VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB REPORT
on

Artificial Intelligence (23CS5PCAIN)

Submitted by

Dipesh Sah (1BM22CS092)

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 **Dipesh Sah (1BM22CS092),** who is a 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.

Pradeep Sadanand Assistant Professor Department of CSE, BMSCE	Dr Kavitha Sooda Professor & HOD Department of CSE, BMSCE

Index

Github Link: https://github.com/Deep-Sah-Dipesh/Artificial-Intelligence-Lab

Lab-1 **01/10/2024**

1. Implement Tic –Tac –Toe Game.

ALGORITHM

Bafna Gold
02-Oct
Date: 2024 Page: 01

Lab - 1

1) Vaccum Cleaner
Algorithm:

1. Initialize goal state:
   - Set goal_state = {'A': '0', 'B': '0'}
     (0 = clean, 1 = dirty)
   - Initialize cost = 0.

2. Input the vaccum location:
   - Get vaccum's location:
     location_input = 'A' or 'B'.
   - Get status of the current room:
     status_input = '0' or '1'.
   - Get status of the other room:
     status_input_complement = '0' or '1'.

3. Vaccum in Room A:
   - If location_input == 'A':
     - If status_input == '1' (A is dirty):
       - clean A, set goal_state['A'] = '0',
         increment cost.
     - If status_input_complement == '1'
       (B is dirty):
       - Move to B, clean it.
       - update goal_state['B'] = '0'.
       - increment cost.

4. Vaccum in Room B:
   - If location_input == 'B':
     - If status_input == '1' (B is dirty):

- Clean B, set goal_state['B'] = '0',
- increment cost.

- If status_input_complement == '1' (A is dirty):
  - Move to A, clean it
  - update goal_state['A'] = '0'.
  - increment cost.

5. Output:
   - print goal_state and cost as the performance measure.

Output:- Initial condition: {'A': '1', 'B': '1'}
Vaccum 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 location B.
Cost for moving Right: 2
Cost for cleaning: 3
Location B has been cleaned.

GOAL State:
{'A': '0', 'B': '0'}
Performance Measurement: 3

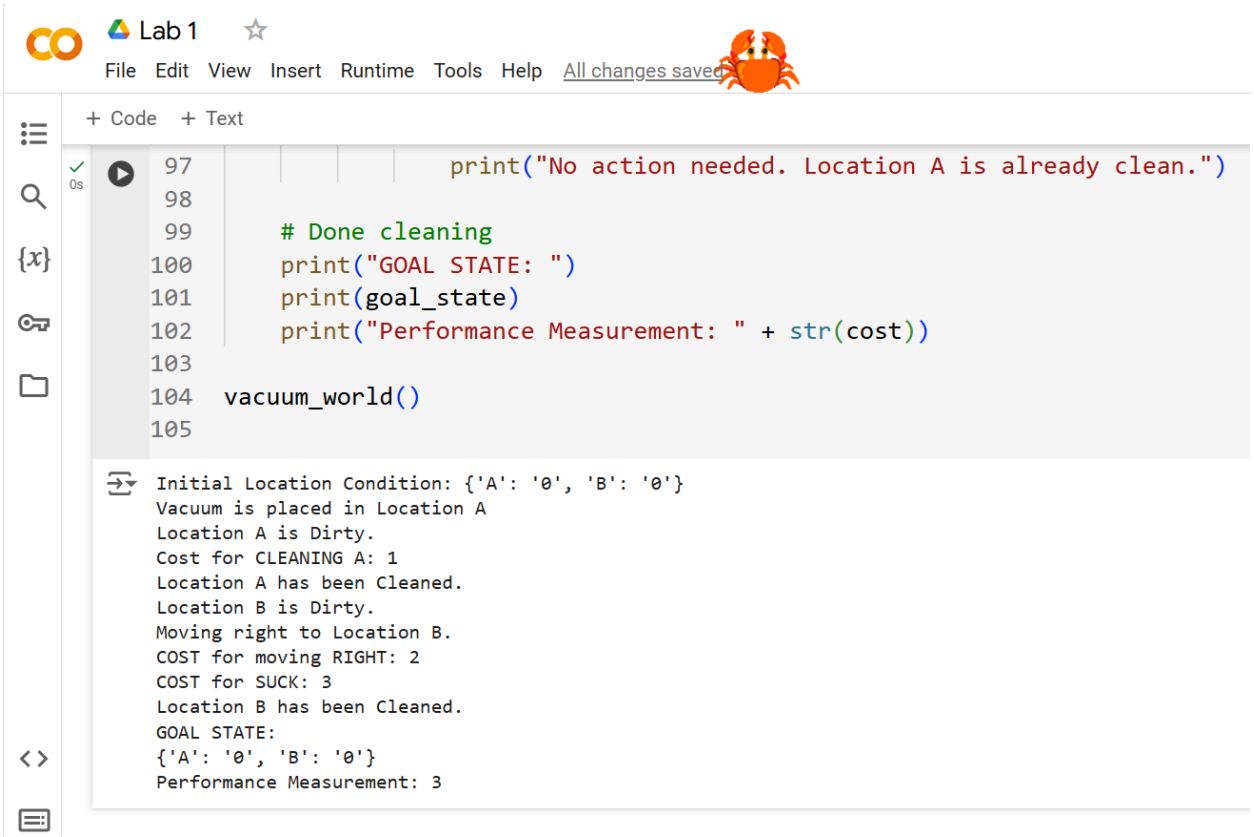
CODE:

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

    # Simulate user input
    location_input = 'A'  # Replace with 'A' or 'B'
    status_input = '1'  # Replace with '0' or '1' (0=Clean, 1=Dirty)
    status_input_complement = '1'  # Replace with '0' or '1' (0=Clean, 1=Dirty)

    print("Initial Location Condition: " + str(goal_state))

    if location_input == 'A':
        # Vacuum is in Location A
        print("Vacuum is placed in Location A")

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

            if status_input_complement == '1':
                # If Location B is Dirty
                print("Location B is Dirty.")
                print("Moving right to 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 cleaning
                print("COST for SUCK: " + str(cost))
                print("Location B has been Cleaned.")
            else:
                print("No action needed. Location B is already clean.")
        else:
            print("Location A is already clean.")

            if status_input_complement == '1':
                # If Location B is Dirty
```

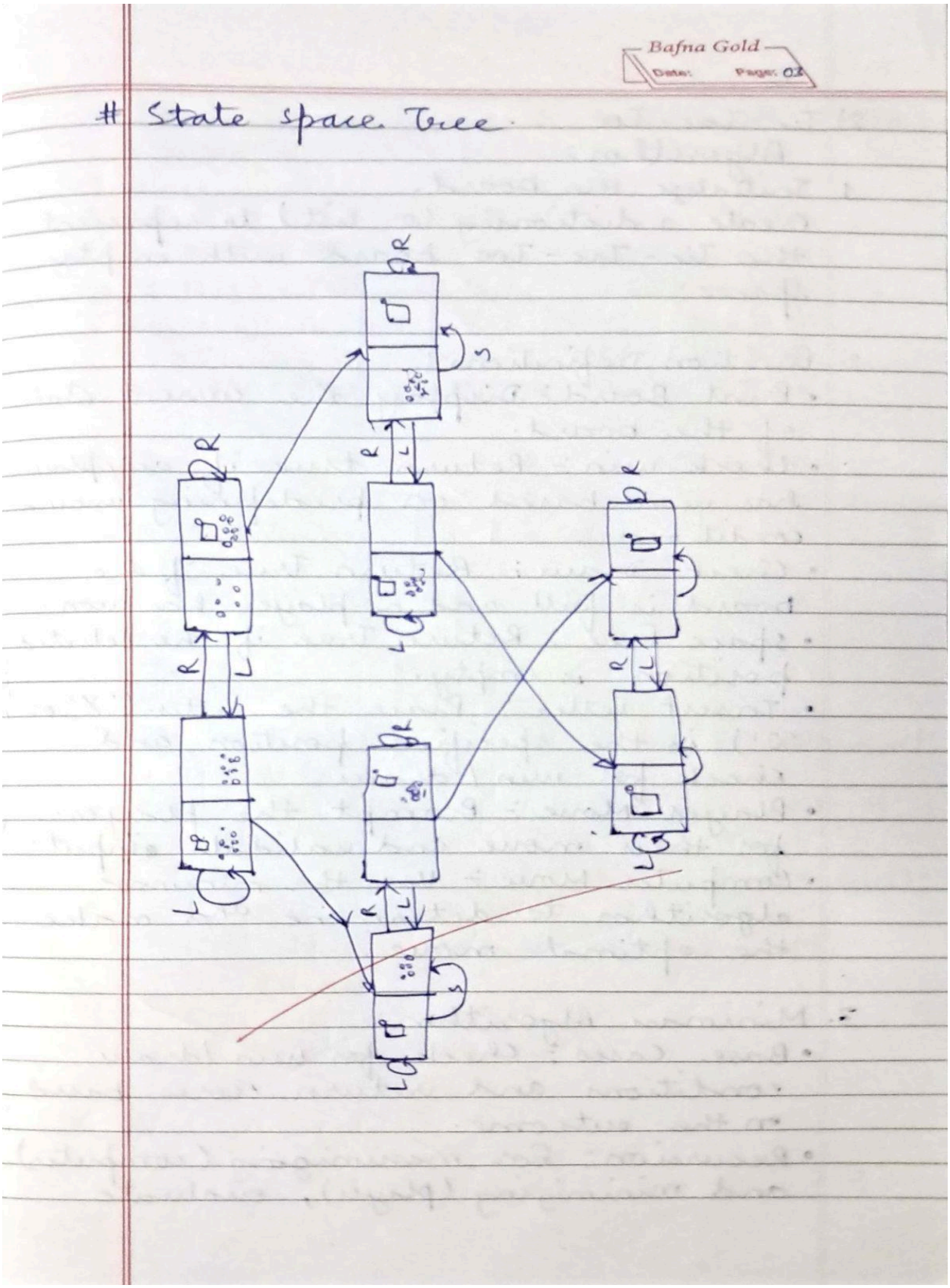
```
            print("Location B is Dirty.")
            print("Moving right to Location B.")
            cost += 1  # Cost for moving right
            print("COST for moving RIGHT: " + str(cost))
            # Suck the dirt and mark it as clean
            goal_state['B'] = '0'
            cost +=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

else:
    # Vacuum is in Location B
    print("Vacuum is placed in Location B")

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

        if status_input_complement == '1':
            # If Location A is Dirty
            print("Location A is Dirty.")
            print("Moving LEFT to Location A.")
            cost += 1  # Cost for moving left
            print("COST for moving LEFT: " + str(cost))
            # Suck the dirt and mark it as clean
            goal_state['A'] = '0'
            cost += 1  # Cost for cleaning
            print("COST for SUCK: " + str(cost))
            print("Location A has been Cleaned.")
        else:
            print("No action needed. Location A is already clean.")
    else:
        print("Location B is already clean.")

        if status_input_complement == '1':
            # If Location A is Dirty
            print("Location A is Dirty.")
```

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

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

vacuum_world()
```

OUTPUT:



STATE-SPACE TREE:



Lab-1 **01/10/2024**

2. Implement Vacuum Cleaner

ALGORITHM:

2) Tic - Tac - Toe.

Algorithm:

1. Initialize the board.
Create a dictionary (or list) to represent the Tic - Tac - Toe board with empty spaces.

2. Function Definitions:
- Print Board: Display the current state of the board.
- check win: Return true if any player has won based on predefining wining conditions.
- Check Draw: Return True if the board is full and no player has won.
- Space Free: Return True if the selected position is empty.
- Insert letter: Place the letter ('X' or 'O') in the specified position and check for win / draw.
- Player Move: Prompt the player for their move and validate input.
- Computer Move: Use the minimax algorithm to determine and make the optimal move.

3. Minimax Algorithm:
- Base Cases: Check for win / draw conditions and return scores based on the outcome.
- Recursion: For maximizing (computer) and minimizing (player), evaluate

possible moves and return the best score.

4. Main Game Loop.
   - While the game is ongoing:
     - If no win/draw, call computer Move.
     - Check for win/draw conditions.
     - If no win/draw, call Player move.
     - Check for win/draw conditions.

5. End Game:
   - Display the final board and announce the result (win/draw).

Pseudocode :-

```
Initialize Board
While true:
    If not checkWin() && not checkDraw();
        compMove()
    If checkWin() or checkDraw();
        break;
    If not checkWin() && not checkDraw();
        playerMove()
    If checkWin() or checkDraw():
        break

Display final board.
Announce result.
```

CODE:

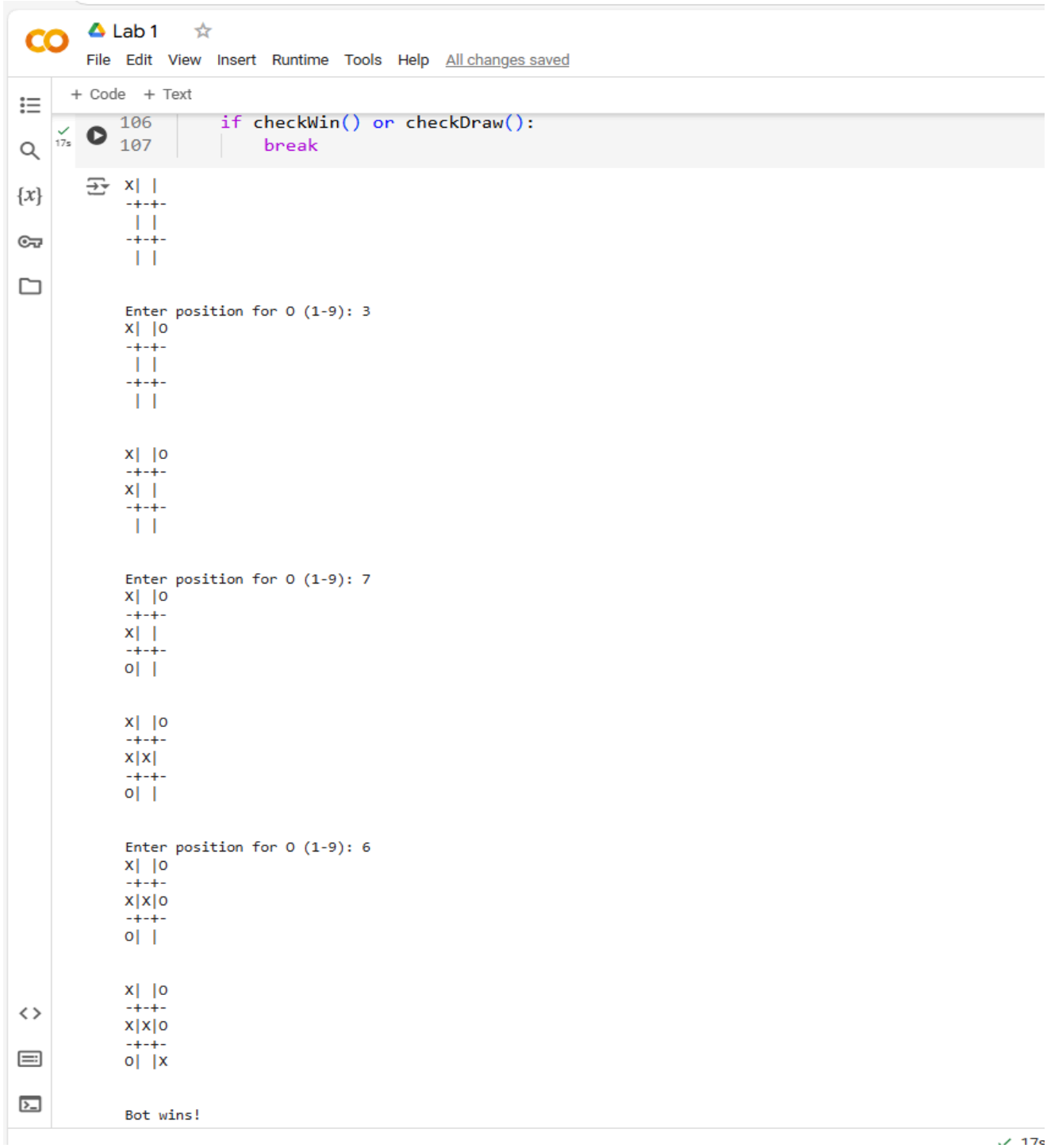
```
# Tic-Tac-Toe Board
board = {1: ' ', 2: ' ', 3: ' ',
        4: ' ', 5: ' ', 6: ' ',
        7: ' ', 8: ' ', 9: ' '}

# Function to print the board
def printBoard(board):
    print(board[1] + '|' + board[2] + '|' + board[3])
    print('-+-+-')
    print(board[4] + '|' + board[5] + '|' + board[6])
    print('-+-+-')
    print(board[7] + '|' + board[8] + '|' + board[9])
    print('\n')

def spaceFree(pos):
    return board[pos] == ' '

# Check if there is a win
def checkWin():
    win_conditions = [(1, 2, 3), (4, 5, 6), (7, 8, 9), (1, 4, 7), (2, 5, 8), (3, 6, 9), (1, 5, 9), (3, 5, 7)]
    for condition in win_conditions:
        if board[condition[0]] == board[condition[1]] == board[condition[2]] != ' ':
            return True
    return False

# Check if it's a draw
def checkDraw():
    return all(space != ' ' for space in board.values())

# Insert the letter into the board
def insertLetter(letter, position):
    if spaceFree(position):
        board[position] = letter
```

```
        printBoard(board)
        if checkWin():
            if letter == 'X':
                print('Bot wins!')
            else:
                print('You win!')
            return True  # Return True if someone won
        if checkDraw():
            print('Draw!')
            return True  # Return True if it's a draw
        return False
    else:
        print('Position taken, please pick a different position.')
        return False

def playerMove():
    position = int(input('Enter position for O (1-9): '))
    while not spaceFree(position):
        position = int(input('Enter valid position for O (1-9): '))
    insertLetter('O', position)

# Computer's move using minimax algorithm
def compMove():
    bestScore = -1000
    bestMove = 0
    for key in board.keys():
        if board[key] == ' ':
            board[key] = 'X'
            score = minimax(board, False)
            board[key] = ' '
            if score > bestScore:
                bestScore = score
                bestMove = key
    insertLetter('X', bestMove)
```

