VISVESVARAYA TECHNOLOGICAL UNIVERSITY

“JnanaSangama”, Belgaum -590014, Karnataka.

LAB REPORT
on

Artificial Intelligence (23CS5PCAIN)

Submitted by

Sakshi B R (1BM22CS233)

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 **Sakshi B R (1BM22CS233),** 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.

Dr. Seema patil Assistant Professor Department of CSE, BMSCE	Dr. Joythi S Nayak Professor & HOD Department of CSE, BMSCE

Index

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

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

Algorithm:

Date 24/3/24

Lab - 1

Tic Tac Toe

→ Initialize the 2D array of 3 row and 3 column with empty space

→ Create the function "board" for display

→ Take the input from the user as 'X' or 'O'

→ Handle the I/P of the user by checking the row, column, diagonal wise by the condition.

→ Create the function victory to check
Iterate and check if all the row/col as occupied by same icon
if occupies → True

If not → False

→ Create a function for draw
Iterate over the board and check whether all the cells are filled or not
if filled → break

→ Create a function for draw
Iterate over the board and check whether all the cells are

→ Create the main game function draw Board:
loop the game

- Ask current player to make move
- Update the move
- Check for the victory and print message
- Check for draw & print draw message
- Check for draw & print draw message
- Ask for draw & print draw message
- Ask for next player move
- Update the move and it goes on.

11.1.23

GoodLuck Page No.
Date 1 | 10 | 24

Lab - 1

Implement vacuum world cleaner.

function Reflex - vacuum - Agent ([Location, Status])
return ~~or~~ an action

if Status = Dirty then
  return suck
else if location = A then
  return Right
else if location = B then
  return left

Input :- Location A
  Status 0
  Dust in another = 1.

Algorithm :-

- Take input:
  Take input from user for the initial location of the vacuum
  Take input from user for the status of the room (0 for clean, 1 for dirty).

- Check if the current room is dirty
  If dirty:
    clean the room
    Increase the total cost by 1
  If already clean, print a message that the room is already clean

- Move to the next room

- Clean the room
- Check goal state
- Print total cleaning cost.

Output

Current state: {'A':0, 'B':0}
Vacuum is in Room A
Total cost so far: 0
Enter initial location of the vacuum (A or B): A
Enter the status for Room A : 1
Enter the status for Room B : 0
Suck at A
Move Right to B
Move Left to A
Total cost : 1
Goal Reached.

1/10/24

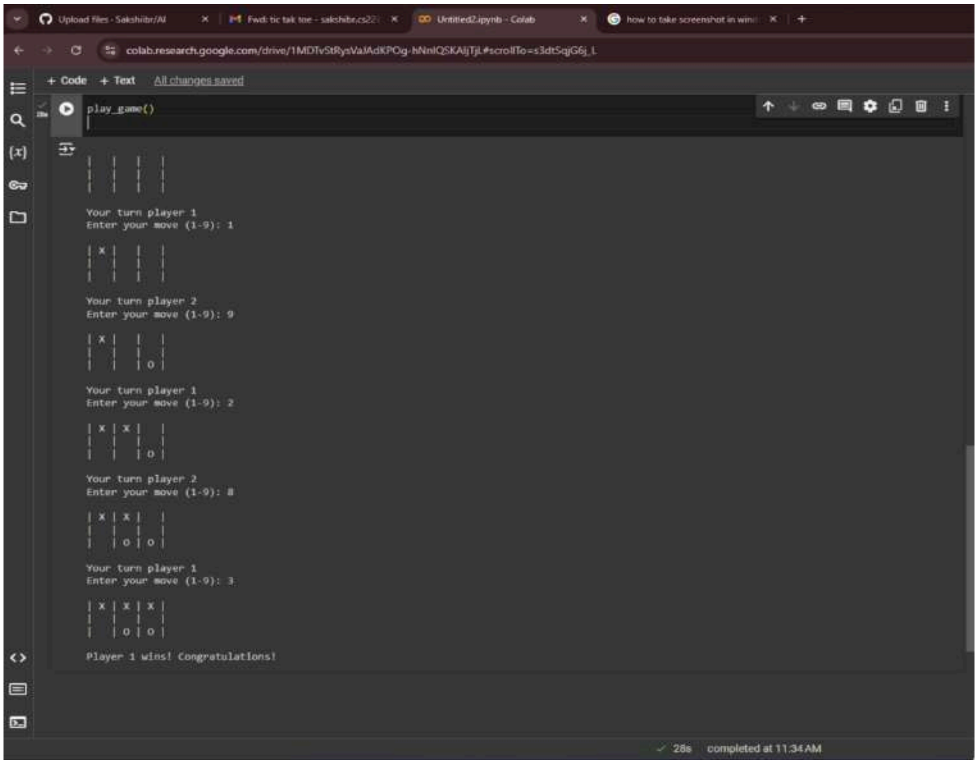
Code:

1.1 Implement Tic –Tac –Toe Game

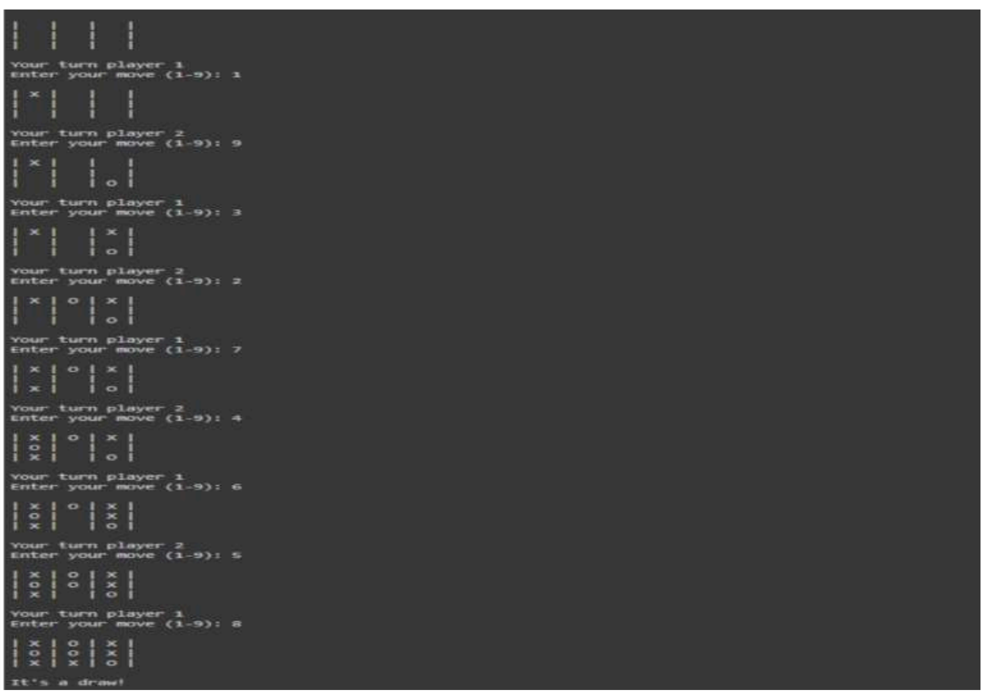
```
# Initialize the board
board = [' ' for _ in range(9)]

# Function to draw the board
def draw_board():
    row1 = '| {} | {} | {} |'.format(board[0], board[1], board[2])
    row2 = '| {} | {} | {} |'.format(board[3], board[4], board[5])
    row3 = '| {} | {} | {} |'.format(board[6], board[7], board[8])
    print()
    print(row1)
    print(row2)
    print(row3)
    print()

# Function for player's move
def player_move(icon):
    if icon == 'X':
        number = 1
    elif icon == 'O':
        number = 2
    print("Your turn, player {}".format(number))
    choice = int(input("Enter your move (1-9): ").strip())
    if board[choice - 1] == ' ':
        board[choice - 1] = icon
    else:
        print()
        print("That space is taken!")

# Function to check for victory
def is_victory(icon):
    if (board[0] == icon and board[1] == icon and board[2] == icon)
or \
       (board[3] == icon and board[4] == icon and board[5] == icon)
or \
       (board[6] == icon and board[7] == icon and board[8] == icon)
or \
       (board[0] == icon and board[3] == icon and board[6] == icon)
or \
       (board[1] == icon and board[4] == icon and board[7] == icon)
or \
       (board[2] == icon and board[5] == icon and board[8] == icon)
or \
       (board[0] == icon and board[4] == icon and board[8] == icon)
or \
```

```
        (board[2] == icon and board[4] == icon and board[6] ==
icon):
        return True
    else:
        return False

# Function to check for a draw
def is_draw():
    if ' ' not in board:
        return True
    else:
        return False

# Function to play the game
def play_game():
    draw_board()
    while True:
        player_move('X')
        draw_board()
        if is_victory('X'):
            print("Player 1 wins! Congratulations!")
            break
        elif is_draw():
            print("It's a draw!")
            break

        player_move('O')
        draw_board()
        if is_victory('O'):
            print("Player 2 wins! Congratulations!")
            break
        elif is_draw():
            print("It's a draw!")
            break

# Start the game
play_game()
```

Output:



+ Code
+ Text
All changes saved
play_game()
Your turn player 1
Enter your move (1-9): 1
Your turn player 2
Enter your move (1-9): 9
Your turn player 1
Enter your move (1-9): 2
Your turn player 2
Enter your move (1-9): 8
Your turn player 1
Enter your move (1-9): 3
Player 1 wins! Congratulations!
28s
completed at 11:34 AM



Your turn player 1
Enter your move (1-9): 1
Your turn player 2
Enter your move (1-9): 9
Your turn player 1
Enter your move (1-9): 3
Your turn player 2
Enter your move (1-9): 2
Your turn player 1
Enter your move (1-9): 7
Your turn player 2
Enter your move (1-9): 4
Your turn player 1
Enter your move (1-9): 6
Your turn player 2
Enter your move (1-9): 5
Your turn player 1
Enter your move (1-9): 8
It's a draw!

