1]
    print("The optimal value is :", minimax(0, 0, True, values, MIN, MAX))
```

Output:

The optimal value is : 5