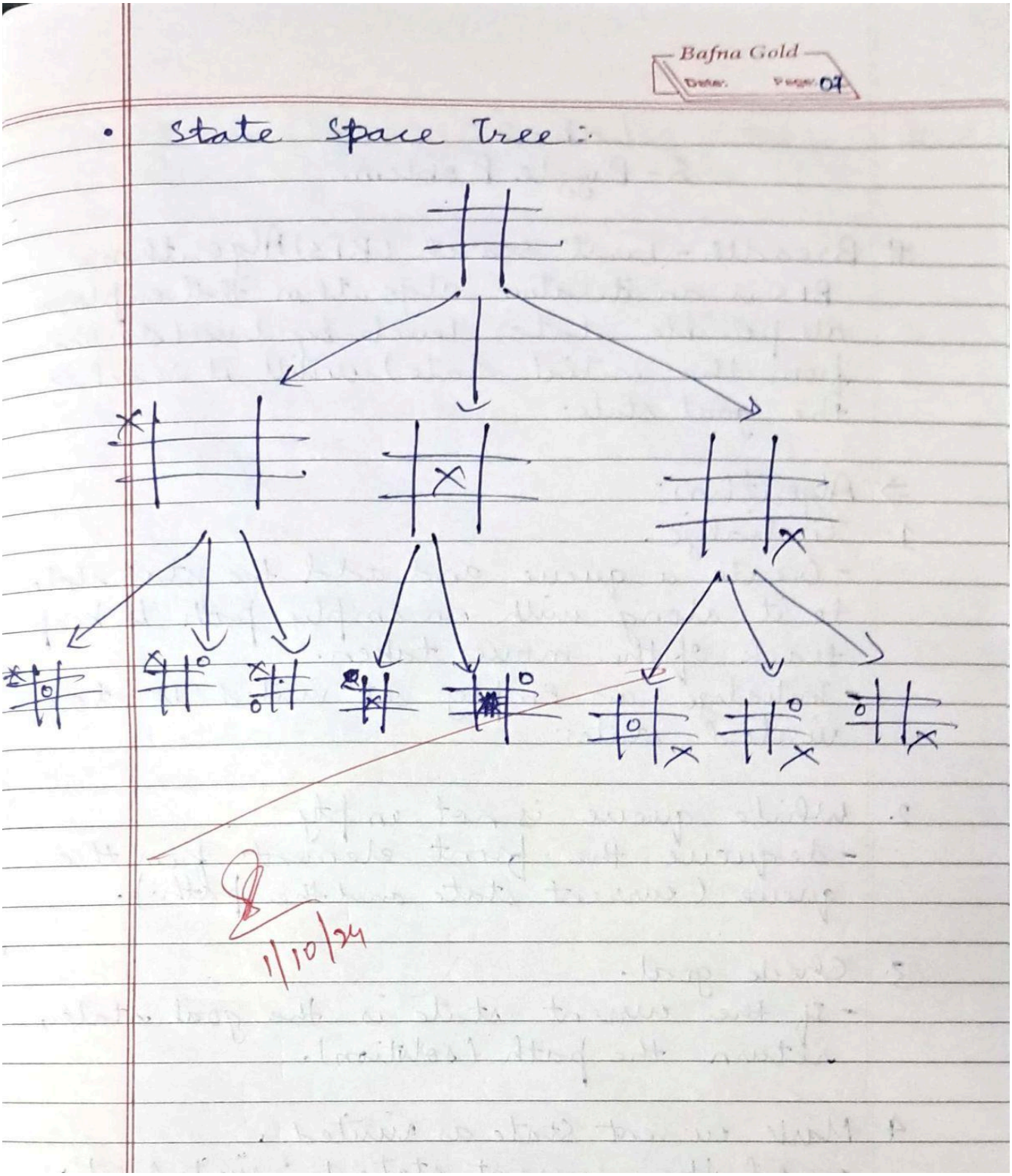
```
# Minimax algorithm to decide the computer's move
def minimax(board, isMaximizing):
    if checkWin():
        if isMaximizing:
            return -1
        else:
            return 1
    elif checkDraw():
        return 0

    if isMaximizing:
        bestScore = -1000
        for key in board.keys():
            if board[key] == ' ':
                board[key] = 'X'
                score = minimax(board, False)
                board[key] = ' '
                bestScore = max(score, bestScore)
        return bestScore
    else:
        bestScore = 1000
        for key in board.keys():
            if board[key] == ' ':
                board[key] = 'O'
                score = minimax(board, True)
                board[key] = ' '
                bestScore = min(score, bestScore)
        return bestScore

# Main game loop
while True:
    if not checkWin() and not checkDraw():
        compMove()
```

```
if not checkWin() and not checkDraw():
    playerMove()
if checkWin() or checkDraw():
    break
```

OUTPUT:



STATE-SPACE TREE:



Lab-2 **08/10/2024**

3. Solve 8 puzzle problems using BFS

ALGORITHM:

Lab - 2

8 - Puzzle Problem.

Breadth - First Search (BFS) Algorithm.

BFS is an iterative algorithm that explores all possible states level by level (starting from the initial state) until it reaches the goal state.

⇒ Algorithm:-

1. Initialize.
   - Create a queue and add the start state to it, along with an empty path to keep track of the moves taken.
   - Initialize an empty set 'visited' to store visited states.

2. While queue is not empty:
   - dequeue the front element from the queue (current state and the path).

3. Check goal.
   - If the current state is the goal state, return the path (solution).

4. Mark current State as visited.
   - add the current state to 'visited set'.

5. Generate successors:
   - For each valid move from the current state:
     - apply the move to get a new state.
     - If the new state has not been visited:

- add the new state and the updated path to the queue.

6. Return Failure:
- If the queue is empty and no solution has been found, return failure (no solution).

⇒ Output:

Initial state:

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

Solving using BFS:

Exploring state in BFS:

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

Exploring state in BFS:

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

- BFS solution: ['down', 'right']

Exploring state in DFS:

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

CODE:

```
from collections import deque

# Define the goal state and the possible moves (up, down, left, right)
goal_state = [[1, 2, 3],
              [4, 5, 6],
              [7, 8, 0]]  # 0 represents the empty space

# Function to find the position of the empty space (0)
def find_zero(state):
    for i in range(3):
        for j in range(3):
            if state[i][j] == 0:
                return (i, j)

# Function to make a move by swapping the empty space with its neighbor
def move(state, direction):
    new_state = [row[:] for row in state]
    zero_pos = find_zero(state)
    i, j = zero_pos

    if direction == "up" and i > 0:
        new_state[i][j], new_state[i-1][j] = new_state[i-1][j], new_state[i][j]
    elif direction == "down" and i < 2:
        new_state[i][j], new_state[i+1][j] = new_state[i+1][j], new_state[i][j]
    elif direction == "left" and j > 0:
        new_state[i][j], new_state[i][j-1] = new_state[i][j-1], new_state[i][j]
    elif direction == "right" and j < 2:
        new_state[i][j], new_state[i][j+1] = new_state[i][j+1], new_state[i][j]
    else:
        return None

    return new_state

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

# Function to print the puzzle state
def print_state(state):
    for row in state:
        print(row)
    print("\n")
```

```
# BFS algorithm
def bfs(initial_state):
    queue = deque([(initial_state, [])])
    visited = set()

    while queue:
        state, path = queue.popleft()
        print("Exploring state in BFS:")
        print_state(state)  # Print the current state being explored

        if is_goal(state):
            return path

        visited.add(str(state))
        for direction in ["up", "down", "left", "right"]:
            new_state = move(state, direction)
            if new_state and str(new_state) not in visited:
                queue.append((new_state, path + [direction]))

    return None

initial_state = [[1, 2, 3],
                 [4, 0, 6],
                 [7, 5, 8]]  # Initial configuration of the 8-puzzle

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

# Solve with BFS
print("Solving using BFS:")
bfs_solution = bfs(initial_state)
if bfs_solution:
    print("BFS Solution:", bfs_solution)
else:
    print("No solution found with BFS.")
```

OUTPUT:

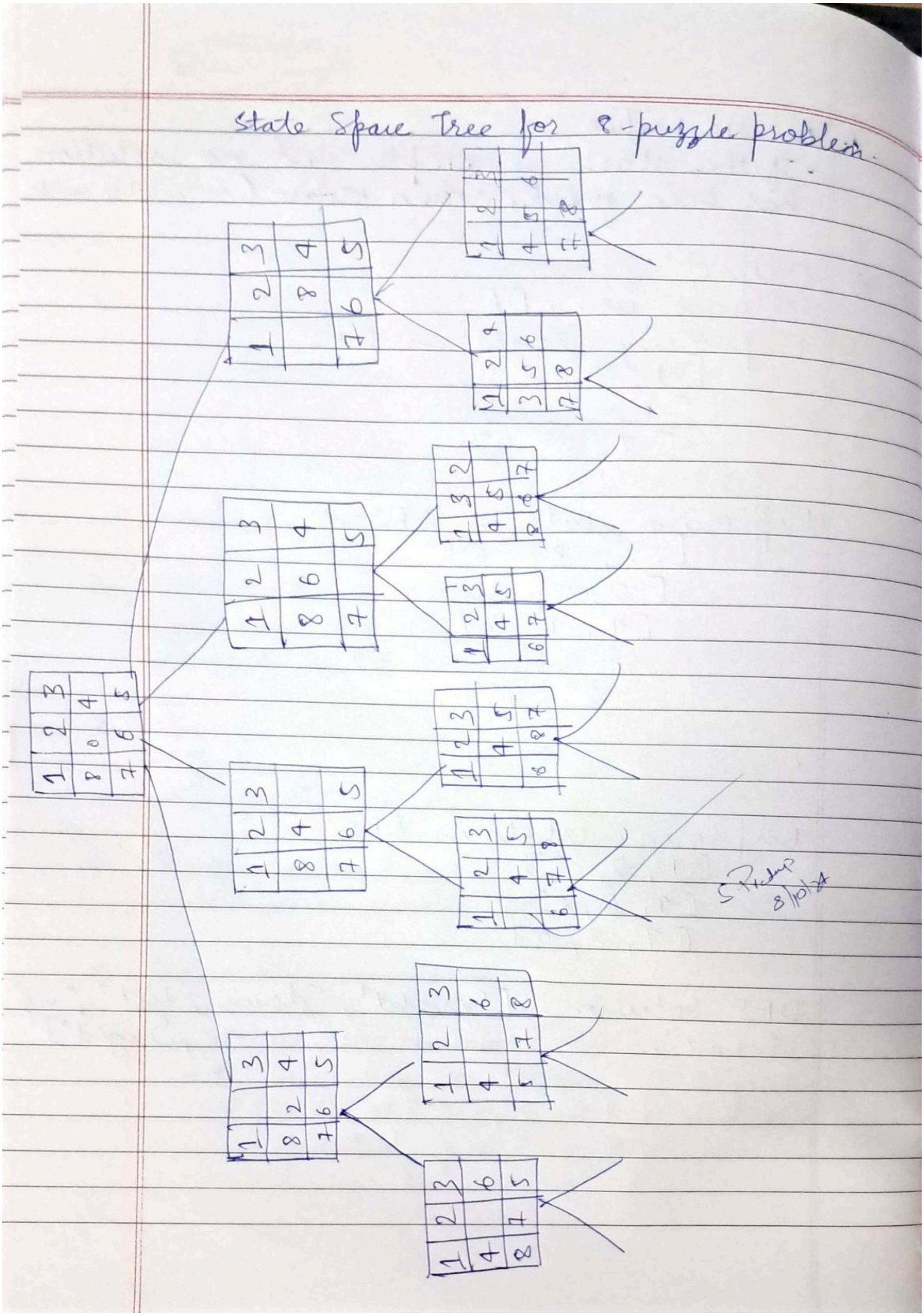
Output for 8 Puzzle Problem using BFS

```
else:
    print("No solution found with DFS.")
```

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


Solving using BFS:
Exploring state in BFS:
[1, 2, 3]
[4, 0, 6]
[7, 5, 8]


Exploring state in BFS:
[1, 0, 3]
[4, 2, 6]
[7, 5, 8]


Exploring state in BFS:
[1, 2, 3]
[4, 5, 6]
[7, 0, 8]


Exploring state in BFS:
[1, 2, 3]
[0, 4, 6]
[7, 5, 8]


Exploring state in BFS:
[1, 2, 3]
[4, 6, 0]
[7, 5, 8]
```

```
Exploring state in BFS:
[1, 2, 3]
[4, 6, 0]
[7, 5, 8]


Exploring state in BFS:
[0, 1, 3]
[4, 2, 6]
[7, 5, 8]


Exploring state in BFS:
[1, 3, 0]
[4, 2, 6]
[7, 5, 8]