1.2 Implement vacuum cleaner agent

```
# Reflex function for vacuum cleaner
def reflex(loc, status, cost):
    s = status  # Track the current status of the location
    if status == 1:  # If the location is dirty
        cost += 1
        print(f"SUCK at {loc}")
        s = 0  # The location is now clean
    if loc == "A":
        print("Move RIGHT to B")
        loc = "B"  # Move to B
    elif loc == "B":
        print("Move LEFT to A")
        loc = "A"  # Move to A
    return cost, loc, s  # Return updated cost, location, and status

# Function to check goal state
def goal(a_status, b_status):
    if a_status == 0 and b_status == 0:
        print("Goal reached")
    else:
        print("Goal not reached")

# Input for the starting location and statuses
loc = input("Enter the starting location of the vacuum (A or B): ").strip()
cost = 0
a_status = int(input("Enter the status of location A (0 for clean, 1 for dirty): "))
b_status = int(input("Enter the status of location B (0 for clean, 1 for dirty): "))

# Simulate cleaning process
if loc == "A":
    cost, loc, a_status = reflex("A", a_status, cost)
    cost, loc, b_status = reflex("B", b_status, cost)
elif loc == "B":
    cost, loc, b_status = reflex("B", b_status, cost)
    cost, loc, a_status = reflex("A", a_status, cost)

# Output the total cost and goal status
print(f"Total cost: {cost}")
goal(a_status, b_status)
```

Output:

```
Enter the starting location of the vacuum (A or B): A
Enter the status of location A (0 for clean, 1 for dirty): 0
Enter the status of location B (0 for clean, 1 for dirty): 0
Move RIGHT to B
Move LEFT to A
Total cost: 0
Goal reached
```

```
Enter the starting location of the vacuum (A or B): B
Enter the status of location A (0 for clean, 1 for dirty): 1
Enter the status of location B (0 for clean, 1 for dirty): 1
SUCK at A
Move RIGHT to B
SUCK at B
Move LEFT to A
Total cost: 2
Goal reached
```

```
Enter the starting location of the vacuum (A or B): A
Enter the status of location A (0 for clean, 1 for dirty): 1
Enter the status of location B (0 for clean, 1 for dirty): 0
SUCK at A
Move RIGHT to B
Move LEFT to A
Total cost: 1
Goal reached
```

```
Enter the starting location of the vacuum (A or B): A
Enter the status of location A (0 for clean, 1 for dirty): 0
Enter the status of location B (0 for clean, 1 for dirty): 1
Move RIGHT to B
SUCK at B
Move LEFT to A
Total cost: 1
Goal reached
```

Program 2
Implement 8 puzzle problems using Depth First Search (DFS)
Implement Iterative deepening search algorithm

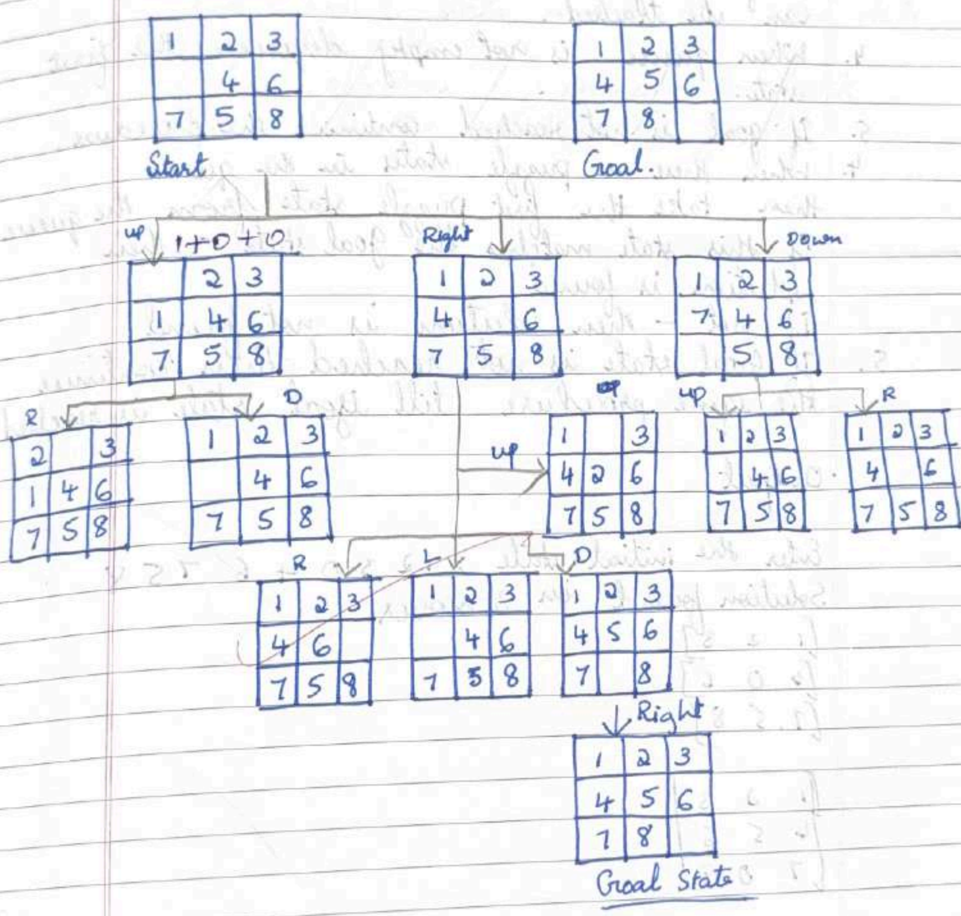
Algorithm:

GoodLuck Page No
Date 8 | 10 | 24

Lab - 2

Solve 8 puzzle using DFS and BFS.

State Space Diagram.



BFS

1. Defining of start state and goal state.
2. Use queue to track the states to reach goal state.
3. By using set visited and non visited states can be tracked.
4. When queue is not empty dequeue the first state.
5. If goal is not reached continue the procedure.
4. When there are puzzle states in the queue then take the first puzzle state from the queue
If this state matches the goal state - then solution is found
If not - then solution is not found
5. If Goal state is not reached then continue the same procedure till goal state is reached.

Output

Enter the initial state : 1 2 3 0 4 6 7 5 8
Solution found in 3 moves

$$\begin{bmatrix}1 & 2 & 3\end{bmatrix}$$
$$\begin{bmatrix}4 & 0 & 6\end{bmatrix}$$
$$\begin{bmatrix}7 & 5 & 8\end{bmatrix}$$

$$\begin{bmatrix}1 & 0 & 3\\4 & 5 & 6\\7 & 0 & 8\end{bmatrix}$$

$$\begin{bmatrix}1 & 2 & 3\\4 & 5 & 6\\7 & 8 & 0\end{bmatrix}$$

DFS Algorithm.

1. Set the goal configuration of the puzzle
2. Locate the empty space
3. Generate valid state by moving adjacent tiles into empty pace
4. If the current state matches the goal state return the path
5 If no solution is found - backtrack.
Continue till the solution is found.

8/10/24

GoodLuck Page No.
Date 22 | 10 | 24

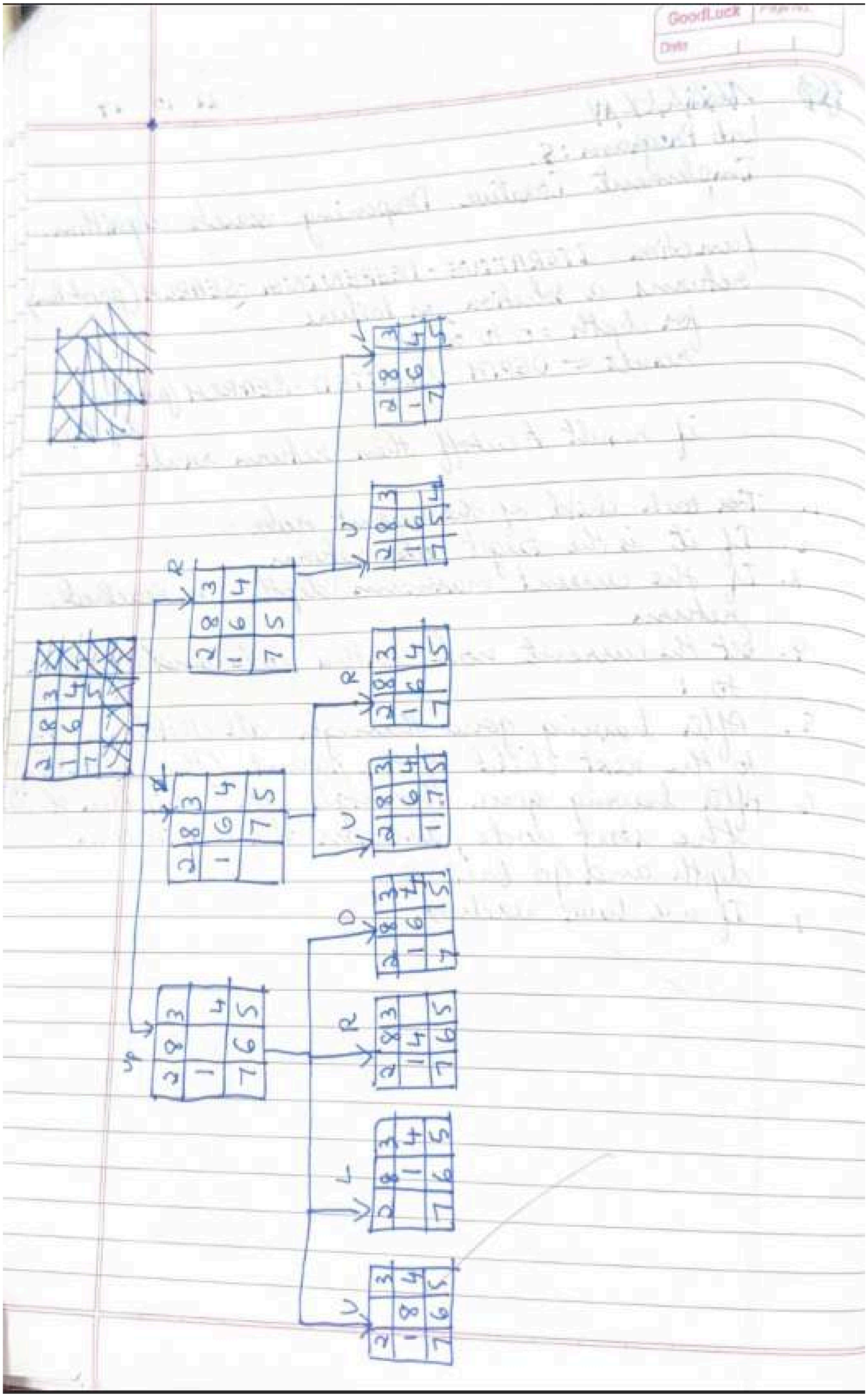
~~[illegible]~~

Lab Program : 5

Implement Iterative Deepening search algorithm.