Exploring state in BFS:
[1, 2, 3]
[4, 5, 6]
[0, 7, 8]


Exploring state in BFS:
[1, 2, 3]
[4, 5, 6]
[7, 8, 0]


BFS Solution: ['down', 'right']
Solving using DFS:
Exploring state in DFS:
[1, 2, 3]
[4, 0, 6]
[7, 5, 8]
```

STATE-SPACE TREE:



Lab-2 **08/10/2024**

4. Solve 8 puzzle problems using DFS

ALGORITHM:

8-puzzle problem using Depth-First Search Algorithm.
DFS explores a branch as far as possible before backtracking when it reaches a dead end (or goal).

⇒ Algorithm:

1. Initialize:
   - Create a stack and add the start state to it, along with an empty path to keep track of the moves taken.
   - Initialize an empty set 'visited' to store visited states.

2. While stack is not empty:
   - pop the top element from the stack (the current state and the path).

3. Check goal:
   - If the current state is the goal, return the path (solution).

4. Mark current state as visited:
   - add the current state to the 'visited set'.

5. Generate successors:
   - For each valid move from the current state:
     - apply the move to get a new state.
     - If the new state has not been visited:
       - push the new state and the updated path to the stack.

6. Return Failure:
- If the stack is empty and no solution has been found, return failure (no solution exists).

⇒ Output:
Solving using DFS:
Exploring state in DFS:
[1, 0, 3]
[4, 2, 6]
[7, 5, 8]
⋮ ⋮ ⋮

Exploring state in DFS:
[2, 3, 6]
[8, 0, 5]
[4, 1, 7]
⋮ ⋮ ⋮

Exploring state in DFS:
[1, 2, 3]
[4, 5, 6]
[7, 8, 0]

DFS solution: ['right', 'down', 'left', 'up', 'right', ... , 'right']

CODE:

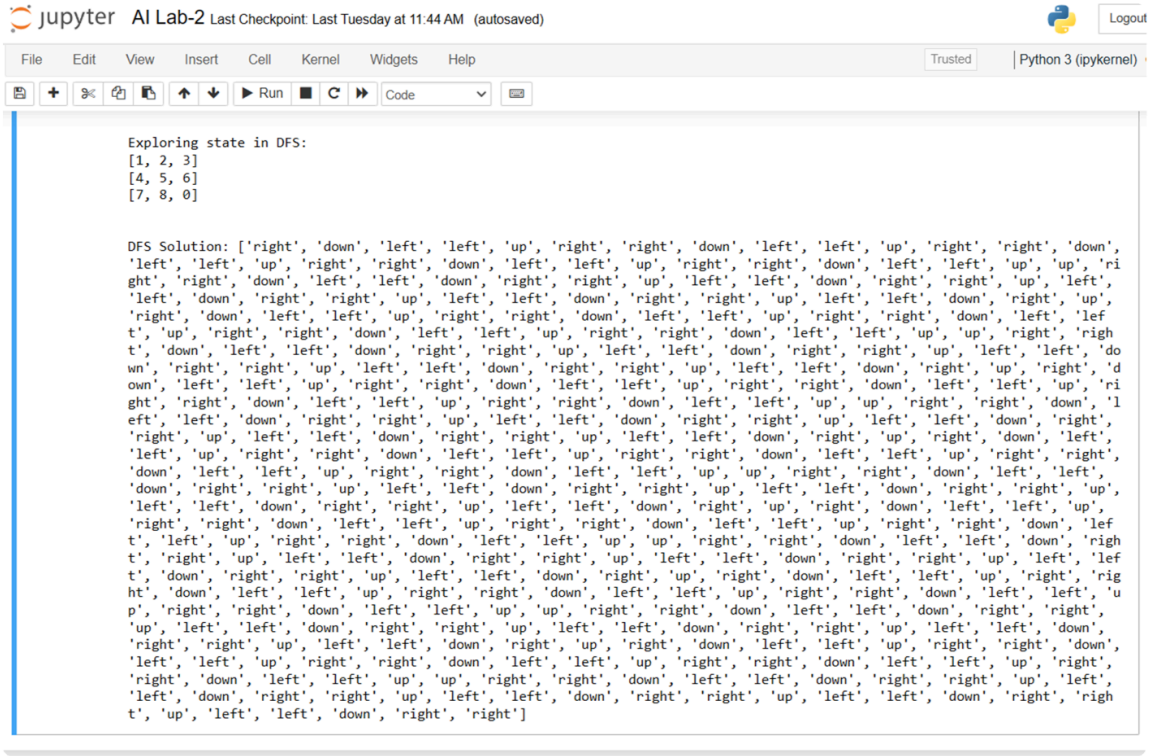
```
from collections import deque

# Define the goal state and the possible moves (up, down, left, right)
goal_state = [[1, 2, 3],
              [4, 5, 6],
              [7, 8, 0]]  # 0 represents the empty space

# Function to find the position of the empty space (0)
def find_zero(state):
    for i in range(3):
        for j in range(3):
            if state[i][j] == 0:
                return (i, j)

# Function to make a move by swapping the empty space with its neighbor
def move(state, direction):
    new_state = [row[:] for row in state]
    zero_pos = find_zero(state)
    i, j = zero_pos

    if direction == "up" and i > 0:
        new_state[i][j], new_state[i-1][j] = new_state[i-1][j], new_state[i][j]
    elif direction == "down" and i < 2:
        new_state[i][j], new_state[i+1][j] = new_state[i+1][j], new_state[i][j]
    elif direction == "left" and j > 0:
        new_state[i][j], new_state[i][j-1] = new_state[i][j-1], new_state[i][j]
    elif direction == "right" and j < 2:
        new_state[i][j], new_state[i][j+1] = new_state[i][j+1], new_state[i][j]
    else:
        return None

    return new_state

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

# Function to print the puzzle state
def print_state(state):
    for row in state:
        print(row)
    print("\n")
```

```
# DFS algorithm
def dfs(initial_state):
    stack = [(initial_state, [])]
    visited = set()

    while stack:
        state, path = stack.pop()
        print("Exploring state in DFS:")
        print_state(state)  # Print the current state being explored

        if is_goal(state):
            return path

        visited.add(str(state))
        for direction in ["up", "down", "left", "right"]:
            new_state = move(state, direction)
            if new_state and str(new_state) not in visited:
                stack.append((new_state, path + [direction]))

    return None


initial_state = [[1, 2, 3],
                 [4, 0, 6],
                 [7, 5, 8]]  # Initial configuration of the 8-puzzle

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

# Solve with DFS
print("Solving using DFS:")
dfs_solution = dfs(initial_state)
if dfs_solution:
    print("DFS Solution:", dfs_solution)
else:
    print("No solution found with DFS.")
```

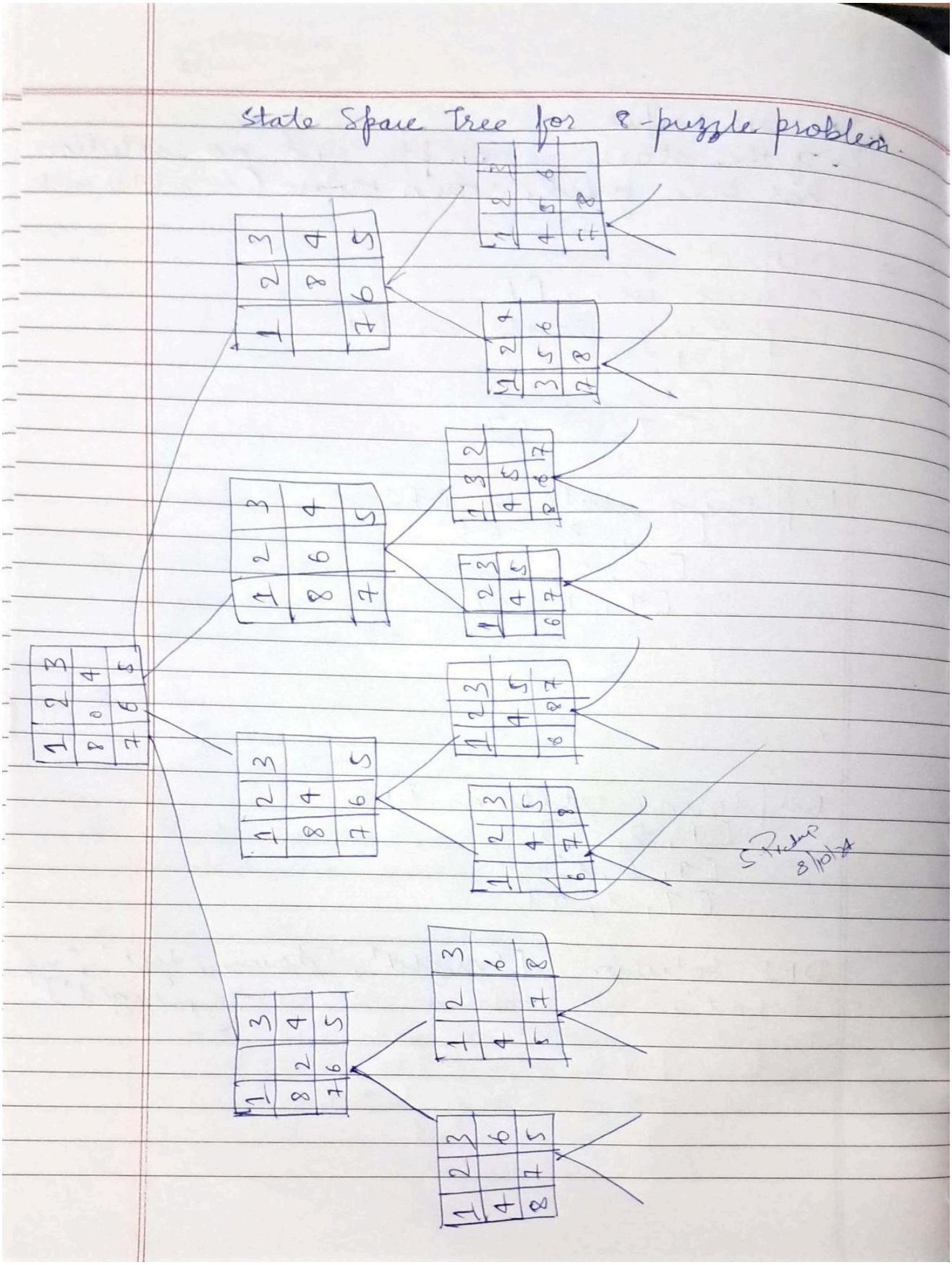
OUTPUT:

Output for 8 Puzzle Problem using DFS

```
Solving using DFS:
Exploring state in DFS:
[1, 2, 3]
[4, 0, 6]
[7, 5, 8]


Exploring state in DFS:
[1, 2, 3]
[4, 6, 0]
[7, 5, 8]


Exploring state in DFS:
[1, 2, 3]
[4, 6, 8]
[7, 5, 0]
```



STATE-SPACE TREE:



Lab-3 **15/10/2024**

5. Implement A* (A star) search algorithm for 8-puzzle Using Misplaced Tiles Heuristic.

Algorithm:

Bafna Gold
Date: 15 Oct 2024 Page: 13

Lab-3

Algorithm 1: A* search for 8-puzzle with Misplaced Tiles Heuristic.

1. Initialize
   Open list ← {start node}
   Closed list ← ∅
   g(start) ← 0
   h(start) ← number of misplaced tiles.
   f(start) ← g(start) + h(start)

2. Loop untill Open list is empty
   - Select the node n from the Open list with the lowest f(n).
   - Remove n from the Open list and add it to the Closed list.

3. For each neighbour of node n.
   - If the neighbour is in the Closed list, skip it.
   - Calculate g(neighbour) ← g(n) + 1 (Depth of node)
   - Calculate h(neighbour) ← number of misplaced tiles
   - Calculate f(neighbour) ← g(neighbour) + h(neighbour)
   - If the neighbour is not in the Open list, add it.
   - If the neighbour is not in the Open list, and this path is better (lower f(n)), update the path.

4. Repeat steps 2-3 untill the goal is reached or the Open list is empty.

Code:

```
import heapq
import numpy as np

class Node:
    def __init__(self, state, parent=None, g=0, h=0):
        self.state = state
        self.parent = parent
        self.g = g
        self.h = h
        self.f = g + h

  def __lt__(self, other):
        return self.f < other.f

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

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

def is_goal(state, goal_state):
    return np.array_equal(state, goal_state)

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

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

    for dx, dy in directions:
        new_x, new_y = x + dx, y + dy
        if 0 <= new_x < 3 and 0 <= new_y < 3:
            new_state = np.copy(node.state)
            new_state[x][y], new_state[new_x][new_y] = new_state[new_x][new_y], new_state[x][y]
            neighbors.append(new_state)
    return neighbors
```

```
def a_star_search(start_state, goal_state):
    open_list = []
    closed_set = set()

    start_node = Node(state=start_state, g=0, h=misplaced_tiles(start_state, goal_state))
    heapq.heappush(open_list, start_node)

    while open_list:
        current_node = heapq.heappop(open_list)

        if is_goal(current_node.state, goal_state):
            return reconstruct_path(current_node)

        closed_set.add(tuple(map(tuple, current_node.state)))

        for neighbor in get_neighbors(current_node):
            neighbor_tuple = tuple(map(tuple, neighbor))
            if neighbor_tuple in closed_set:
                continue

            g = current_node.g + 1
            h = misplaced_tiles(neighbor, goal_state)
            neighbor_node = Node(state=neighbor, parent=current_node, g=g, h=h)

            heapq.heappush(open_list, neighbor_node)
    return None

def reconstruct_path(node):
    path = []
    while node:
        path.append(node.state)
        node = node.parent
    return path[::-1]

def input_state(prompt):
    while True:
        try:
            print(prompt)
            state = list(map(int, input().split()))
            if len(state) != 9 or set(state) != set(range(9)):
                raise ValueError("Input must be 9 numbers from 0 to 8 with no duplicates.")
            return np.array(state).reshape((3, 3))
        except ValueError as e:
            print(f"Invalid input: {e}. Please try again.")
```

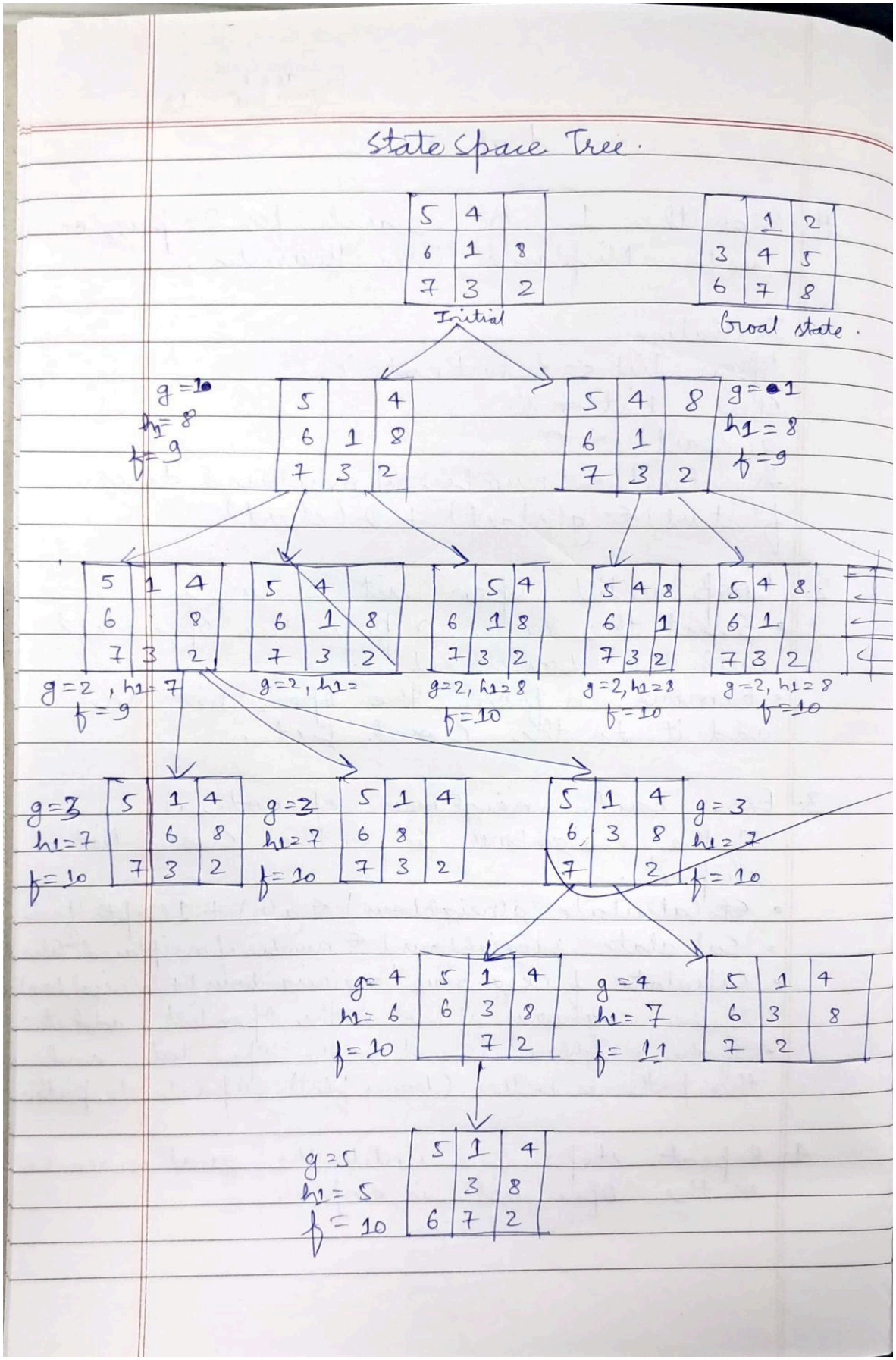
```
if __name__ == "__main__":
    start_state = input_state("Enter the start state (9 numbers, 0 for blank):")
    goal_state = input_state("Enter the goal state (9 numbers, 0 for blank):")

    print("Using Misplaced Tiles Heuristic:")
    solve_puzzle(start_state, goal_state)
```

Output:

```
96
97      print("Using Misplaced Tiles Heuristic:")
98      solve_puzzle(start_state, goal_state)
99
```

```
Enter the start state (9 numbers, 0 for blank):
5 4 0 6 1 8 7 3 2
Enter the goal state (9 numbers, 0 for blank):
0 1 2 3 4 5 6 7 8
Using Misplaced Tiles Heuristic:
Solution found with 24 moves!
[5 4 0]
[6 1 8]
[7 3 2]

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

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

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

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

[5 1 4]
```

State Space Diagram:



Lab-3 **15/10/2024**

6. Implement A* (A star) search algorithm for 8-puzzle Using Manhattan Distance Heuristic.

Algorithm:

Date: Page: 15

Algorithm 2: A* Search for 8-puzzle with Manhattan Distance Heuristic.

1. Initialize
   open list ← {start node}
   closed list ← ϕ
   g(start) ← 0
   h(start) ← Manhattan distance
   f(start) ← g(start) + h(start)

2. Loop until open list is empty.
   - Select the node n from the open list with the lowest f(n)
   - If n is the goal state/node, terminate and return the path.
   - Remove n from the open list and add it to the closed list.

3. For each neighbour of node n,
   - If the neighbour is in the closed list, skip it.
   - Calculate g(neighbour) ← g(n) + 1 (Depth of node)
   - Calculate h(neighbour) ← Manhattan Distance.
   - Calculate f(neighbour) ← g(neighbour) + h(neighbour)
   - If the neighbour is not in the open list, add it.
   - If the neighbour is in the open list and this path is better (lower f(n)), update the path.

4. Repeat steps 2-3 until the goal is reached or the open list is empty.

Code:

```
import heapq
import numpy as np

class Node:
    def __init__(self, state, parent=None, g=0, h=0):
        self.state = state
        self.parent = parent
        self.g = g
        self.h = h
        self.f = g + h

    def __lt__(self, other):
        return self.f < other.f

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

# Heuristic: Manhattan distance
def manhattan_distance(state, goal_state):
    distance = 0
    for i in range(3):
        for j in range(3):
            if state[i][j] != 0:
                x, y = divmod(state[i][j] - 1, 3)  # Expected position
                distance += abs(x - i) + abs(y - j)
    return distance

def is_goal(state, goal_state):
    return np.array_equal(state, goal_state)

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

def get_neighbors(node):
    neighbors = []
    x, y = find_blank(node.state)
    directions = [(-1, 0), (1, 0), (0, -1), (0, 1)]
```

```
    for dx, dy in directions:
        new_x, new_y = x + dx, y + dy
        if 0 <= new_x < 3 and 0 <= new_y < 3:
            new_state = np.copy(node.state)
            new_state[x][y], new_state[new_x][new_y] = new_state[new_x][new_y], new_state[x][y]
            neighbors.append(new_state)

    return neighbors

def a_star_search(start_state, goal_state):
    open_list = []
    closed_set = set()

    start_node = Node(state=start_state, g=0, h=manhattan_distance(start_state, goal_state))
    heapq.heappush(open_list, start_node)

    while open_list:
        current_node = heapq.heappop(open_list)

        if is_goal(current_node.state, goal_state):
            return reconstruct_path(current_node)

        closed_set.add(tuple(map(tuple, current_node.state)))

        for neighbor in get_neighbors(current_node):
            neighbor_tuple = tuple(map(tuple, neighbor))
            if neighbor_tuple in closed_set:
                continue

            g = current_node.g + 1
            h = manhattan_distance(neighbor, goal_state)
            neighbor_node = Node(state=neighbor, parent=current_node, g=g, h=h)

            heapq.heappush(open_list, neighbor_node)

    return None

def reconstruct_path(node):
    path = []
    while node:
        path.append(node.state)
        node = node.parent
    return path[::-1]
```

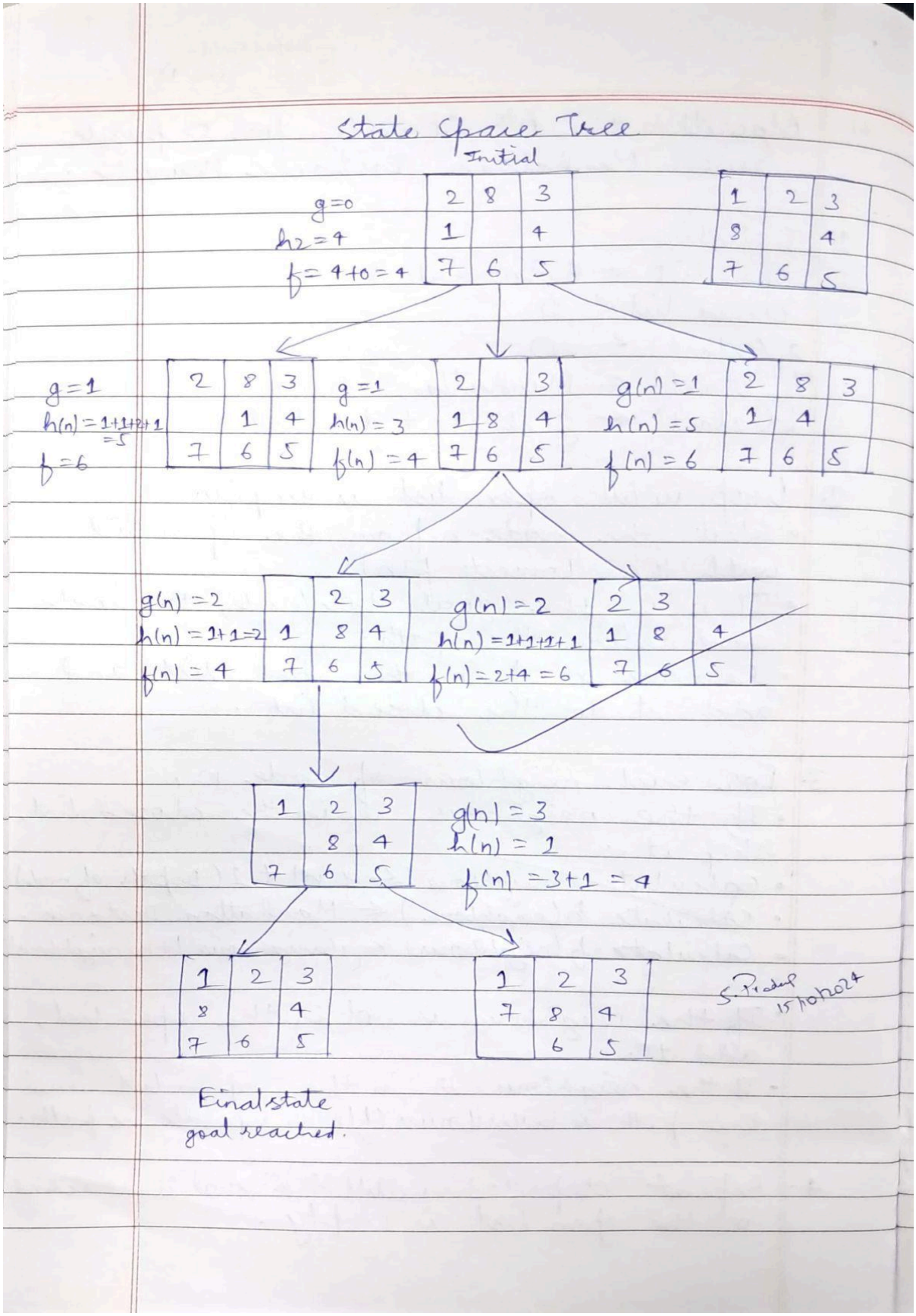
```
def input_state(prompt):
    while True:
        try:
            print(prompt)
            state = list(map(int, input().split()))
            if len(state) != 9 or set(state) != set(range(9)):
                raise ValueError("Input must be 9 numbers from 0 to 8 with no duplicates.")
            return np.array(state).reshape((3, 3))
        except ValueError as e:
            print(f"Invalid input: {e}. Please try again.")

if __name__ == "__main__":
    start_state = input_state("Enter the start state (9 numbers, 0 for blank):")
    goal_state = input_state("Enter the goal state (9 numbers, 0 for blank):")

    print("Using Manhattan Distance Heuristic:")
    solve_puzzle(start_state, goal_state)
```

Output:

```
102
103     print("Using Manhattan Distance Heuristic:
104     solve_puzzle(start_state, goal_state)
105
```

```
Enter the start state (9 numbers, 0 for blank):
2 8 3 1 0 4 7 6 5
Enter the goal state (9 numbers, 0 for blank):
1 2 3 8 0 4 7 6 5
Using Manhattan Distance Heuristic:
Solution found with 4 moves!
[2 8 3]
[1 0 4]
[7 6 5]

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

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

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

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

State Space Diagram:



Lab-4 **22/10/2024**

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

Algorithm:

Lab - 4

Algorithm for : Implement Hill Climbing Search algorithm to solve N-Queens problem.

1. Generate a random initial configuration of N queens on the board with one queen per column.

2. Evaluate the cost (heuristic cost) of the current configuration. This cost is the number of pairs of queens that are attacking each other.

3. Generate neighbours by moving each queen within its column to every other row and calculate the cost for each neighbour.

4. Select the neighbour with the lowest cost. If no neighbour has a lower cost than the current state, terminate. (reached a local or global optimum).

5. If a neighbour has a lower cost, move to that neighbour and repeat the process.

6. If stuck at a local optimum, optionally restart with a new random configuration (random-restart hill climbing).

7. Stop when the goal (zero conflict) is reached.

Pseudocode:

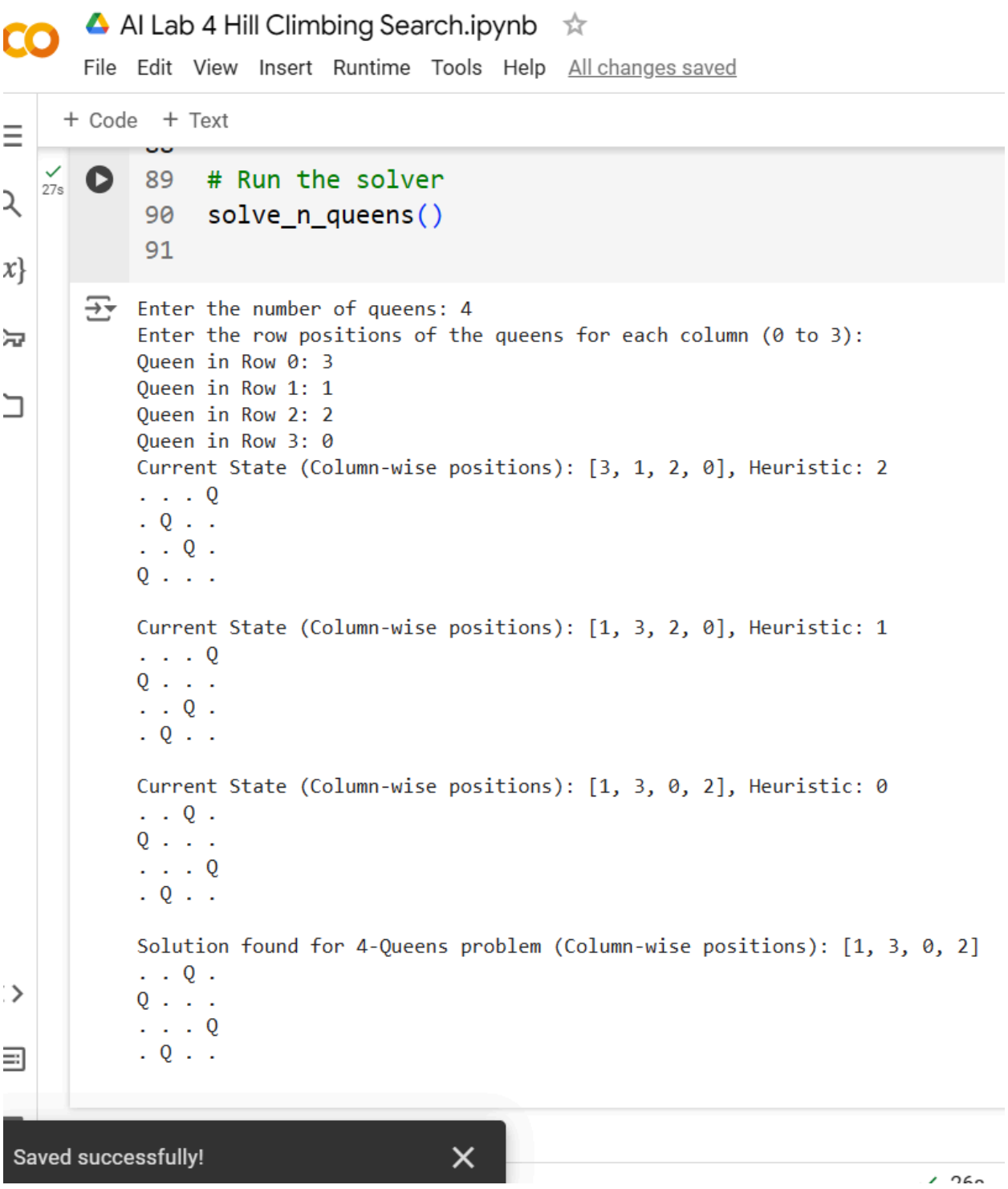
```
# Pseudocode:

function hill_climbing_n_queens( initial state):
    current_state = initial_state

    while true:
        current_heuristic = calculate_heuristic (current)

        if current_heuristic == 0:
            return current_state

        neighbours = generate_neighbours (current_state)
        best_neighbour = None
        best_heuristic = infinity

        for each neighbour in neighbours:
            heuristic = calculate_heuristic (neighbour)
            if heuristic < best_heuristic
                best_heuristic = heuristic
                best_neighbour = neighbour

        if best heuristic >= current_heuristic:
            break // No better neighbour found.

        current_state = best_neighbour

    return None // No solutions found.
```