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.

1. For each child of the current node
2. If it is the target node, return
3. If the current maximum 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 (the next sibling)
6. After having gone through all children of the start node, increase the maximum depth and go back to.
7. If we have reached



Code:

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

```
from collections import deque

# Goal state for the 8-puzzle
goal_state = [[1, 2, 3],
              [4, 5, 6],
              [7, 8, 0]]

# Possible moves
moves = {'up': (-1, 0), 'down': (1, 0), 'left': (0, -1), 'right': (0, 1)}

# Function to find the position of the blank (0)
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

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

# Function to generate neighbors by moving the blank tile
def get_neighbors(state):
    neighbors = []
    blank_row, blank_col = find_blank(state)
    for move, (dr, dc) in moves.items():
        new_row, new_col = blank_row + dr, blank_col + dc
        if 0 <= new_row < 3 and 0 <= new_col < 3:
            # Create a new state by swapping tiles
            new_state = [row[:] for row in state]
            new_state[blank_row][blank_col], new_state[new_row][new_col] =
new_state[new_row][new_col], new_state[blank_row][blank_col]
            neighbors.append((new_state, move))
    return neighbors

# Function to print the puzzle state
def print_puzzle(state):
    for row in state:
        print(row)
    print()

# BFS algorithm to solve the puzzle
def bfs(start_state):
```

```
    queue = deque([(start_state, [])])  # Queue stores (current state, path to reach it)
    visited = set()  # To avoid revisiting states
    visited.add(tuple(tuple(row) for row in start_state))  # Convert state to a tuple for
hashing

    while queue:
        current_state, path = queue.popleft()

        # Check if the goal is reached
        if is_goal(current_state):
            return current_state, path

        # Explore neighbors
        for neighbor, move in get_neighbors(current_state):
            state_tuple = tuple(tuple(row) for row in neighbor)
            if state_tuple not in visited:
                visited.add(state_tuple)
                queue.append((neighbor, path + [move]))

    return None, None

# Function to get user input for the initial state
def get_user_input():
    print("Enter the initial state of the 8-puzzle (row by row):")
    state = []
    for i in range(3):
        while True:
            try:
                row = list(map(int, input(f"Enter row {i+1} (3 integers, space-separated):
").split()))
                if len(row) != 3 or any(x not in range(9) for x in row):
                    raise ValueError
                state.append(row)
                break
            except ValueError:
                print("Invalid input. Please enter 3 integers between 0 and 8.")
    return state

# Function to demonstrate the solution step by step
def demonstrate_solution(start_state, solution_path):
    current_state = start_state
    print("Initial state:")
    print_puzzle(current_state)
    for move in solution_path:
        print(f"Move: {move}")
        for neighbor, move_name in get_neighbors(current_state):
            if move_name == move:
```

```
            current_state = neighbor
            print_puzzle(current_state)
            break

# Main function
if __name__ == "__main__":
    start_state = get_user_input()
    final_state, solution_path = bfs(start_state)
    if solution_path:
        print("Solution found. Steps are demonstrated below:")
        demonstrate_solution(start_state, solution_path)
    else:
        print("No solution found.")
```

Output:

```
Enter the initial state of the 8-puzzle (row by row):
Enter row 1 (3 integers, space-separated): 1 2 3
Enter row 2 (3 integers, space-separated): 0 4 6
Enter row 3 (3 integers, space-separated): 7 5 8
Solution found. Steps are demonstrated below:
Initial state:
[1, 2, 3]
[0, 4, 6]
[7, 5, 8]

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

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

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

```
Enter the initial state of the 8-puzzle (row by row):
Enter row 1 (3 integers, space-separated): 1 3 2
Enter row 2 (3 integers, space-separated): 4 6 0
Enter row 3 (3 integers, space-separated): 7 5 8
No solution found.
```

2.2 Implement Iterative deepening search algorithm:

```
from copy import deepcopy

# Directions for moving the blank space (0): up, down, left, right
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

    # Find the position of the blank (0)
    def get_blank_position(self):
        for i in range(3):
            for j in range(3):
                if self.board[i][j] == 0:
                    return i, j

    # Generate successor states by moving the blank space
    def generate_successors(self):
        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

    # Check if the current state matches the goal state
    def is_goal(self, goal_state):
        return self.board == goal_state

    # String representation of the puzzle state
    def __str__(self):
        return "\n".join([" ".join(map(str, row)) for row in self.board])

# Depth-limited search (DLS)
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

# Iterative deepening search (IDS)
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

# Get user input for start and goal states
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

# Main function
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.")
```

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

Output:

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

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

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

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
```

Program 3

Implement A* search algorithm

Algorithm:

Date: 15/10/24

Lab - 3

For 8 puzzle problem using A* implementation to calculate $f(n)$ using

a) $g(n)$ = depth of a node

$h(n)$ = heuristic value ⇒ no of misplaced tiles.

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

b) $g(n)$ = depth

$h(n)$ = heuristic value ⇒ manhattan distance

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

Draw the state space diagram for

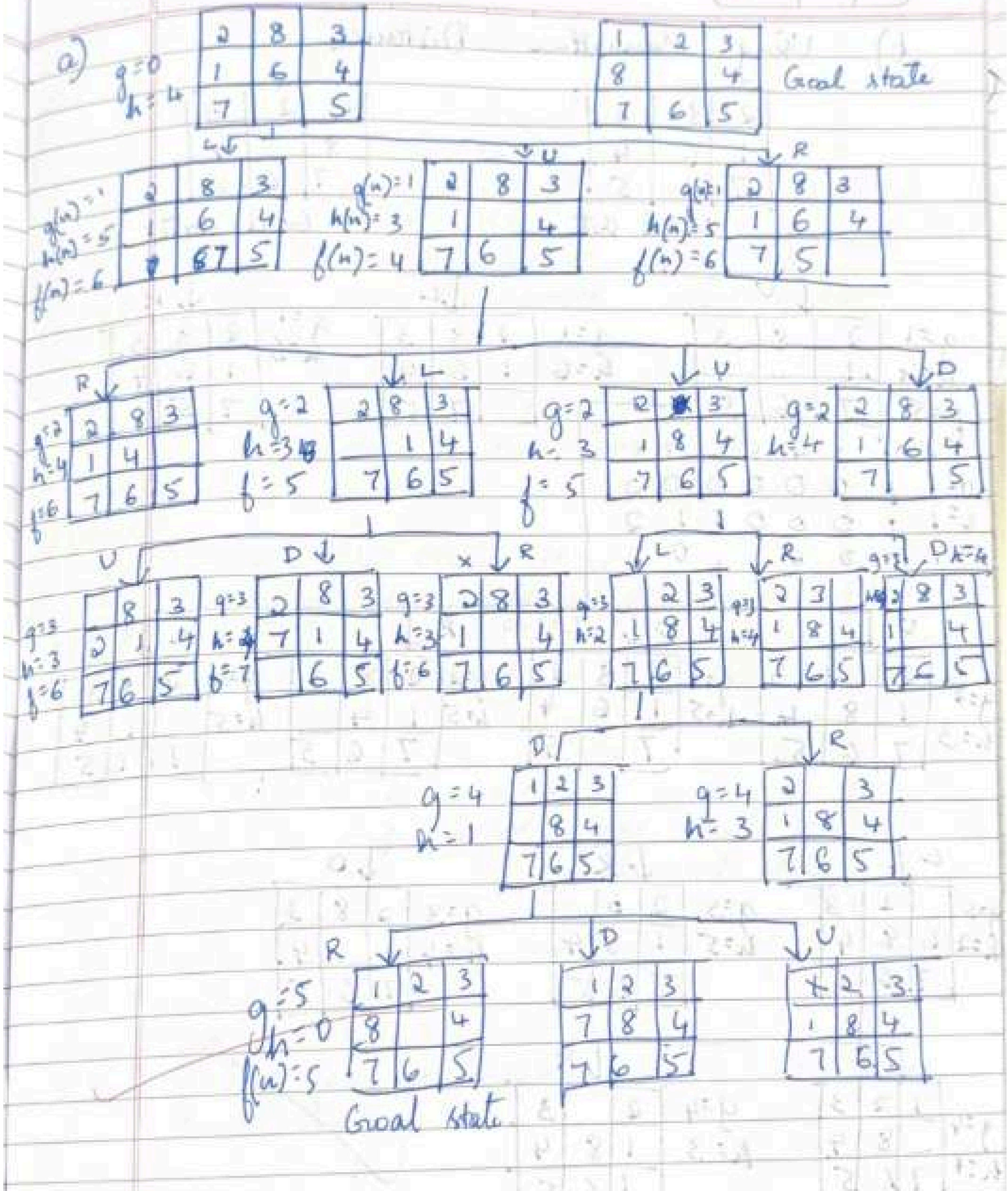
2	8	3
1	6	4
7		5

Initial state

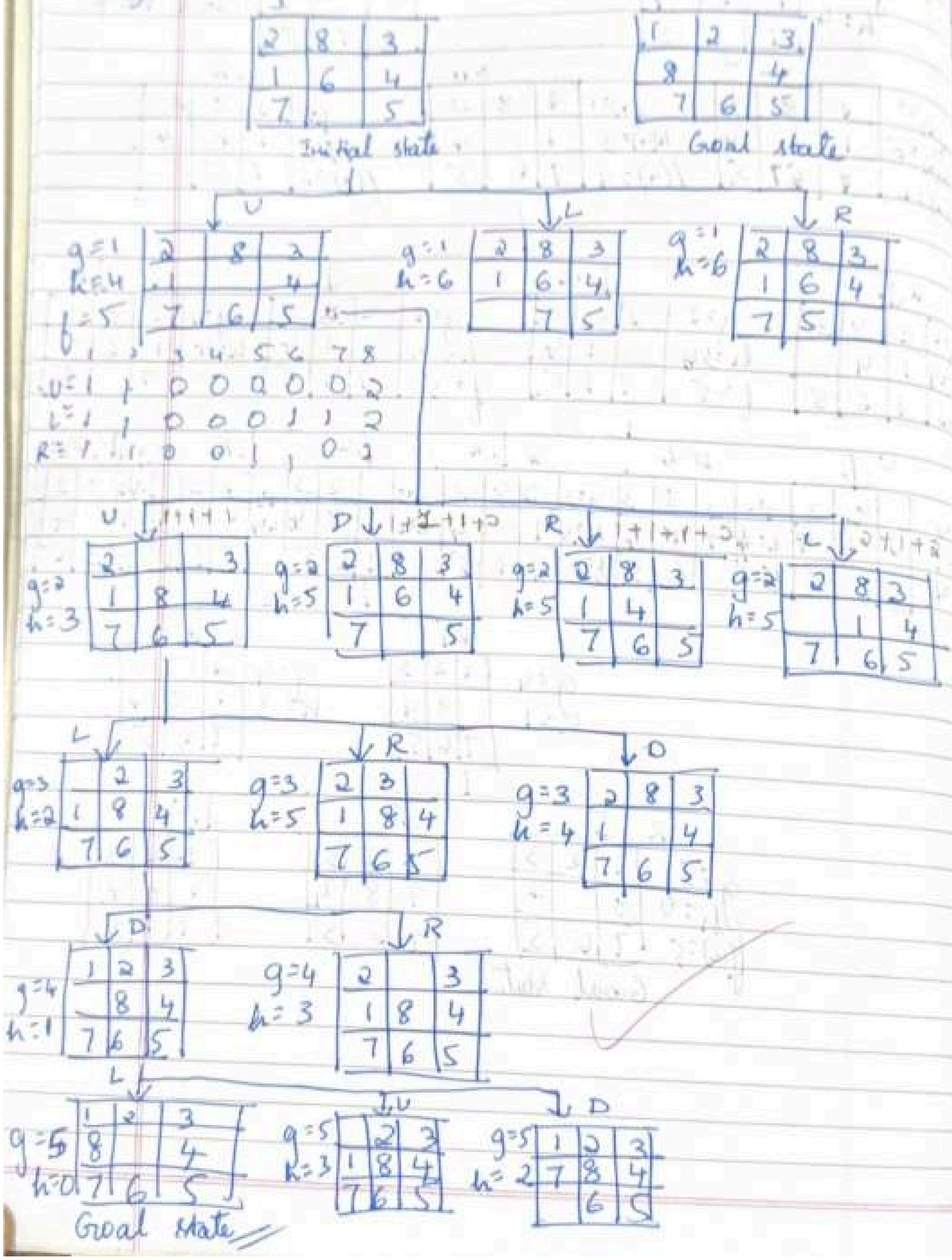
1	2	3
8		4
7	6	5

Goal state.

a)


g=0
h=4
Goal state
L
U
R
g(n)=1
h(n)=5
f(n)=6
g(n)=1
h(n)=3
f(n)=4
g(n)=1
h(n)=5
f(n)=6
R
L
U
D
g=2
h=4
f=6
g=2
h=3
f=5
g=2
h=3
f=5
g=2
h=4
U
D
R
L
R
D
g=3
h=3
f=6
g=3
f=7
g=3
h=3
f=6
g=3
h=2
g=3
h=4
h=4
D
R
g=4
h=1
g=4
h=3
R
D
U
g=5
h=0
f(n)=5
Goal state

b) Using Manhattan Distance



Algorithm for heuristic :

1. Initialize open list and closed list.
   open list - States to be explored.
   Closed list - States that are already explored.
2. Define function that is $g(n)$, $h(n)$, ~~f~~ and $f(n)$ where $h(n)$ is no misplaced tiles.
3. Initialize open list with start state.
   Start $g(n) = 0$.
4. Select $f(n)$ which is least and continue with the procedure. Where $f(n) = g(n) + h(n)$
5. If goal is reached then stop the procedure.

Algorithm for Manhattan.

1. Initialize open list and closed list where open list is states that are to be explored and closed list is states that are already explored.
2. Define functions where
   $g(n)$ is depth of node
   $h(n)$ is number of moves and $f(n)$
3. Initialize open list with start state
   Start $g(n) = 0$.
4. Select $f(n)$ which has lowest value and continue with the procedure
   where $f(n) = g(n) + h(n)$
5. If the goal is reached then stop the procedure

(c) ~~(i)~~ O/P

Initial state

$$\begin{bmatrix} 2 & 8 & 3 \\ 1 & 6 & 4 \\ 7 & 0 & 5 \end{bmatrix}$$

Goal state

$$\begin{bmatrix} 1 & 2 & 3 \\ 8 & 0 & 4 \\ 7 & 6 & 5 \end{bmatrix}$$

Sol found with cost : 5

steps

$$\begin{bmatrix} 2 & 8 & 3 \\ 1 & 6 & 4 \\ 7 & 0 & 5 \end{bmatrix}$$

$$\begin{bmatrix} 2 & 8 & 3 \\ 1 & 0 & 4 \\ 7 & 6 & 5 \end{bmatrix}$$

$$\begin{bmatrix} 2 & 0 & 3 \\ 1 & 8 & 4 \\ 7 & 6 & 5 \end{bmatrix}$$

$$\begin{bmatrix} 1 & 2 & 3 \\ 0 & 8 & 4 \\ 7 & 6 & 5 \end{bmatrix}$$

$$\begin{bmatrix} 1 & 2 & 3 \\ 8 & 0 & 4 \\ 7 & 6 & 5 \end{bmatrix}$$

Code:

Heuristic

```
import heapq

# Define the goal state as a tuple of tuples
GOAL_STATE = ((1, 2, 3),
              (8, 0, 4),
              (7, 6, 5))

# Heuristic: Count the number of misplaced tiles
def misplaced_tile(state):
    misplaced = 0
    for i in range(3):
        for j in range(3):
            if state[i][j] != 0 and state[i][j] != GOAL_STATE[i][j]:
                misplaced += 1
    return misplaced

# Find the position of the blank tile (0)
def find_blank(state):
    for i in range(3):
        for j in range(3):
            if state[i][j] == 0:
                return i, j

# Generate neighbors by moving the blank tile
def generate_neighbors(state):
    neighbors = []
    x, y = find_blank(state)
    directions = [(0, 1), (0, -1), (1, 0), (-1, 0)]  # Right, Left, Down, Up
    for dx, dy in directions:
        nx, ny = x + dx, y + dy
        if 0 <= nx < 3 and 0 <= ny < 3:
            # Create a new state by swapping tiles
            new_state = [list(row) for row in state]
            new_state[x][y], new_state[nx][ny] = new_state[nx][ny], new_state[x][y]
            neighbors.append(tuple(tuple(row) for row in new_state))
    return neighbors

# Reconstruct the path from the start state to the goal state
def reconstruct_path(came_from, current):
    path = [current]
    while current in came_from:
        current = came_from[current]
        path.append(current)
```

```
    path.reverse()
    return path

# A* search algorithm
def a_star(start):
    open_list = []
    heapq.heappush(open_list, (0 + misplaced_tile(start), 0, start))  # (f(n), g(n), state)
    g_score = {start: 0}  # Cost from start to the current state
    came_from = {}
    visited = set()

    while open_list:
        _, g, current = heapq.heappop(open_list)

        # Check if we have reached the goal
        if current == GOAL_STATE:
            path = reconstruct_path(came_from, current)
            return path, g

        visited.add(current)

        # Explore neighbors
        for neighbor in generate_neighbors(current):
            if neighbor in visited:
                continue

            tentative_g = g_score[current] + 1  # Each move has a cost of 1

            # If this path is better, update scores and add to the queue
            if tentative_g < g_score.get(neighbor, float('inf')):
                came_from[neighbor] = current
                g_score[neighbor] = tentative_g
                f_score = tentative_g + misplaced_tile(neighbor)  # f(n) = g(n) + h(n)
                heapq.heappush(open_list, (f_score, tentative_g, neighbor))

    return None, None  # No solution found

# Print a given puzzle state
def print_state(state):
    for row in state:
        print(row)
    print()

# Main function
if __name__ == "__main__":
    start_state = ((2, 8, 3),
                   (1, 6, 4),
```

```
        (7, 0, 5))

print("Initial State:")
print_state(start_state)
print("Goal State:")
print_state(GOAL_STATE)

solution, cost = a_star(start_state)
if solution:
    print(f"Solution found with cost: {cost}")
    print("Steps:")
    for step in solution:
        print_state(step)
else:
    print("No solution found.")
```

Output:

```
Initial State:
(2, 8, 3)
(1, 6, 4)
(7, 0, 5)

Goal State:
(1, 2, 3)
(8, 0, 4)
(7, 6, 5)

Solution found with cost: 5
Steps:
(2, 8, 3)
(1, 6, 4)
(7, 0, 5)

(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)
```