Code:

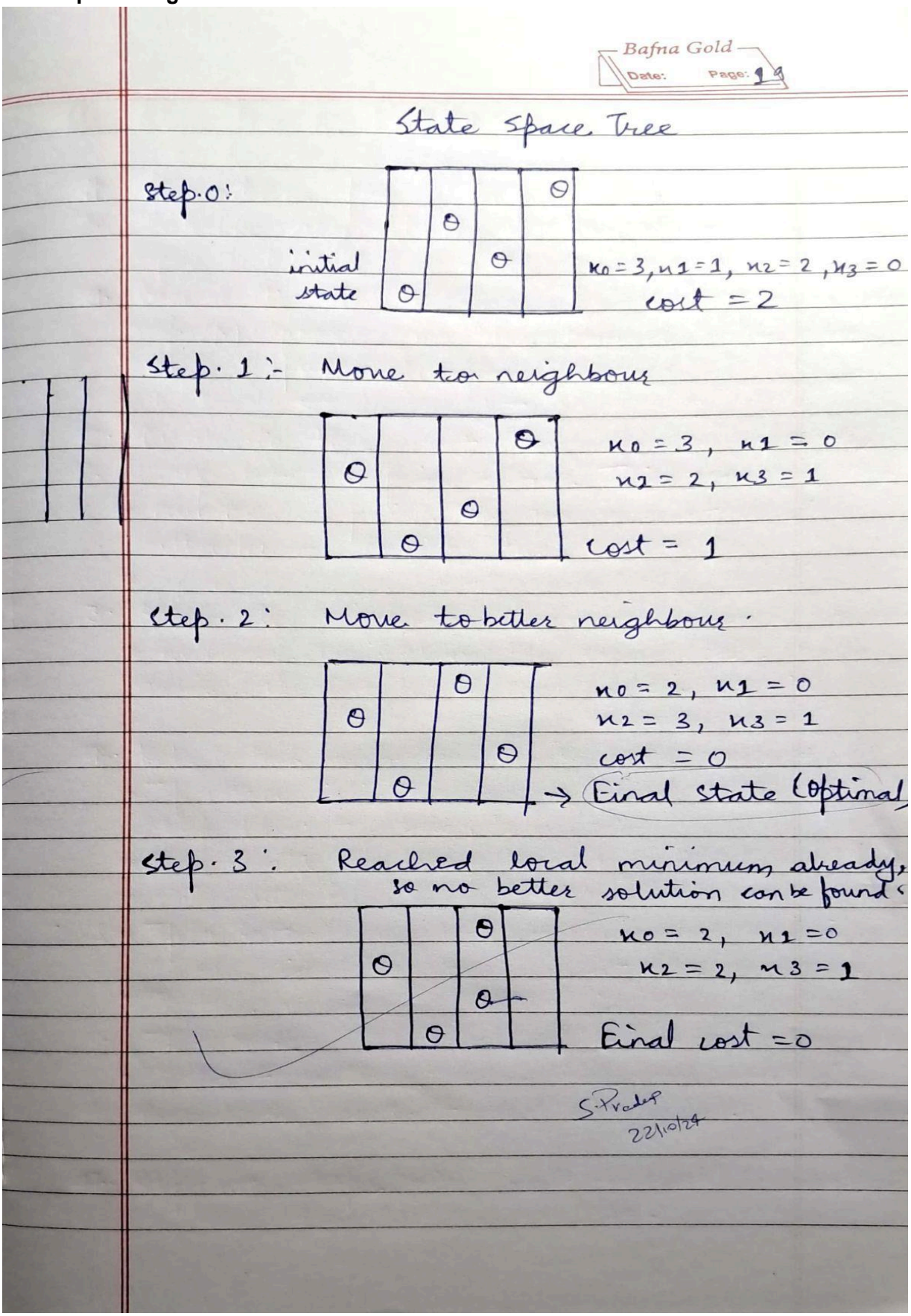
```
import random
def calculate_attacks(board):
    attacks = 0
    n = len(board)
    for i in range(n):
        for j in range(i + 1, n):
            if board[i] == board[j] or abs(board[i] - board[j]) == abs(i - j):
                attacks += 1
    return attacks

def get_neighbors(board):
    neighbors = []
    n = len(board)

    for row in range(n):
        for col in range(n):
            if col != board[row]:
                new_board = board[:]
                new_board[row] = col
                neighbors.append(new_board)

    return neighbors


def hill_climbing(n):

    current_board = [random.randint(0, n-1) for _ in range(n)]


    step = 0
    while True:

        current_attacks = calculate_attacks(current_board)


        print(f"Step {step}:")
        print_board(current_board)
        print(f"Heuristic (attacks): {current_attacks}\n")


        neighbors = get_neighbors(current_board)
```

```
        best_board = current_board
        best_attacks = current_attacks

        for neighbor in neighbors:
            attacks = calculate_attacks(neighbor)
            if attacks < best_attacks:
                best_board = neighbor
                best_attacks = attacks


        if best_attacks == current_attacks:
            break
        else:
            current_board = best_board
            step += 1

    return current_board


def print_board(board):
    n = len(board)
    for i in range(n):
        row = ['Q' if col == board[i] else '.' for col in range(n)]
        print(" ".join(row))


n = 10000000
solution = hill_climbing(n)


print("Final Solution:")
print_board(solution)
print(f"Heuristic (attacks): {calculate_attacks(solution)}")
```

Output:


AI Lab 4 Hill Climbing Search.ipynb
File Edit View Insert Runtime Tools Help All changes saved
+ Code + Text
27s
89 # Run the solver
90 solve_n_queens()
91
Enter the number of queens: 4
Enter the row positions of the queens for each column (0 to 3):
Queen in Row 0: 3
Queen in Row 1: 1
Queen in Row 2: 2
Queen in Row 3: 0
Current State (Column-wise positions): [3, 1, 2, 0], Heuristic: 2
. . . Q
. Q . .
. . Q .
Q . . .
Current State (Column-wise positions): [1, 3, 2, 0], Heuristic: 1
. . . Q
Q . . .
. . Q .
. Q . .
Current State (Column-wise positions): [1, 3, 0, 2], Heuristic: 0
. . Q .
Q . . .
. . . Q
. Q . .
Solution found for 4-Queens problem (Column-wise positions): [1, 3, 0, 2]
. . Q .
Q . . .
. . . Q
. Q . .
Saved successfully!

State Space Diagram:



Lab-5 **29/10/2024**

8. Simulated Annealing to solve N-Queens problem.

Algorithm:

Lab-5 (continued)

Stimulated Annealing for N-Queens.

Algorithm:

1. Initialization
   - Start with a random configuration of queens on board.
   - Calculate initial energy.
   - set the initial temperature To and final temperature Tf.

2. Annealing Loop:
   - Repeat with $T < T_f$:
     - Generate neighbouring configuration.
     - Calculate the energy of new config.
     - If new config has lower energy, accept it.
     - If new config has higher energy, ΔE is change in energy.
     - Reduce temp T using cooling rate 'd' ; $T = T \times d$.

3. Termination :
   - Stop when T reaches T_f, or a solution with zero conflicts is found.

4. Output :
   - If a valid solution (zero conflicts) is found, then return solution.
   - Otherwise, return the best config found.

Code:

```
import random
import math

def create_board(n, initial_config=None):
    """Create a board with the specified initial configuration."""
    if initial_config:
        return initial_config  # Use the provided initial configuration
    return [random.randint(0, n-1) for _ in range(n)]

def count_conflicts(board):
    """Count the number of pairs of queens that are attacking each other."""
    n = len(board)
    conflicts = 0
    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 get_neighbors(board):
    """Generate neighbors by moving one queen at a time to a different row."""
    neighbors = []
    for i in range(len(board)):
        for row in range(len(board)):
            if row != board[i]:
                neighbor = board[:]
                neighbor[i] = row
                neighbors.append(neighbor)
    return neighbors

def print_board(board):
    """Print the board in a human-readable format."""
    n = len(board)
    for row in range(n):
        line = ['Q' if board[col] == row else '.' for col in range(n)]
        print(' '.join(line))
    print()

def simulated_annealing(n, initial_config, max_iterations=1000, initial_temperature=100,
cooling_rate=0.95):
    """Solve the N-Queens problem using simulated annealing."""
    current_board = create_board(n, initial_config)
    current_conflicts = count_conflicts(current_board)
```

```
    temperature = initial_temperature

    best_board = current_board[:]
    best_conflicts = current_conflicts

    # Print the initial configuration
    print("Initial configuration:")
    print_board(current_board)
    print(f"Initial number of conflicts: {current_conflicts}\n")

    for iteration in range(max_iterations):
        if current_conflicts == 0:
            # If the solution is found, print the final configuration and exit
            print(f"Solution found at iteration {iteration + 1}!")
            print("Final configuration (solution):")
            print_board(current_board)
            return current_board

        neighbors = get_neighbors(current_board)
        next_board = random.choice(neighbors)
        next_conflicts = count_conflicts(next_board)

        if next_conflicts < current_conflicts:
            current_board = next_board
            current_conflicts = next_conflicts
        else:
            probability = math.exp((current_conflicts - next_conflicts) / temperature)
            if random.random() < probability:
                current_board = next_board
                current_conflicts = next_conflicts

        temperature *= cooling_rate

    print(f"No solution found within {max_iterations} iterations.")
    print("Best configuration found:")
    print_board(best_board)
    return best_board

# Set the number of queens (4 for this example) and initial configuration
n = 4
initial_config = [2, 1, 3, 0]
solution = simulated_annealing(n, initial_config)
```

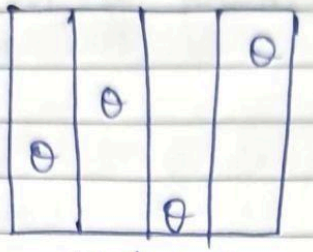
Output:

```
initial_config = [2, 1, 3, 0]
solution = simulated_annealing(n, initial_config)
```

```
Initial configuration:
. . . Q
. Q . .
Q . . .
. . Q .

Initial number of conflicts: 1

Solution found at iteration 114!
Final configuration (solution):
. . Q .
Q . . .
. . . Q
. Q . .
```

State Space Diagram:

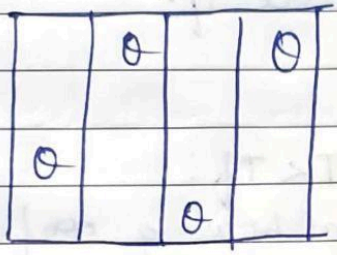
State - Space Tree.



cost = 1

no. of queens = 4
initial set = {2, 1, 3, 0}
T = 100
$\alpha = 0.95$

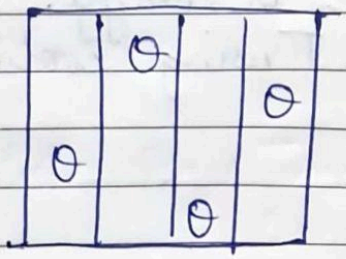
- Generate a neighbour:
  Move queen 2 from row 1 to row 0.



new state = {2, 0, 3, 0}
cost = 1
A cost = 1 - 1 = 0

$n = e - T_{cost} = e - 0 = e$
$T = T \times 0.95 = 95$

- Generate neighbour.



new state = {2, 0, 3, 1}
cost = 0
since cost = 0, stop.

∴ accept solution, thus final state

Lab-6 **12/11/2024**

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

Algorithm:

Lab - 6

Propositional Logic.

1. Initialize Truth Value Combinations.
   - Generate all possible True/False combinations for the propositional variables.

2. Define Operator Priority.
   - Set priorities for '~' highest, 'v' lowest.

3. User Input.
   - Prompt the user for the knowledge Base (KB) and Query (Q) expressions.

4. Convert Expressions to Postfix.
   - Convert both KB and Q from infix to postfix notation.
   - Add operands directly to the postfix expressions.
   - Use a stack to handle parenthesis and operator precedence.

5. Evaluate Postfix Expressions.
   For each truth value combination :
   - Evaluate KB and evaluate Q using a stack - based postfix evaluation.
   - Push operand values from the current combination onto the stack.
   - Apply operators using stack operations (e.g. '~' negates the top, '^' and 'v' operate on the two values.