Manhattan:

```
import heapq

# Define the goal state as a tuple of tuples
GOAL_STATE = ((1, 2, 3),
              (8, 0, 4),
              (7, 6, 5))

# Heuristic: Calculate the Manhattan distance for each tile
def manhattan_distance(state):
    distance = 0
    for i in range(3):
        for j in range(3):
            value = state[i][j]
            if value != 0:
                goal_x, goal_y = divmod(value - 1, 3)  # Find the goal position of the current tile
                distance += abs(goal_x - i) + abs(goal_y - j)
    return distance

# Find the position of the blank tile (0)
def find_blank(state):
    for i in range(3):
        for j in range(3):
            if state[i][j] == 0:
                return i, j

# Generate neighbors by moving the blank tile
def generate_neighbors(state):
    neighbors = []
    x, y = find_blank(state)
    directions = [(0, 1), (0, -1), (1, 0), (-1, 0)]  # Right, Left, Down, Up
    for dx, dy in directions:
        nx, ny = x + dx, y + dy
        if 0 <= nx < 3 and 0 <= ny < 3:
            # Create a new state by swapping tiles
            new_state = [list(row) for row in state]
            new_state[x][y], new_state[nx][ny] = new_state[nx][ny], new_state[x][y]
            neighbors.append(tuple(tuple(row) for row in new_state))
    return neighbors

# Reconstruct the path from the start state to the goal state
def reconstruct_path(came_from, current):
    path = [current]
    while current in came_from:
        current = came_from[current]
        path.append(current)
```

```
    path.reverse()
    return path

# A* search algorithm
def a_star(start):
    open_list = []
    heapq.heappush(open_list, (manhattan_distance(start), 0, start))  # (f(n), g(n), state)
    g_score = {start: 0}  # Cost from start to the current state
    came_from = {}
    visited = set()

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

        # Check if we have reached the goal
        if current == GOAL_STATE:
            path = reconstruct_path(came_from, current)
            return path, g

        visited.add(current)

        # Explore neighbors
        for neighbor in generate_neighbors(current):
            if neighbor in visited:
                continue

            tentative_g = g_score[current] + 1  # Each move has a cost of 1

            # If this path is better, update scores and add to the queue
            if tentative_g < g_score.get(neighbor, float('inf')):
                came_from[neighbor] = current
                g_score[neighbor] = tentative_g
                f_score = tentative_g + manhattan_distance(neighbor)  # f(n) = g(n) + h(n)
                heapq.heappush(open_list, (f_score, tentative_g, neighbor))

    return None, None  # No solution found

# Print a given puzzle state
def print_state(state):
    for row in state:
        print(row)
    print()

# Main function
if __name__ == "__main__":
    start_state = ((2, 8, 3),
                   (1, 6, 4),
```

```
              (7, 0, 5))

    print("Initial State:")
    print_state(start_state)
    print("Goal State:")
    print_state(GOAL_STATE)

    solution, cost = a_star(start_state)
    if solution:
        print(f"Solution found with cost: {cost}")
        print("Steps:")
        for step in solution:
            print_state(step)
    else:
        print("No solution found.")
```

Output:

```
Initial State:
(2, 8, 3)
(1, 6, 4)
(7, 0, 5)

Goal State:
(1, 2, 3)
(8, 0, 4)
(7, 6, 5)

Solution found with cost: 5
Steps:
(2, 8, 3)
(1, 6, 4)
(7, 0, 5)

(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)
```

Program 4

Implement Hill Climbing search algorithm to solve N-Queens problem

Algorithm:

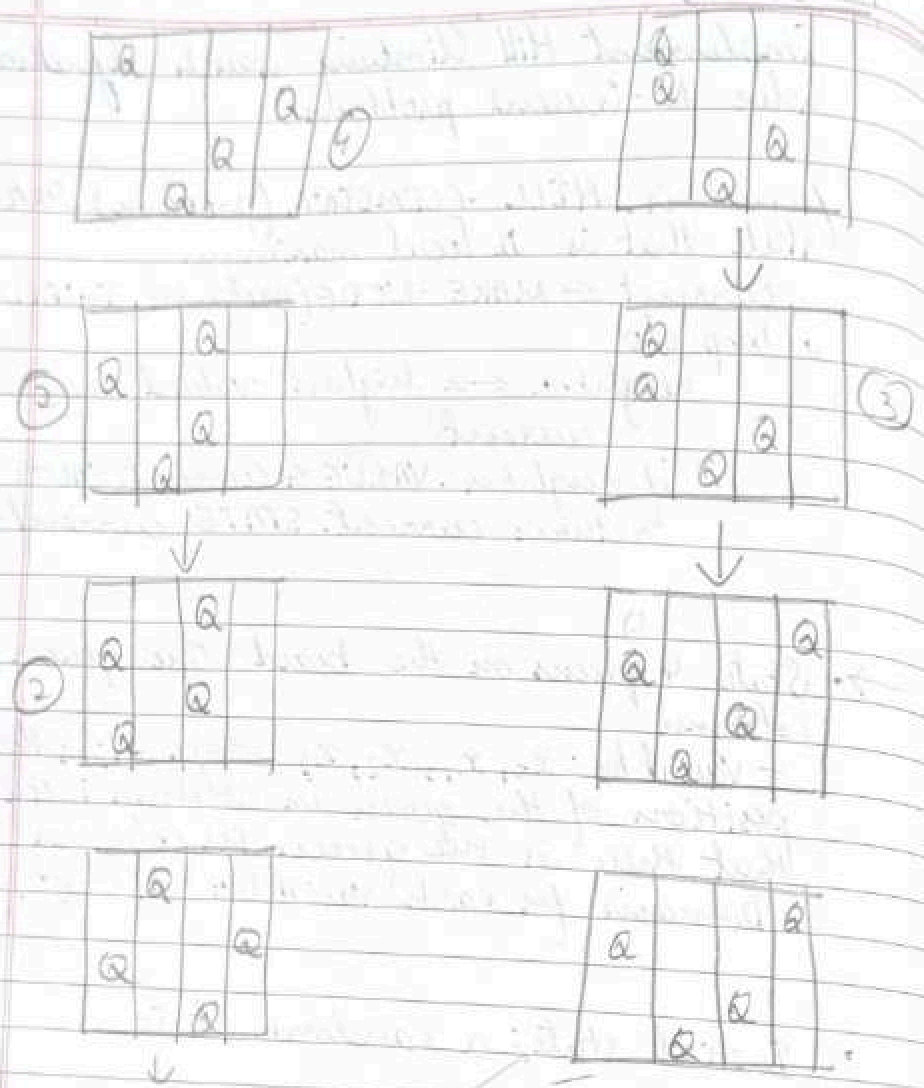
Lab - 5

Implement Hill Climbing search algorithm to solve N-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 returns current.STATE current ← neighbor

→ 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
- Neighbour relation:
  Swap the row position of two queens
- Cost function: The number of pairs of queens attacking each other, directly or indirectly.



Solution

Code:

```
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):
            # Check if queens are in the same column or diagonals
            if state[i] == state[j] or 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]:  # Only generate new states with different columns
                new_state = state[:]
                new_state[row] = col
                successors.append(new_state)
    return successors

def hill_climbing(n):
    """Hill climbing algorithm for N-Queens problem."""
    # Start with a random state
    current = [random.randint(0, n - 1) for _ in range(n)]
    steps = 0

    while True:
        current_attacks = get_attacking_pairs(current)
        successors = generate_successors(current)

        # Find the neighbor with the minimum attacks
        neighbor = min(successors, key=get_attacking_pairs)
        neighbor_attacks = get_attacking_pairs(neighbor)
        steps += 1

        print(f"Step {steps}: Current State: {current}, Attacks: {current_attacks}")

        # If no better neighbor is found, return the current state
        if neighbor_attacks >= current_attacks:
```

```
            return current, current_attacks

        # Move to the better neighbor
        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")

# Set the size of the board
n = 8  # Change this value to test with different board sizes

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

Output:

```
Step 1: Current State: [0, 0, 0, 2], Attacks: 4, Cost: 4
Step 2: Current State: [0, 3, 0, 2], Attacks: 1, Cost: 1
Step 3: Current State: [1, 3, 0, 2], Attacks: 0, Cost: 0
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

Algorithm:

GoodLuck Page No.
Date 29 | 10 | 24

Lab - 5

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^(ΔE/T)
```

The Simulated Annealing Algorithm
The alg can be decomposed in 4 simple steps

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

$$P(x, x_j, T) = \begin{cases} 1 & \text{if } F(x_j) \geq F(x) \\ e^{\frac{F(x_j) - F(x)}{T}} & \text{if } F(x_j) < F(x) \end{cases}$$

4. Reduce T

Output
8 queens
The best position found is [0 8 5 2 6 3 7 4
The no of queens that are not attacking each other is 0

MST
Edges in Minimum Spanning Tree
0 -- 2 (weight = 1)
2 -- 3 (weight = 3)
2 -- 1 (weight = 2)
Total weight

Sudoku using simulated annealing
[[5 3 4 6 7 8 1 9 2
[6 7 2 1 9 5 8 3 4
[1 9 8 3 4 2 5 6 7
8 5 9 7 6 1 4 8 3
- - - —
]]

Code:

```
import numpy as np
from scipy.optimize import dual_annealing

def queens_max(position):
    # This function calculates the number of pairs of queens that are not attacking each other
    position = np.round(position).astype(int)
    n = len(position)
    queen_not_attacking = 0

    for i in range(n - 1):
        for j in range(i + 1, n):
            # Check if queens i and j are not attacking each other
            if position[i] != position[j] and abs(position[i] - position[j]) != (j - i):
                queen_not_attacking += 1

    return -queen_not_attacking  # Return negative for maximization

# Bounds for each queen's position (0 to 7 for an 8x8 chessboard)
bounds = [(0, 7) for _ in range(8)]

# Use dual_annealing for simulated annealing optimization
result = dual_annealing(queens_max, bounds)

# Display the results
best_position = np.round(result.x).astype(int)
best_objective = -result.fun

print('The best position found is:', best_position)
print('The number of queens that are not attacking each other is:', best_objective)
```

Output:

```
The best position found is: [3 5 7 2 0 6 4 1]
The number of queens that are not attacking each other is: 28
```

5.2 SUDOKU PROBLEM

```
import numpy as np
import random
import math

def is_valid(puzzle, row, col, num):
    """Check if a number can be placed at a specific position."""
```

```
    if num in puzzle[row] or num in puzzle[:, col]:
        return False
    box_x, box_y = row // 3 * 3, col // 3 * 3
    if num in puzzle[box_x:box_x + 3, box_y:box_y + 3]:
        return False
    return True

def initial_fill(puzzle):
    """Fill the empty cells in the puzzle with valid random values."""
    filled = puzzle.copy()
    for row in range(9):
        for col in range(9):
            if filled[row][col] == 0:
                possible_values = [num for num in range(1, 10) if is_valid(filled, row, col, num)]
                if possible_values:
                    filled[row][col] = random.choice(possible_values)
    return filled

def objective(puzzle):
    """Calculate the number of conflicts in the Sudoku puzzle."""
    conflicts = 0
    # Row conflicts
    for row in range(9):
        conflicts += 9 - len(set(puzzle[row]))
    # Column conflicts
    for col in range(9):
        conflicts += 9 - len(set(puzzle[:, col]))
    # Box conflicts
    for box_x in range(0, 9, 3):
        for box_y in range(0, 9, 3):
            box = puzzle[box_x:box_x+3, box_y:box_y+3].flatten()
            conflicts += 9 - len(set(box))
    return conflicts

def simulated_annealing(puzzle, max_iter=100000, start_temp=1.0, end_temp=0.01, alpha=0.99):
    """Solve the Sudoku puzzle using simulated annealing."""
    current_state = initial_fill(puzzle)
    current_score = objective(current_state)
    temp = start_temp

    for iteration in range(max_iter):
        if current_score == 0:  # Solution found
            break
        # Randomly pick an empty cell
        row, col = random.randint(0, 8), random.randint(0, 8)
        while puzzle[row][col] != 0:
            row, col = random.randint(0, 8), random.randint(0, 8)
```

```
        # Create a new state with a random value in the chosen cell
        new_state = current_state.copy()
        new_value = random.randint(1, 9)
        if is_valid(new_state, row, col, new_value):
            new_state[row][col] = new_value

        new_score = objective(new_state)
        delta_score = new_score - current_score

        # Accept new state based on simulated annealing criteria
        if delta_score < 0 or random.uniform(0, 1) < math.exp(-delta_score / temp):
            current_state, current_score = new_state, new_score

        # Decrease temperature
        temp *= alpha
        if temp < end_temp:
            break

    return current_state

# Example usage:
puzzle = np.array([
    [5, 3, 0, 0, 7, 0, 0, 0, 0],
    [6, 0, 0, 1, 9, 5, 0, 0, 0],
    [0, 9, 8, 0, 0, 0, 0, 6, 0],
    [8, 0, 0, 0, 6, 0, 0, 0, 3],
    [4, 0, 0, 8, 0, 3, 0, 0, 1],
    [7, 0, 0, 0, 2, 0, 0, 0, 6],
    [0, 6, 0, 0, 0, 0, 2, 8, 0],
    [0, 0, 0, 4, 1, 9, 0, 0, 5],
    [0, 0, 0, 0, 8, 0, 0, 7, 9]
])

solved_puzzle = simulated_annealing(puzzle)
print("Solved Sudoku:\n", solved_puzzle)
```

Output:

```
Solved Sudoku:
 [[5 3 4 2 7 6 9 1 0]
 [6 2 7 1 9 5 4 3 8]
 [1 9 8 3 4 0 5 6 2]
 [8 1 2 7 6 4 0 9 3]
 [4 0 9 8 5 3 7 2 1]
 [7 0 3 9 2 1 8 5 6]
 [3 6 5 0 0 7 2 8 4]
 [2 8 0 4 1 9 3 0 5]
 [0 4 1 5 8 2 6 7 9]]
```

5.3 MST (Minimum Spanning Tree)

```
import random
import math
from collections import defaultdict

class Graph:
    def __init__(self):
        self.edges = defaultdict(list)

    def add_edge(self, u, v, weight):
        """Add an edge to the graph."""
        self.edges[u].append((v, weight))
        self.edges[v].append((u, weight))

    def get_edges(self):
        """Get all edges of the graph."""
        return [(u, v, weight) for u in self.edges for v, weight in self.edges[u] if u < v]

def random_spanning_tree(graph):
    """Generate a random spanning tree using a randomized Prim's-like approach."""
    nodes = list(graph.edges.keys())
    random.shuffle(nodes)
    tree_edges = set()
    selected = {nodes[0]}
    while len(selected) < len(nodes):
        u = random.choice(list(selected))
        candidates = [(v, weight) for v, weight in graph.edges[u] if v not in selected]
        if candidates:
            v, weight = random.choice(candidates)
            tree_edges.add((u, v, weight))
            selected.add(v)
    return tree_edges

def energy(tree):
    """Calculate the total weight of the tree."""
    return sum(weight for u, v, weight in tree)

def generate_neighbor(tree, graph):
    """Generate a neighboring tree by removing and adding an edge."""
    tree_list = list(tree)
    if len(tree_list) < 2:
        return tree

    # Remove a random edge from the tree
    u, v, weight = random.choice(tree_list)
    new_tree = tree - {(u, v, weight)}
```

```
    # Try to add a valid edge to keep the tree connected
    nodes_in_tree = {x for edge in new_tree for x in edge[:2]}
    candidates = [
        (u, v, w) for u in nodes_in_tree for v, w in graph.edges[u]
        if v not in nodes_in_tree and (u, v, w) not in tree and (v, u, w) not in tree
    ]

    if candidates:
        u, v, weight = random.choice(candidates)
        new_tree.add((u, v, weight))

    return new_tree

def simulated_annealing(graph):
    """Find a minimum spanning tree using simulated annealing."""
    T = 1.0
    final_temperature = 0.001
    cooling_factor = 0.95

    current_solution = random_spanning_tree(graph)
    best_solution = current_solution

    while T > final_temperature:
        for _ in range(100):
            neighbor = generate_neighbor(current_solution, graph)
            current_energy = energy(current_solution)
            neighbor_energy = energy(neighbor)

            # Accept neighbor if it's better or probabilistically
            if neighbor_energy < current_energy or random.random() < math.exp((current_energy - 
neighbor_energy) / T):
                current_solution = neighbor

            # Update the best solution found
            if energy(current_solution) < energy(best_solution):
                best_solution = current_solution

        # Cool down
        T *= cooling_factor

    return best_solution

if __name__ == "__main__":
    random.seed(42)
    graph = Graph()
    edges = [(0, 1, 4), (0, 2, 1), (1, 2, 2), (1, 3, 5), (2, 3, 3)]
```

```
for u, v, weight in edges:
    graph.add_edge(u, v, weight)

mst = simulated_annealing(graph)
print("Edges in the Minimum Spanning Tree:")
for u, v, weight in mst:
    print(f"{u} -- {v} (weight: {weight})")
print("Total weight:", energy(mst))
```

Output:

```
Edges in the Minimum Spanning Tree:
2 -- 0 (weight: 1)
2 -- 1 (weight: 2)
Total weight: 3
```

Program 6

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

Algorithm:

GoodLuck Page No. Date 12 / 11 / 24

Lab - 6

Program - 6

Implementation of Truth - Table enumeration algorithm for deciding propositional entailment i.e, create a knowledge base using propositional logic and show that the given query entails the knowledge base or not

Algorithm:

function T-T- Entails ? (KB α) returns True or false
  inputs : KB, the knowledge base, a sentence in propositional logic α, the query, a sentence in propositional logic
  Symbols ← a list of propositional symbols in KB and α
  return TT - CHECK - ALL (KB, α, symbols, { })

function TT - CHECK - ALL (KB, α, symbols, models) 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})

⇒ Propositional Inference Enumeration Method

α = A∨B $\quad$ KB = (A∨C) ∧ (B∨¬C)

A	B	C	A∨C	B∨¬C	KB	α
false	false	false	false	true	false	false
false	false	true	true	false	false	false
false	~~false~~ true	false	false	true	false	true
false	true	true	true	true	true	true
true	false	false	true	true	true	true
true	false	true	true	false	false	true
true	~~false~~ true	false	true	true	true	true
true	true	true	true	true	true	true

Combination where both KB and α (A∨B) are true:

A	B	C
0	1	1
1	0	0
1	1	0
1	1	1

Code:

```
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'])
    print(f"{'A':<10}{'B':<10}{'C':<10}{'KB':<15}{'alpha':<15}{'KB entails alpha?'}")  # Header for
the truth table
    print("-" * 70)  # Separator for readability

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

```
        # Print the current truth assignment and the results for KB and alpha
        print(f"{str(assignment[0]):<10}{str(assignment[1]):<10}{str(assignment[2]):<10}"
              f"{str(KB_value):<15}{str(alpha_value):<15}{'Yes' if KB_value and alpha_value else
'No'}")

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

    # If no counterexample was found, then KB entails alpha
    return True

# 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? {'Yes' if result else 'No'}")
```

Output:

```
A         B         C         KB             alpha          KB entails alpha?
----------------------------------------------------------------------
False     False     False     False          False          No
False     False     True      False          False          No
False     True      False     False          True           No
False     True      True      True           True           Yes
True      False     False     True           True           Yes
True      False     True      False          True           No
True      True      False     True           True           Yes
True      True      True      True           True           Yes

Does KB entail alpha? True
```

Program 7

Implement unification in first order logic

Algorithm:

Date: 19 | 11 | 24

Lab - 7

Implement unification in first order logic

Algorithm : Unify (Ψ_1, Ψ_2)

Step 1 : If Ψ_1 or Ψ_2 is a variable or constant, then :
- a) If Ψ_1 or Ψ_2 are identical, then return NIL
- b) Else if Ψ_1 is a variable,
  - a. then if Ψ_1 occurs in Ψ_2, then return FAILURE
  - b. Else return {(Ψ_2 / Ψ_1)}
- c) Else if Ψ_2 is a variable,
  - a. If Ψ_2 occurs in Ψ_1 then return FAILURE,
  - b. Else return {(Ψ_1 / Ψ_2)}
- d) Else return FAILURE

Step 2 : If the initial Predicate symbol in Ψ_1 and Ψ_2 are not same, then return FAILURE.

Step 3 : If Ψ_1 and Ψ_2 have a different number of arguments, Then return FAILURE

Step 4 : Set Substitution set (SUBST) to NIL

Step 5 : For i = 1 to the number of elements in Ψ_1
- a) Call Unify function with the ith element of Ψ_1 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).

Step 6 : Return SUBST

Eg : P(X, F(y)) → ①
P(a, F(g(x))) → ②

P(a, F(y)) → P(a, F(g(x)))
P(a, F(g(x)))

① & ② are identical if X is replaced with

Ex → $P(x, F(y)) \rightarrow ①$

$P(a, F(g(n)) \rightarrow ②$

① and ② are identical if x is replaced with a in ① a/x

$P(a, F(y)) \rightarrow ①$

if y is replaced with g(n)

$P(a, F(g(n))) \rightarrow ①$

Now ① & ② are same, so they are unified

→ Eg. Eats(x, Apple)

Eats(Riya, y)

x is replaced with Riya

Riya/X

Eats (Riya (Apple)

y is replace with Apple

Apple/y

Eat (Riya, Apple).

→ Eg $g(x, a), f(y)$

$Q(a, \cancel{(n, a)}, f(y)) \rightarrow ①$

$Q(a, g(f(b), a), x) \rightarrow ②$

replace n with f(n) → ①

$Q(a, g(f(n), a), f(y))$

replace x with f(y) → ②

$x/f(y)$

$Q(a, g(f(n), a), f(y))$

Now both are same, they are unified

GoodLuck Page No.
Date

Lab - 8

Forward Reasoning Algorithm

Q $\psi_1 = P(f(a), g(y))$
$\psi_2 = P(x, x)$

1st one is fail

Q $\psi_1 = P(b, x, f(g(z)))$
$\psi_2 = P(z, f(y), f(y))$.

2nd one is pass

Iteration 1:
Attempting to unify : $P(f(a), g(y))$ & $P(x, x)$
current substitution : empty.

Iteration 2:
Attempting to unify : $f(a)$ & x
current substitution : empty
Added substitution : $x \to f(a)$

Attempting to unify : $g(y)$ & x
current substitution : $x \to f(a)$
unification failed : Diff
Predicates and argument length.

Output
Unifying $P(b, x, f(g(z)))$ with $P(z, f(y), f(y))$
Unifying b with z
Substitution : $b \to z$
Unifying $f(g(z))$ with $f(y)$
Substitution : $f(g(z)) \to f(y)$

Final Result :
Unification successful
Substitution : $b \to z$, $x \to f(y)$, $f(g(z) \to f(y)$.

19.11.24

Code:

```
class UnificationError(Exception):
    """Custom exception for unification errors."""
    pass

def occurs_check(var, term, subst):
    """Check if `var` occurs in `term` (to prevent circular substitutions)."""
    if var == term:
        return True
    elif isinstance(term, (list, tuple)):
        return any(occurs_check(var, t, subst) for t in term)
    elif isinstance(term, str) and term in subst:
        return occurs_check(var, subst[term], subst)
    return False

def is_variable(term):
    """Check if `term` is a variable (starting with `?`)."""
    return isinstance(term, str) and term.startswith('?')

def unify(psi1, psi2, subst=None):
    """Attempt to unify two terms, `psi1` and `psi2`, under the given substitution."""
    if subst is None:
        subst = {}

    if psi1 == psi2:
        return subst
    elif is_variable(psi1):
        if psi1 in subst:
            return unify(subst[psi1], psi2, subst)
        elif occurs_check(psi1, psi2, subst):
            raise UnificationError(f"Occurs check failed: {psi1} in {psi2}")
        else:
            subst[psi1] = psi2
            return subst
    elif is_variable(psi2):
        if psi2 in subst:
            return unify(psi1, subst[psi2], subst)
        elif occurs_check(psi2, psi1, subst):
            raise UnificationError(f"Occurs check failed: {psi2} in {psi1}")
        else:
            subst[psi2] = psi1
            return subst
    elif isinstance(psi1, list) and isinstance(psi2, list):
        if psi1[0] != psi2[0]:
            raise UnificationError(f"Predicate symbols don't match: {psi1[0]} != {psi2[0]}")
        if len(psi1) != len(psi2):
```

```
            raise UnificationError(f"Argument lengths don't match: {len(psi1)} != {len(psi2)}")
        for arg1, arg2 in zip(psi1[1:], psi2[1:]):  # Skip the predicate symbol (first element)
            subst = unify(arg1, arg2, subst)
        return subst
    else:
        raise UnificationError(f"Cannot unify {psi1} with {psi2}")

def get_input():
    """Get input from the user and perform unification."""
    try:
        term1 = eval(input("Enter the first term (e.g., ['P', 'b', 'x', ['f', ['g', 'z']]]): "))
        term2 = eval(input("Enter the second term (e.g., ['P', 'z', ['f', 'y'], ['f', 'y']]): "))
        substitution = unify(term1, term2)
        print("Unification successful!")
        print("Substitution:", substitution)
    except UnificationError as e:
        print("Unification failed:", e)
    except Exception as e:
        print("Invalid input or error:", e)

# Run the unification input prompt
get_input()
```

Output:

```
Enter the first term (e.g., ['P', 'b', 'x', ['f', ['g', 'z']]]): ['P',['f',['a']],['g',['?y']]]
Enter the second term (e.g., ['P', 'z', ['f', 'y'], ['f', 'y']]):  ['P','?x','?x']
Unification failed: Predicate symbols don't match: g != f
```

+ Code + Text

```
Enter the first term (e.g., ['P', 'b', 'x', ['f', ['g', 'z']]]): ['P', 'b', '?x', ['f', ['g', '?z']]]
Enter the second term (e.g., ['P', 'z', ['f', 'y'], ['f', 'y']]): ['P', '?z', ['f', '?y'], ['f', '?y']]
Unification successful!
Substitution: {'?z': 'b', '?x': ['f', '?y'], '?y': ['g', '?z']}
```

Program 8

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

Algorithm:

GoodLuck Page No.
Date 26 | 11 | 24

Lab - 8

→ First Order logic

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

$(p_1 \wedge \ldots \wedge p_n \Rightarrow q) \leftarrow$ STANDARDIZE-VARIABLES(rule)

for each θ such that SUBST(θ, $p_1 \wedge \ldots \wedge p_n$) = SUBST(θ, $p_1' \wedge \ldots \wedge p_n'$)

for some $p_1' \ldots p_n'$ in KB

q' ← SUBST(θ, q)

if q' does not unify with some sentence already in KB or new then

add q' to new

φ ← UNIFY(q', α)

if φ is not fail then return φ

add new to KB

return false

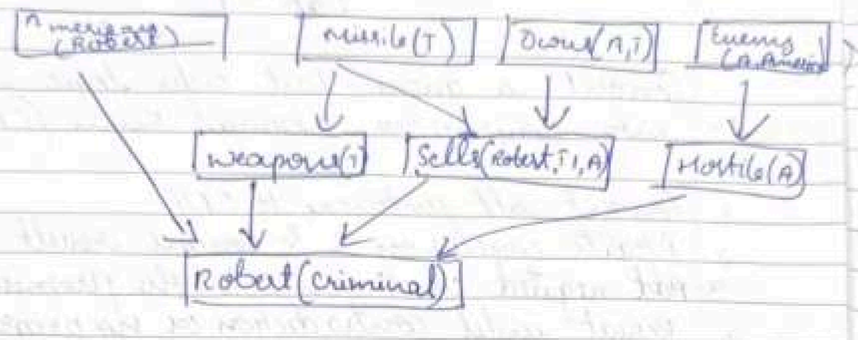
26/11/24

Q3
As per the law, it is a crime for an American to sell weapons to hostile nations. Country A, an enemy of America, has some missiles were sold to it by Robert, who is an American citizen".