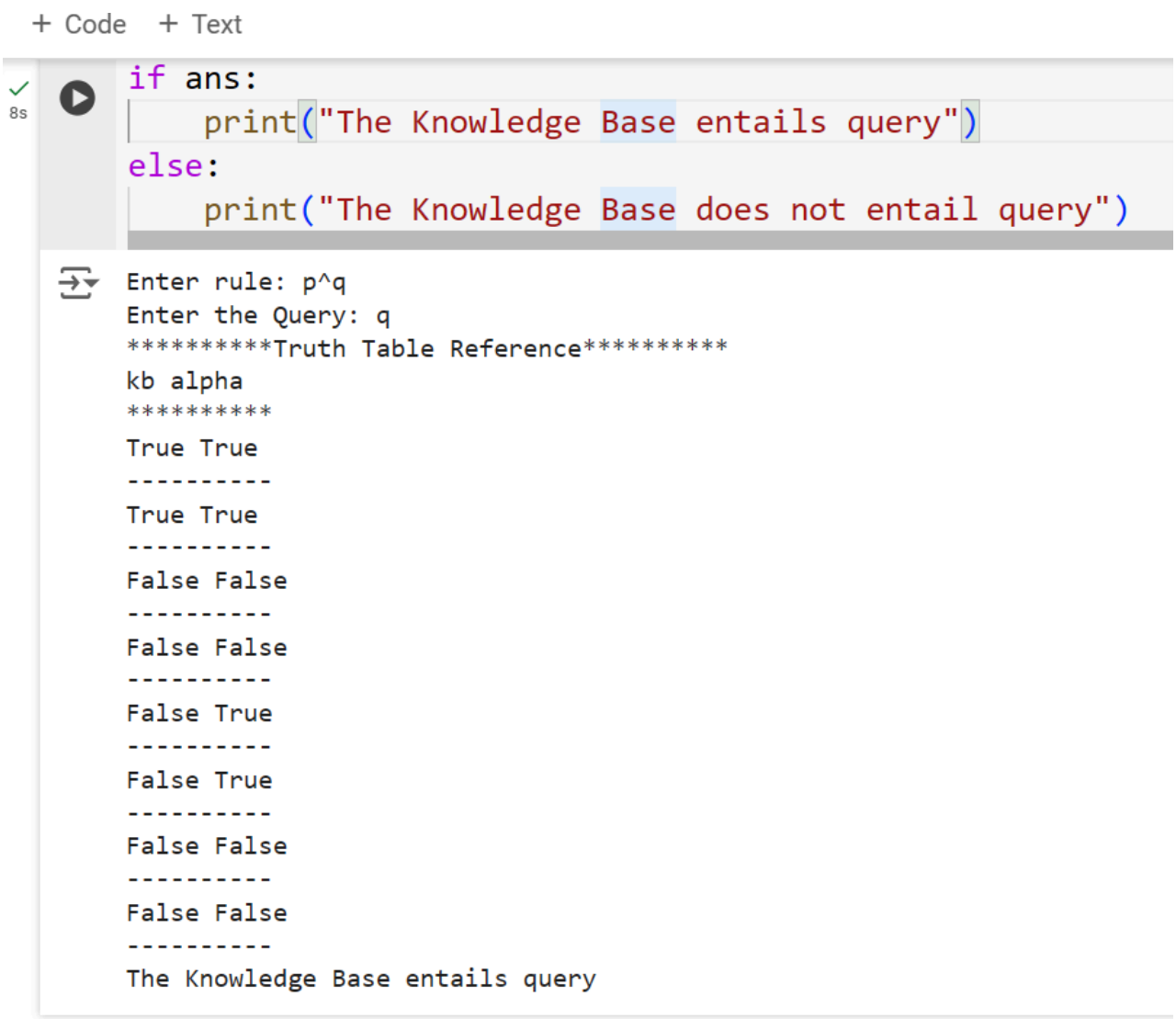
Code:

```
combinations = [(True, True, True), (True, True, False), (True, False, True), (True, False, False),
                (False, True, True), (False, True, False), (False, False, True), (False, False, False)]
variable = {'p': 0, 'q': 1, 'r': 2}
kb = ''
q = ''
priority = {'~': 3, 'v': 1, '^': 2}


def input_rules():
    global kb, q
    kb = input("Enter rule: ")
    q = input("Enter the Query: ")


def entailment():
    global kb, q
    print('*' * 10 + "Truth Table Reference" + '*' * 10)
    print('kb', 'alpha')
    print('*' * 10)
    for comb in combinations:
        s = evaluatePostfix(toPostfix(kb), comb)
        f = evaluatePostfix(toPostfix(q), comb)
        print(s, f)
        print('-' * 10)
        if s and not f:
            return False
    return True


def isOperand(c):
    return c.isalpha() and c != 'v'


def isLeftParanthesis(c):
    return c == '('


def isRightParanthesis(c):
    return c == ')'


def isEmpty(stack):
    return len(stack) == 0
```

```
def peek(stack):
    return stack[-1]


def hasLessOrEqualPriority(c1, c2):
    try:
        return priority[c1] <= priority[c2]
    except KeyError:
        return False


def toPostfix(infix):
    stack = []
    postfix = ''
    for c in infix:
        if isOperand(c):
            postfix += c
        else:
            if isLeftParanthesis(c):
                stack.append(c)
            elif isRightParanthesis(c):
                operator = stack.pop()
                while not isLeftParanthesis(operator):
                    postfix += operator
                    operator = stack.pop()
            else:
                while (not isEmpty(stack)) and hasLessOrEqualPriority(c, peek(stack)):
                    postfix += stack.pop()
                stack.append(c)
    while (not isEmpty(stack)):
        postfix += stack.pop()
    return postfix


def evaluatePostfix(exp, comb):
    stack = []
    for i in exp:
        if isOperand(i):
            stack.append(comb[variable[i]])
        elif i == '~':
            val1 = stack.pop()
            stack.append(not val1)
```

```
        else:
            val1 = stack.pop()
            val2 = stack.pop()
            stack.append(_eval(i, val2, val1))
    return stack.pop()


def _eval(i, val1, val2):
    if i == '^':
        return val2 and val1
    return val2 or val1


input_rules()
ans = entailment()
if ans:
    print("The Knowledge Base entails query")
else:
    print("The Knowledge Base does not entail query")
```

Output:


+ Code
+ Text
8s
if ans:
print("The Knowledge Base entails query")
else:
print("The Knowledge Base does not entail query")
Enter rule: p^q
Enter the Query: q
**********Truth Table Reference**********
kb alpha
****
True True

True True

False False

False False

False True

False True

False False

False False

The Knowledge Base entails query

State Space Diagram:

6. Check Entailment.
- If there's any combination where KB is True and Q is false, return False.
- If no such combination exists, return True.

7. Output Result.
- Print "The Knowledge Base entails the query" if entailment holds; otherwise, print "The knowledge Base does not entail the query".

#

State	p	q	r	KB evaluation	Query	KB → Query
1.	True	True	True	True	True	True
2.	True	True	False	True	False	False
3.	True	False	True	False	True	True
4.	True	False	False	False	False	True
5.	False	~~True~~ False	True	False	True	True
6.	False	True	False	False	False	True
7.	False	False	True	False	True	True
8.	False	False	False	False	False	True

Result: The query doesn't entail the query as there is a True-False relation which entails to False.

S. Pradeep
12/10/24

Lab-7 **19/11/2024**

10. Implement unification in first order logic.

Algorithm:

Lab - 7

Unification in First Order Logic.

Algorithm.

1. If Ψ_1 or Ψ_2 is a variable or constant:
   a) If Ψ_1 or Ψ_2 are identical:
      then return NIL.
   b) Else if Ψ_1 is a variable,
      i) then if Ψ_1 occurs in Ψ_2, then return Failure.
      ii) else return $\{(\Psi_2/\Psi_1)\}$.
   c) Else if Ψ_2 is a variable,
      i) If Ψ_2 occurs in Ψ_1, then return Failure.
      ii) Else return $\{(\Psi_1/\Psi_2)\}$
   d) Else return ~~$\{(\Psi_1/\Psi_2)\}$~~ Failure.

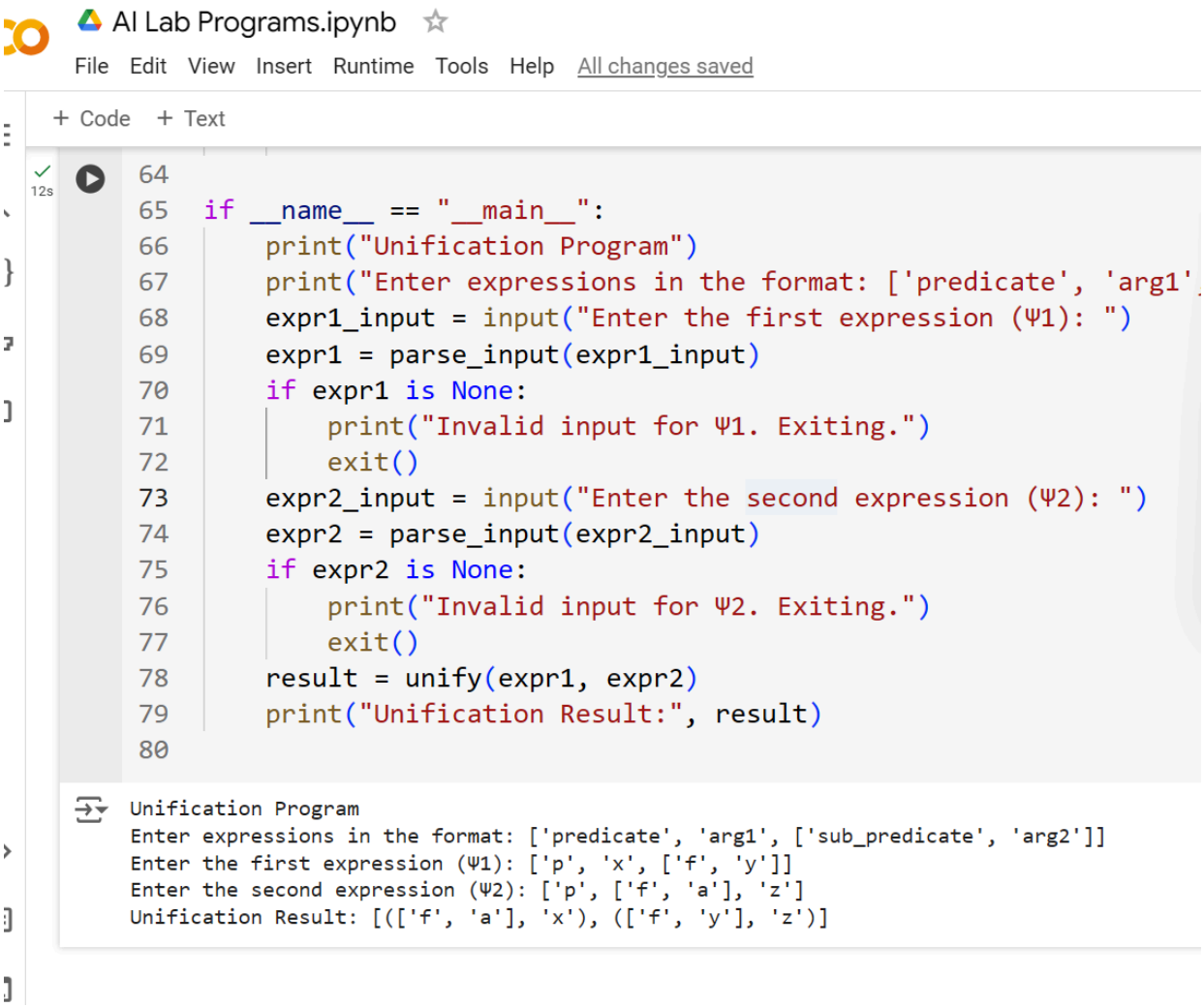
2. If the initial Predicate symbol in Ψ_1 and Ψ_2 are not same, then return Failure.
3. If Ψ_1 and Ψ_2 have a different number of arguements, then return Failure.
4. Set Substitution set(SUBS) to NIL.
5. For i = 1 to the number of elements in Ψ_1:
   a) Call Unify function with the ith element of Ψ_1 and ith element of Ψ_2, and put the result into S.
   b) If S = failure then return Failure.
   c) If S ≠ NIL, then do.
      i) Apply S to the remainder of both L_1 & L_2.
      ii) SUBST = APPEND(S, SUBST).
6. Return SUBST.

Code:

```
def unify(expr1, expr2):
    if is_variable_or_constant(expr1) or is_variable_or_constant(expr2):
        if expr1 == expr2:
            return []
        if is_variable(expr1):
            if occurs_check(expr1, expr2):
                return "FAILURE"
            return [(expr2, expr1)]
        if is_variable(expr2):
            if occurs_check(expr2, expr1):
                return "FAILURE"
            return [(expr1, expr2)]
        return "FAILURE"
    if not same_predicate(expr1, expr2):
        return "FAILURE"
    args1 = get_arguments(expr1)
    args2 = get_arguments(expr2)
    if len(args1) != len(args2):
        return "FAILURE"
    SUBST = []
    for i in range(len(args1)):
        s = unify(args1[i], args2[i])
        if s == "FAILURE":
            return "FAILURE"
        if s:
            args1 = apply_substitution(s, args1)
            args2 = apply_substitution(s, args2)
            SUBST.extend(s)
    return SUBST

def is_variable_or_constant(expr):
    return isinstance(expr, str) and expr.islower()

def is_variable(expr):
    return isinstance(expr, str) and expr.islower()

def occurs_check(var, expr):
    if var == expr:
        return True
    if isinstance(expr, list):
        return any(occurs_check(var, sub) for sub in expr)
    return False
```

```
def same_predicate(expr1, expr2):
    return expr1[0] == expr2[0] if isinstance(expr1, list) and isinstance(expr2, list) else False

def get_arguments(expr):
    return expr[1:] if isinstance(expr, list) else []

def apply_substitution(subst, expr):
    if isinstance(expr, list):
        return [apply_substitution(subst, sub) for sub in expr]
    for (new, old) in subst:
        if expr == old:
            return new
    return expr

def parse_input(expr):
    try:
        return eval(expr)
    except Exception as e:
        print(f"Error in input format: {e}")
        return None

if __name__ == "__main__":
    print("Unification Program")
    print("Enter expressions in the format: ['predicate', 'arg1', ['sub_predicate', 'arg2']]")
    expr1_input = input("Enter the first expression (Ψ1): ")
    expr1 = parse_input(expr1_input)
    if expr1 is None:
        print("Invalid input for Ψ1. Exiting.")
        exit()
    expr2_input = input("Enter the second expression (Ψ2): ")
    expr2 = parse_input(expr2_input)
    if expr2 is None:
        print("Invalid input for Ψ2. Exiting.")
        exit()
    result = unify(expr1, expr2)
    print("Unification Result:", result)
```

Output:


AI Lab Programs.ipynb
File Edit View Insert Runtime Tools Help All changes saved
+ Code + Text
12s
64
65 if __name__ == "__main__":
66 print("Unification Program")
67 print("Enter expressions in the format: ['predicate', 'arg1'
68 expr1_input = input("Enter the first expression (Ψ1): ")
69 expr1 = parse_input(expr1_input)
70 if expr1 is None:
71 print("Invalid input for Ψ1. Exiting.")
72 exit()
73 expr2_input = input("Enter the second expression (Ψ2): ")
74 expr2 = parse_input(expr2_input)
75 if expr2 is None:
76 print("Invalid input for Ψ2. Exiting.")
77 exit()
78 result = unify(expr1, expr2)
79 print("Unification Result:", result)
80
Unification Program
Enter expressions in the format: ['predicate', 'arg1', ['sub_predicate', 'arg2']]
Enter the first expression (Ψ1): ['p', 'x', ['f', 'y']]
Enter the second expression (Ψ2): ['p', ['f', 'a'], 'z']
Unification Result: ([['f', 'a'], 'x'), (['f', 'y'], 'z')]

State Space Diagram:

Output:

Case 1: Basic Unification.
Enter the first expression (Ψ_1): ['p', 'x', ['f', 'y']]
Enter the second expression (Ψ_2): ['p', ['f', 'a'], 'z']
Output:
Unification Result: [[['f', 'a'], 'x'], ['z', 'y']]

Case-2: Failure Due to Mismatched Predicates
Enter the first expression (Ψ_1): ['p', 'x', 'y']
Enter the second expression (Ψ_2): ['q', 'x', 'y']
Output:
Unification Result: FAILURE.

Case-3: Substitution Chains.
Enter the first expression (Ψ_1): ['p', 'x', 'y']
Enter the second expression (Ψ_2): ['p', 'a', 'b']
Output:
Unification Results: [['a', 'x'], ['b', 'y']]

S. Pradeep 19/11/24

Lab-8 **26/11/2024**

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

Algorithm:

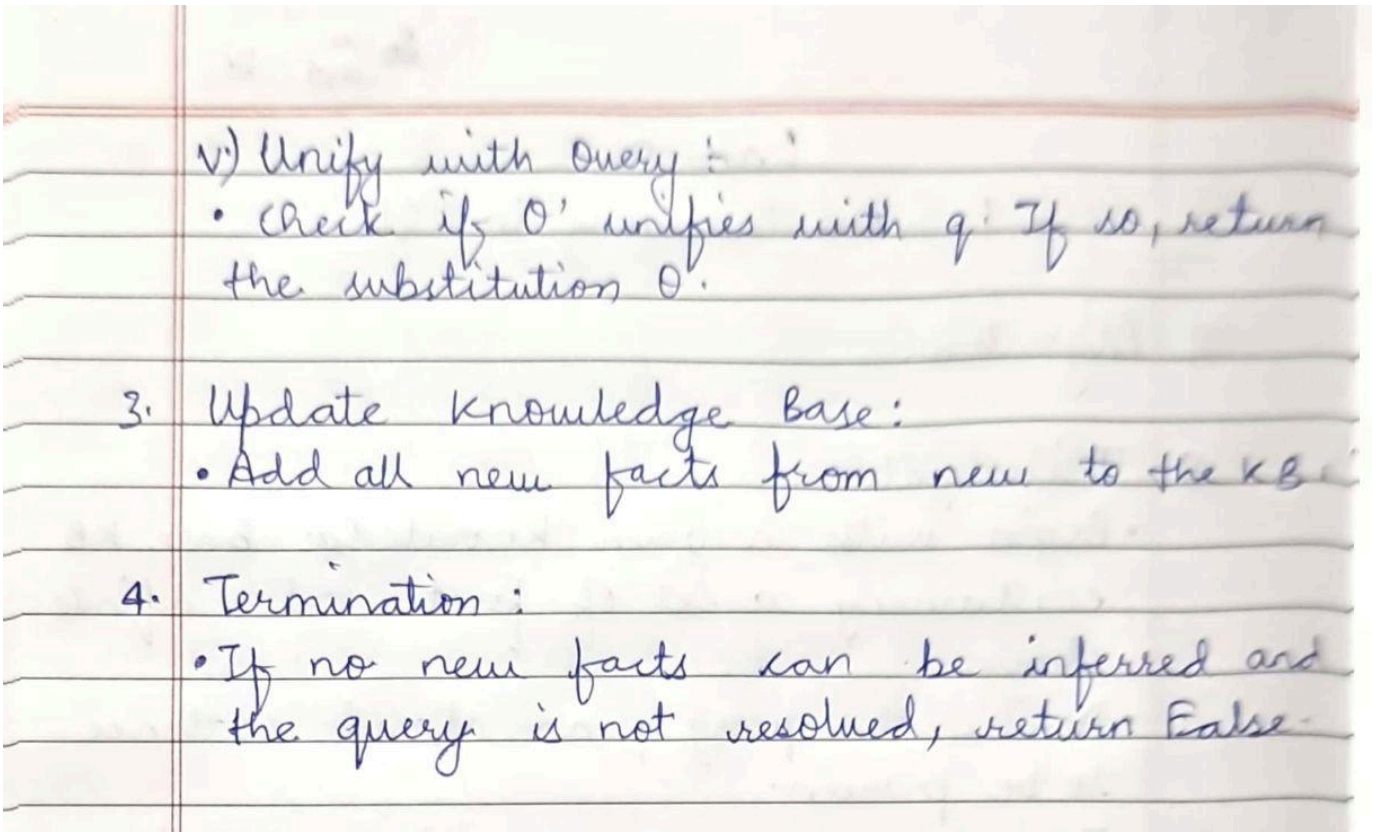
Lab-8

Forward Reasoning in FOL

Algorithm:

1. Initialization:
   - Begin with a given knowledge base KB containing a set of first-order definite clauses.
   - Define the query q, an atomic sentence to be proven.
   - Initialize new as an empty set to track newly derived facts.

2. Iteration:
   - Loop until no new facts can be inferred.
   - For each rule r in the knowledge base:
     i) Standardize Variables:
        - Replace all variables in r with unique variables to avoid clashes with existing variables in the KB.
     ii) Check Premises:
        - Find substitutions θ such that all premises $P_1, P_2, \ldots, P_n$ of the rule hold in the current KB.
     iii) Derive Conclusions:
        - Apply θ to the conclusion Q to infer a new fact 'Q'.
     iv) Avoid Redundancy:
        - Ensure Q' is not already in KB or new before adding it to new.



Code:

```
class ForwardChainingFOL:
    def __init__(self):
        self.facts = set()
        self.rules = []

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

    def add_rule(self, premises, conclusion):
        self.rules.append((premises, conclusion))

    def unify(self, fact1, fact2):

        if fact1 == fact2:
            return {}
        if "(" in fact1 and "(" in fact2:

            pred1, args1 = fact1.split("(", 1)
            pred2, args2 = fact2.split("(", 1)
            args1 = args1[:-1].split(",")
            args2 = args2[:-1].split(",")
            if pred1 != pred2 or len(args1) != len(args2):
                return None
```

```
        substitution = {}
        for a1, a2 in zip(args1, args2):
            if a1 != a2:
                if a1.islower():
                    substitution[a1] = a2
                elif a2.islower():
                    substitution[a2] = a1
                else:
                    return None
        return substitution
    return None

def apply_substitution(self, fact, substitution):

    if "(" in fact:
        pred, args = fact.split("(", 1)
        args = args[:-1].split(",")
        substituted_args = [substitution.get(arg, arg) for arg in args]
        return f"{pred}({','.join(substituted_args)})"
    return fact

def forward_chain(self, goal):
    iteration = 1
    while True:
        new_facts = set()
        print(f"\n=== Iteration {iteration} ===")
        print("Known Facts:")
        for fact in self.facts:
            print(f"  - {fact}")

        print("\nApplying rules...")
        rule_triggered = False

        for premises, conclusion in self.rules:
            substitutions = [{}]
            for premise in premises:
                new_substitutions = []
                for fact in self.facts:
                    for sub in substitutions:
                        unified = self.unify(self.apply_substitution(premise, sub), fact)
                        if unified is not None:
                            new_substitutions.append({**sub, **unified})
                substitutions = new_substitutions
```

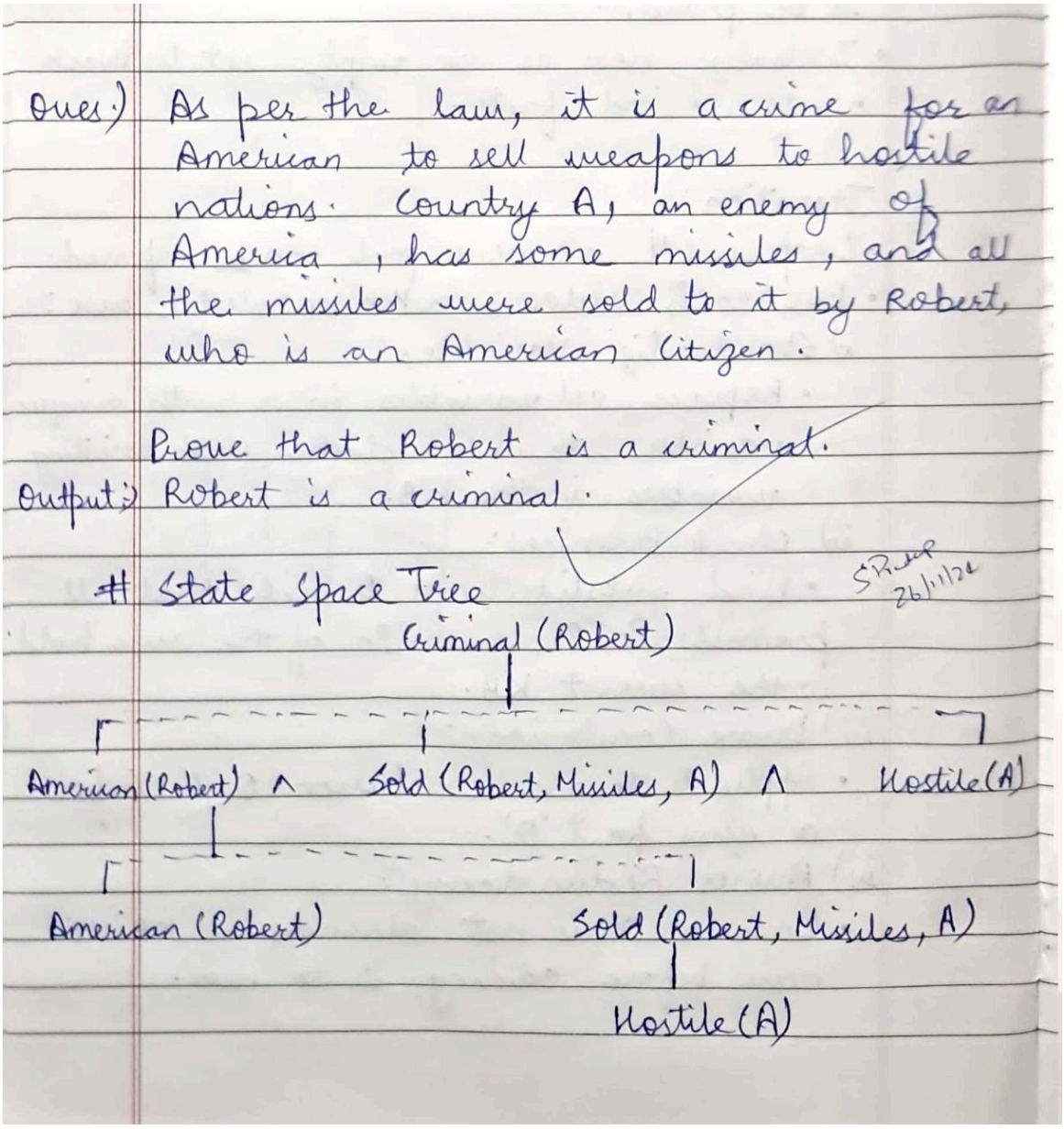
```
            for sub in substitutions:
                inferred_fact = self.apply_substitution(conclusion, sub)
                if inferred_fact not in self.facts:
                    rule_triggered = True
                    print(f"Rule triggered: {premises} → {conclusion}")
                    print(f"  New fact inferred: {inferred_fact}")
                    new_facts.add(inferred_fact)

        if not new_facts:
            if not rule_triggered:
                print("No rules triggered in this iteration.")
            print("No new facts inferred in this iteration.")
            break

        self.facts.update(new_facts)
        if goal in self.facts:
            print(f"\nGoal {goal} reached!")
            return True
        iteration += 1

    print("\nGoal not reached.")
    return False


fc = ForwardChainingFOL()

fc.add_fact("American(Robert)")
fc.add_fact("Enemy(A,America)")
fc.add_fact("Owns(A,T1)")
fc.add_fact("Missile(T1)")

fc.add_rule(["Missile(T1)"], "Weapon(T1)")
fc.add_rule(["Enemy(A,America)"], "Hostile(A)")
fc.add_rule(["Missile(p)", "Owns(A,p)"], "Sells(Robert,p,A)")
fc.add_rule(["American(p)", "Weapon(q)", "Sells(p,q,r)", "Hostile(r)"], "Criminal(p)")

goal = "Criminal(Robert)"

if fc.forward_chain(goal):
    print(f"\nFinal result: Goal achieved: {goal}")
else:
    print("\nFinal result: Goal not achieved.")
```

Output:

```
else:
    print("\nFinal result: Goal not achieved.")
```

```
=== Iteration 1 ===
Known Facts:
  - Owns(A,T1)
  - Missile(T1)
  - American(Robert)
  - Enemy(A,America)

Applying rules...
Rule triggered: ['Missile(T1)'] → Weapon(T1)
  New fact inferred: Weapon(T1)
Rule triggered: ['Enemy(A,America)'] → Hostile(A)
  New fact inferred: Hostile(A)
Rule triggered: ['Missile(p)', 'Owns(A,p)'] → Sells(Robert,p,A)
  New fact inferred: Sells(Robert,T1,A)

=== Iteration 2 ===
Known Facts:
  - Owns(A,T1)
  - Weapon(T1)
  - American(Robert)
  - Sells(Robert,T1,A)
  - Missile(T1)
  - Hostile(A)
  - Enemy(A,America)

Applying rules...
Rule triggered: ['American(p)', 'Weapon(q)', 'Sells(p,q,r)', 'Hostile(r)'] → Criminal(p)
  New fact inferred: Criminal(Robert)

Goal Criminal(Robert) reached!

Final result: Goal achieved: Criminal(Robert)
```

State Space Diagram:



Lab-9 **03/12/2024**

12. For a given case study write a Proof Tree generated using Resolution.

Algorithm:

Lab-9
FOL Resolution.

Algorithm:

1. Convert all sentences to CNF.

a) Eliminate biconditionals & implications:
- Eliminate $\Leftrightarrow$, replacing $\alpha \Leftrightarrow \beta$ with $(\alpha \Rightarrow \beta) \wedge (\beta \Rightarrow \alpha)$.
- Eliminate $\Rightarrow$, replacing $\alpha \Rightarrow \beta$ with $\neg\alpha \Rightarrow \beta$.

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

c) Standardize variables apart by renaming them: each quantifier should use a different variable.

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

e) Drop universal quantifiers
- For instance, $\forall x$ Person(x) becomes Person(x).

f) Distribute $\wedge$ over $\vee$:
- $(\alpha \wedge \beta) \vee \gamma \equiv (\alpha \vee \gamma) \wedge (\beta \vee \gamma)$.

2. Negate conclusion S and convert the result to CNF

3. Add the negated query to the set of premises (negated conclusion S).

4. Repeat until contradiction or no progress is made:
a) Select 2 parent clauses (any 2 clauses).
b) Resolve them together, performing all required unifications.
c) If resolvent is empty clause, a contradiction has been found.
(i.e., S followed the premise).
d) If not, add resolvent to premises.

When succeeded in step 4, we have proved the conclusion.

Proof by Resolution :

Given KB or premise :

a) John likes all kinds of food.
b) Apple and vegetables are food.
c) Anything anyone eats and not killed is food.
d) Anil eats peanuts and still alive.
e) Harry eats everything that Anil eats.
f) Anyone who is alive implies not killed.
g) Anyone who is not killed implies alive.

To prove :

h) John likes peanuts.

I) First order logic :

a) $\forall x : food(x) \rightarrow likes(John, x)$
b) $food(Apple) \wedge food(vegetables)$
c) $\forall x \forall y : eats(x, y) \wedge \neg killed(x) \rightarrow food(y)$
d) $eats(Anil, peanuts) \wedge alive(Anil)$.
e) $\forall x : eats(Anil, x) \rightarrow eats(Harry, x)$
f) $\forall x : \neg killed(x) \rightarrow alive(x)$
g) $\forall x : alive(x) \rightarrow \neg killed(x)$
h) $likes(John, Peanuts)$.

II) Eliminate implications :

a) $\forall x : \neg food(x) \vee likes(John, x)$
b) $food(Apple) \wedge food(vegetables)$
c) $\forall x \forall y \rightarrow [eats(x, y) \wedge \neg killed(x)] \vee food(y)$.

d.) eats (Anil, Peanuts) ∧ alive (Anil)
e.) ∀x ¬ eats (Anil, x) ∨ eats (Harry, x)
f.) ∀x ¬ [¬ killed (x)] ∨ alive (x)
g.) ∀x ¬ alive (x) ∨ ¬ killed (x)
h.) likes (John, peanuts).

III.) Standardize variables:
c.) ∀y ∀z ¬ eats (y, z) ∨ killed (y) ∨ food (z)
e.) ∀w ¬ eats (Anil, w) ∨ eats (Harry, w)
f.) ∀g killed (g) ∨ alive (g)
g.) ∀k ¬ alive (k) ∨ ¬ killed (k)

IV.) Drop universal quantifiers:
a.) ¬ food (x) ∨ likes (John, x)
b.) food (Apple)
c.) food (Vegetables)
d.) ¬ eats (y, z) ∨ killed (y)
e.) eats (Anil, Peanuts)
f.) Alive (Anil)
g.) ¬ eats (Anil, w) ∨ eats (Harry, w)
h.) killed (g) ∨ alive (g)
i.) ¬ alive (k) ∨ ¬ killed (k)
j.) likes (John, Peanuts).

Output:> John likes Peanuts.

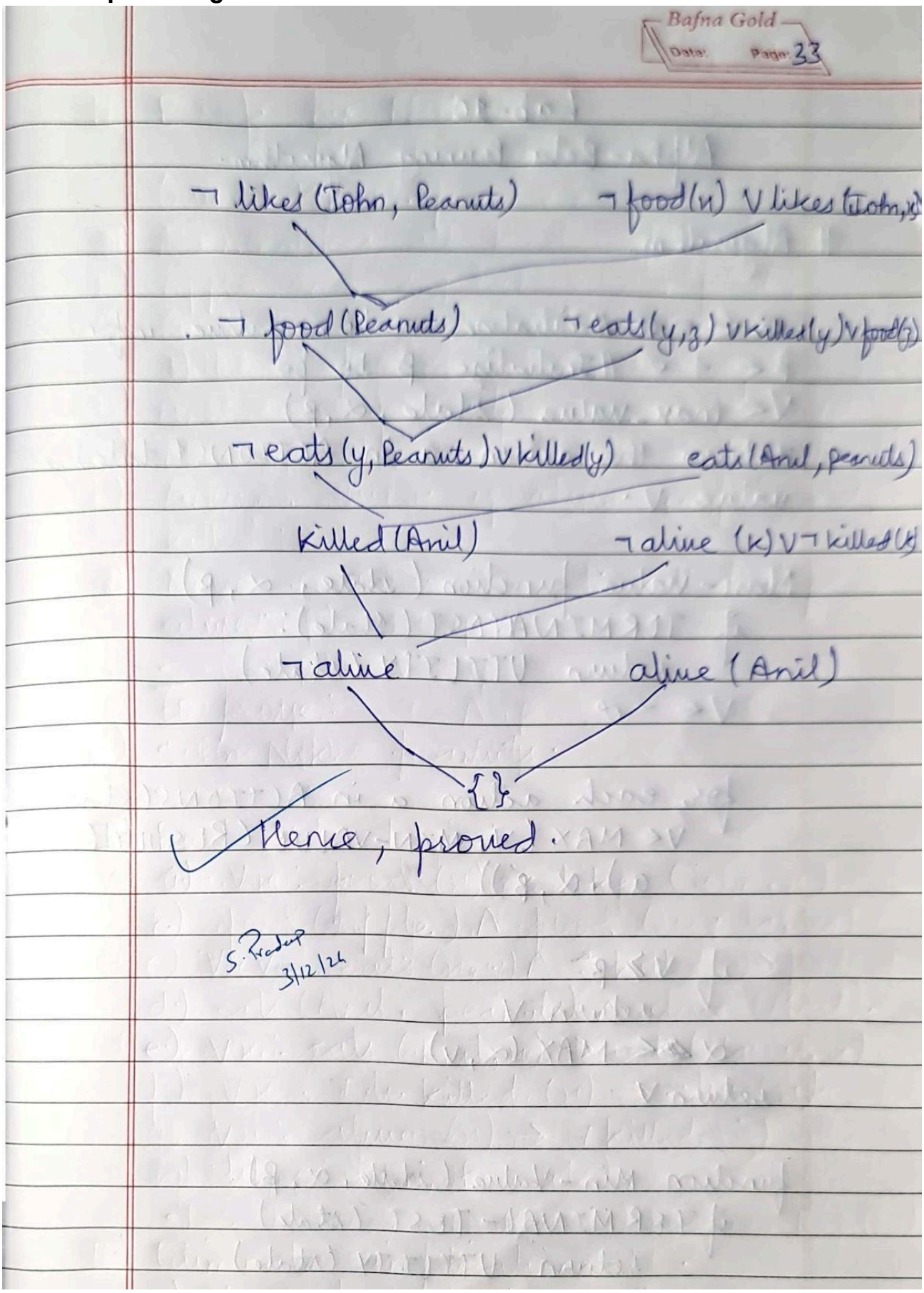
Code:

```
class Clause:
    def __init__(self, literals):
        self.literals = set(literals)

    def __repr__(self):
        return " ∨ ".join(sorted(self.literals))

    def resolve(self, other):
        resolvents = []
        for literal in self.literals:
            negated_literal = f"¬{literal}" if not literal.startswith('¬') else literal[1:]
            if negated_literal in other.literals:
                new_literals = (self.literals - {literal}) | (other.literals - {negated_literal})
                resolvents.append(Clause(new_literals))
        return resolvents


def resolution(clauses, query):
    negated_query = Clause([f"¬{query}"])
    clauses.append(negated_query)

    new = set()
    seen_pairs = set()

    while True:
        pairs = [(clauses[i], clauses[j]) for i in range(len(clauses)) for j in range(i + 1, len(clauses))]
        for ci, cj in pairs:
            if (ci, cj) in seen_pairs or (cj, ci) in seen_pairs:
                continue
            seen_pairs.add((ci, cj))

            resolvents = ci.resolve(cj)
            for resolvent in resolvents:
                if not resolvent.literals:
                    return True
                new.add(frozenset(resolvent.literals))

        if new.issubset(set(map(frozenset, (c.literals for c in clauses)))):
            return False
        clauses.extend(Clause(list(literals)) for literals in new - set(map(frozenset, (c.literals for c in
clauses))))
        new.clear()
```

```
KB = [
    Clause(["¬Food(Peanuts)", "Likes(John, Peanuts)"]),  # John likes all kinds of food
    Clause(["Food(Apple)"]),
    Clause(["Food(Vegetables)"]),
    Clause(["Food(Peanuts)"]),  # Explicit Peanuts as Food
    Clause(["¬Eats(Anil, Peanuts)", "Food(Peanuts)"]),
    Clause(["Eats(Anil, Peanuts)"]),  # Anil eats peanuts
    Clause(["Alive(Anil)"]),          # Anil is alive
    Clause(["¬Alive(Anil)", "¬Killed(Anil)"]),  # Alive -> Not Killed
    Clause(["Killed(Anil)", "Alive(Anil)"])     # Not Killed -> Alive
]

# Query: John likes peanuts
query = "Likes(John, Peanuts)"

# Run the resolution algorithm
if resolution(KB, query):
    print(f"The conclusion '{query}' is proven by resolution.")
else:
    print(f"The conclusion '{query}' cannot be proven.")
```

Output:

State Space Diagram:



Lab-10 **17/12/2024**

12. Write the Alpha-Beta values generated to identify the final value of MAX node and subtrees pruned for a given Game tree using the Alpha-Beta pruning algorithm.

Algorithm:

Lab-10

Alpha-Beta Pruning Algorithm.

Algorithm:

$\alpha \leftarrow -\infty$ // Initialize α to negative infinity.
$\beta \leftarrow +\infty$ // Initialize β to ∞
$V \leftarrow$ max_value (state, α, β)
return the action a in ACTIONS (state) with value V.

Max-value function (state, α, β)
if TERMINAL TEST (state):
return UTILITY (state)
$V \leftarrow -\infty$

for each action a in ACTIONS (state):
$V \leftarrow$ MAX (V, MIN-VALUE (RESULT (state, a), α, β))

if $V \geq \beta$
return V
$\alpha \leftarrow$ MAX (α, V)
return V

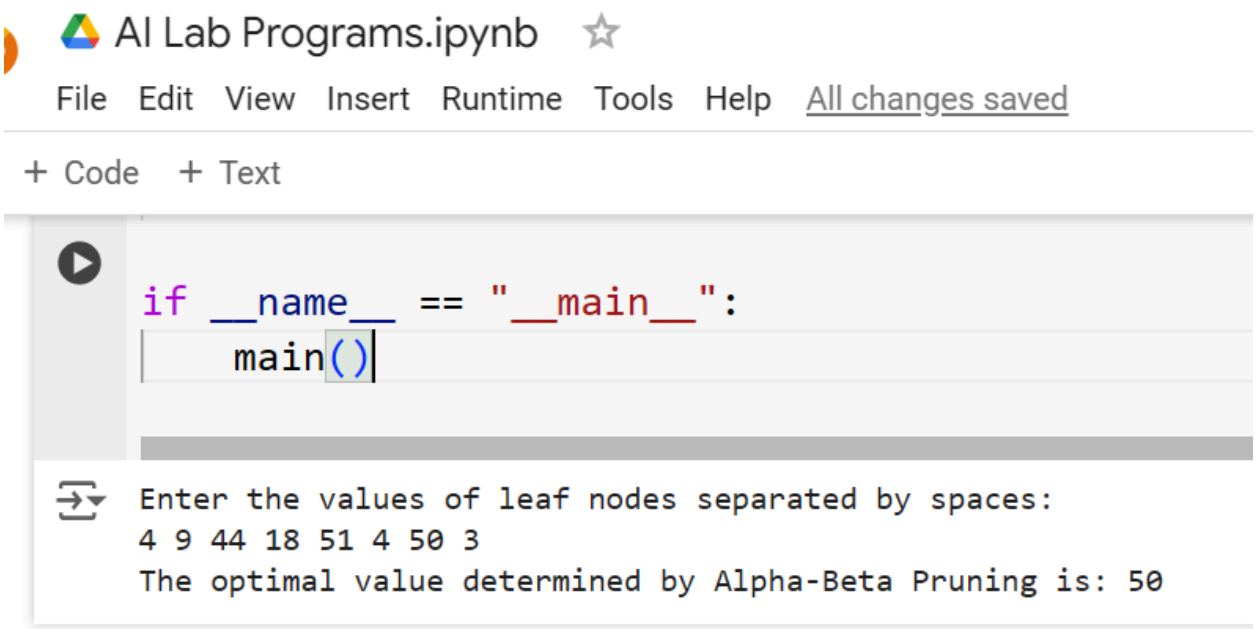
function Min-Value (state, α, β).
if TERMINAL-TEST (state)
return UTILITY (state)
$V \leftarrow +\infty$
for each action a in ACTIONS (state):
$V \leftarrow$ MIN (V, MAX-Value (RESULT (state, a), α, β))
If $V \leq \alpha$ then return V
$\beta \leftarrow$ Min (β, V)
return V

Code:

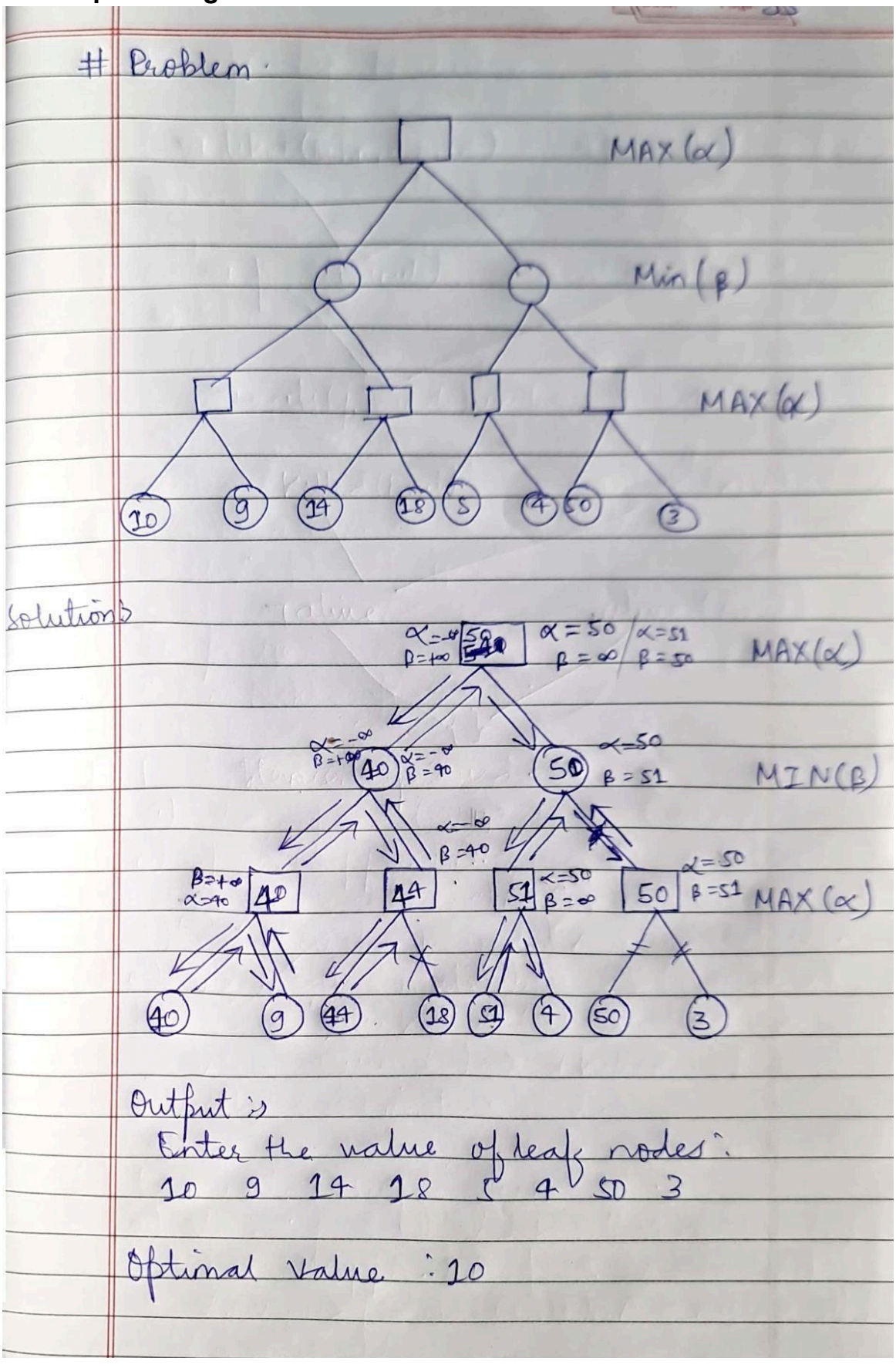
```
import math

# Alpha-Beta Pruning Algorithm
def alpha_beta_search(depth, index, is_max, values, alpha, beta, target_depth):
    """Recursive function for Alpha-Beta Pruning."""
    # Base case: If the target depth is reached, return the leaf node value
    if depth == target_depth:
        return values[index]

    if is_max:
        # Maximizer's turn
        best = -math.inf
        for i in range(2):
            val = alpha_beta_search(depth + 1, index * 2 + i, False, values, alpha, beta,
target_depth)
            best = max(best, val)
            alpha = max(alpha, best)
            if beta <= alpha:
                break  # Prune remaining branches
        return best
    else:
        # Minimizer's turn
        best = math.inf
        for i in range(2):
            val = alpha_beta_search(depth + 1, index * 2 + i, True, values, alpha, beta,
target_depth)
            best = min(best, val)
            beta = min(beta, best)
            if beta <= alpha:
                break  # Prune remaining branches
        return best

def main():
    # User Input: Values of leaf nodes
    print("Enter the values of leaf nodes separated by spaces:")
    values = list(map(int, input().split()))

    # Calculate depth of the game tree
    target_depth = math.log2(len(values))
    if target_depth != int(target_depth):
        print("Error: The number of leaf nodes must be a power of 2.")
        return
    target_depth = int(target_depth)
```

```
    # Run Alpha-Beta Pruning
    result = alpha_beta_search(0, 0, True, values, -math.inf, math.inf, target_depth)

    # Display the result
    print(f"The optimal value determined by Alpha-Beta Pruning is: {result}")

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

Output:



State Space Diagram:



Output is

Enter the value of leaf nodes:

10 9 14 18 5 4 50 3

Optimal value : 10