Prove that "Robert is criminal"

Representation in FOL

It is a crime for American to sell weapons to hostile nations
~~let's say~~
American(p) ∧ Wepon(q) ∧ Sells(p,q,r)
∧ Hostile(r) ⇒ Criminal(p)
Country A has some missiles
∃x Owns(A, x) ∧ Missiles(x)
Existential Instantiation, introducing a new constant T1:
Owns(A, T1)
Missile(T1)
All of the missiles were sold to country A by Robert
∀x Missiles(x) ∧ Owns(A,x) ⇒ Sells(Robert, x, A)
Missiles are weapons
Missile(x) ⇒ weapon(x)
Enemy of America is known as hostile
∀x Enemy(x, America) ⇒ Hostile(x)



Output:
Enter your FOL statements
> American (p) ∧ Weapon (q), Sells (p,q,r) ∧ Hostile(r) => Criminal (p)
> ∃ x owns (A, x) ∧ Missile (x)
> owns (A, T1)
> Missile (T1)
> ∀ x Missile (x) ∧ Owns (A, x) => Sells (Robert, x, A)
> Missile (x) => Weapon (x)
> ∀ x Enemy (x, America) => Hostile (x)
> American (Robert
> Enemy (A, America)
> done
Enter the query to prove Criminal(Robert)
Proven : criminal (Robert

Code:

```
class UnificationError(Exception):
    """Custom exception for unification errors."""
    pass

def occurs_check(var, term, subst):
    """Check if `var` occurs in `term` (to prevent circular substitutions)."""
    if var == term:
        return True
    elif isinstance(term, (list, tuple)):
        return any(occurs_check(var, t, subst) for t in term)
    elif isinstance(term, str) and term in subst:
        return occurs_check(var, subst[term], subst)
    return False

def is_variable(term):
    """Check if `term` is a variable (starting with `?`)."""
    return isinstance(term, str) and term.startswith('?')

def unify(psi1, psi2, subst=None):
    """Attempt to unify two terms, `psi1` and `psi2`, under the given substitution."""
    if subst is None:
        subst = {}

    if psi1 == psi2:
        return subst
    elif is_variable(psi1):
        if psi1 in subst:
            return unify(subst[psi1], psi2, subst)
        elif occurs_check(psi1, psi2, subst):
            raise UnificationError(f"Occurs check failed: {psi1} in {psi2}")
        else:
            subst[psi1] = psi2
            return subst
    elif is_variable(psi2):
        if psi2 in subst:
            return unify(psi1, subst[psi2], subst)
        elif occurs_check(psi2, psi1, subst):
            raise UnificationError(f"Occurs check failed: {psi2} in {psi1}")
        else:
            subst[psi2] = psi1
            return subst
    elif isinstance(psi1, list) and isinstance(psi2, list):
        if psi1[0] != psi2[0]:
            raise UnificationError(f"Predicate symbols don't match: {psi1[0]} != {psi2[0]}")
        if len(psi1) != len(psi2):
```

```
            raise UnificationError(f"Argument lengths don't match: {len(psi1)} != {len(psi2)}")
        for arg1, arg2 in zip(psi1[1:], psi2[1:]):  # Skip the predicate symbol (first element)
            subst = unify(arg1, arg2, subst)
        return subst
    else:
        raise UnificationError(f"Cannot unify {psi1} with {psi2}")

def get_input():
    """Get input from the user and perform unification."""
    try:
        term1 = eval(input("Enter the first term (e.g., ['P', 'b', 'x', ['f', ['g', 'z']]]): "))
        term2 = eval(input("Enter the second term (e.g., ['P', 'z', ['f', 'y'], ['f', 'y']]): "))
        substitution = unify(term1, term2)
        print("Unification successful!")
        print("Substitution:", substitution)
    except UnificationError as e:
        print("Unification failed:", e)
    except Exception as e:
        print("Invalid input or error:", e)

# Run the unification input prompt
get_input()
```

Output:

```
Enter the rules:
Enter rule (or 'done' to finish): American(p) AND Weapon(q) AND Sells(p, q, r) AND Hostile(r) => Criminal(p)
Enter rule (or 'done' to finish): ∃x (Owns(A, x) AND Missile(x)) => Missile(x) AND Weapon(x)
Enter rule (or 'done' to finish): ∀x(Missile(x) AND Owns(A, x)) => Sells(Robert, x, A)
Enter rule (or 'done' to finish): Missile(x) => Weapon(x)
Enter rule (or 'done' to finish): ∀x (Enemy(x, America)) => Hostile(x)
Enter rule (or 'done' to finish): done

Enter the facts:
Enter fact (or 'done' to finish): American(Robert)
Enter fact (or 'done' to finish): Enemy(A, America)
Enter fact (or 'done' to finish): Owns(A, T1)
Enter fact (or 'done' to finish):  Missile(T1)
Enter fact (or 'done' to finish):  done

Enter the query:
Enter the query: Criminal(Robert)

Final facts:
{'Enemy(A, America)', 'Owns(A, T1)', 'American(Robert)', 'Missile(T1)'}

Query 'Criminal(Robert)' inferred: True
```

Program 9

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

Algorithm:

Date 06 | 11 | 24

Lab - 9

Convert a given first order logic statement into Conjunctive Normal Form (CNF)

1. Convert all sentences to CNF
2. Negate conclusion S & convert result to CNF
3. Add negated 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 ~~resolved~~ resolvent is the empty clause, a contradiction has been found (i.e., S follows from the premises)
   d. If not, add resolvent to the premises.

If we succeed in Step 4, we have proved the conclusion.

Give KB or Premises:

a. John likes all kind 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
h. Prove by res that?

Representation in FOL

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. $likes(John, Peanuts)$

Eliminate implication $\alpha \to \beta$ with $\neg\alpha \vee \beta$

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

Move negation ($\neg$) inwards and rewrite

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

Rename variables or standardize variables

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

Drop Universal Quantifier

a. ¬ food (x) v likes (John, x)

b. food (apple)

c. food (vegetables)

d. ¬ eats (y, z) v killed (y) v food (z)

e. eats (Anil, Peanuts)

f. alive (Anil)

g. ¬ eats (Anil, w) v eats (Harry, w)

h. killed (g) v alive (g)

i. ¬ alive (k) v ¬ killed (k)

j. likes (John, Peanuts)

Proof by Resolution

¬ likes (John, Peanuts) ——— ¬ food (x) v likes (John, x)

{Peanuts/x}

¬ food (Peanuts) ——— ¬ eats (y, z) v killed (y) v food (z)

{Peanuts/z}

¬ eats (y, Peanuts) v killed (y) ——— eats (Anil, Peanuts)

{Anil/y}

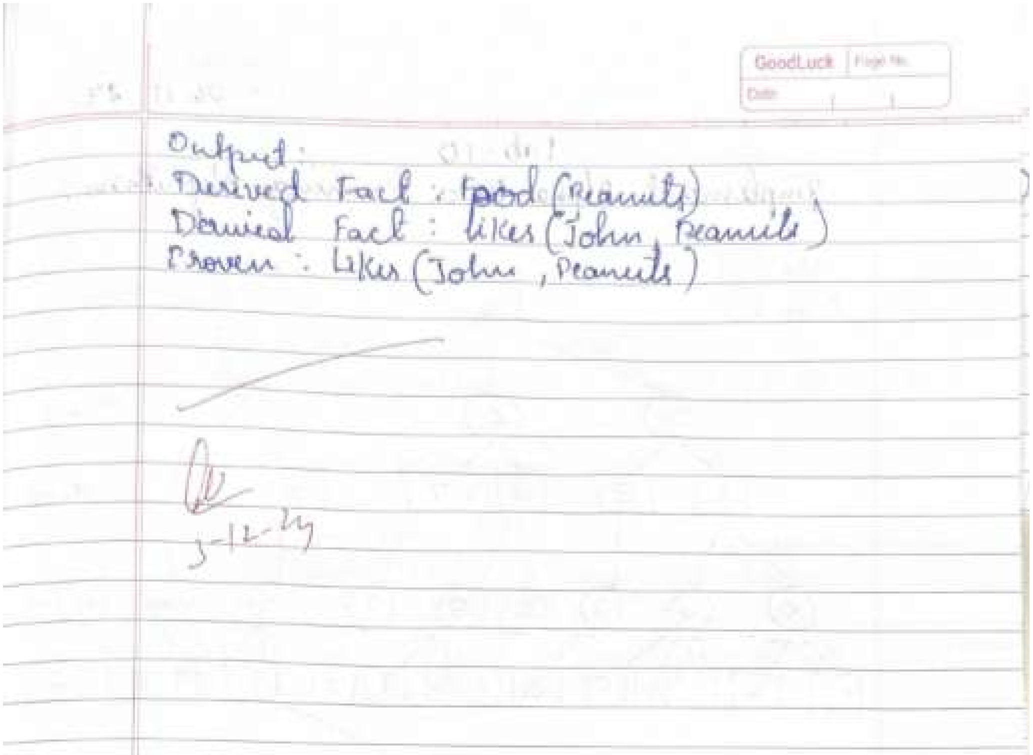
killed (Anil) ——— ¬ alive (k) v ¬ killed (k)

{Anil/k}

¬ alive (Alive) ——— alive (Anil)

{}

Hence proved



Output:
Derived Fact : food(Peanuts)
Derived Fact : likes(John, Peanuts)
Proven : likes(John, Peanuts)

3-12-24

Code:

```
import re

class ForwardReasoning:
    def __init__(self, rules, facts):
        self.rules = rules  # List of rules (condition -> result)
        self.facts = set(facts)  # Known facts

    def match_condition(self, condition):
        """Check if all conditions match the current facts."""
        variable_map = {}
        for cond in condition:
            fact_match = False
            for fact in self.facts:
                if self.match_fact(cond, fact, variable_map):
                    fact_match = True
                    break
            if not fact_match:
                return False, variable_map
        return True, variable_map

    def match_fact(self, cond, fact, variable_map):
        """Match a condition to a fact, handling variables."""
        var_pattern = re.compile(r'\b\?[a-zA-Z]+\b')  # Match variables starting with '?'
```

```
        condition_parts = cond.split()
        fact_parts = fact.split()

        if len(condition_parts) != len(fact_parts):
            return False

        for cond_part, fact_part in zip(condition_parts, fact_parts):
            if var_pattern.match(cond_part):  # If it's a variable
                if cond_part not in variable_map:
                    variable_map[cond_part] = fact_part
                elif variable_map[cond_part] != fact_part:
                    return False
            elif cond_part != fact_part:  # If it's a constant, they must match
                return False
        return True

    def infer(self, query):
        """Forward chaining algorithm to infer if the query can be derived."""
        applied_rules = True
        while applied_rules:
            applied_rules = False
            for condition, result in self.rules:
                matched, variable_map = self.match_condition(condition)
                if matched and result not in self.facts:
                    self.facts.add(result)  # Add the result to known facts
                    applied_rules = True
                    print(f"Applied rule: {condition} -> {result}")
                    # If the query is inferred, return True immediately
                    if self.match_fact(query, result, variable_map):
                        return True
        # Return True if the query is in facts after the reasoning process, else False
        return query in self.facts

def get_input_rules():
    """Get input for rules from the user."""
    rules = []
    while True:
        rule = input("Enter rule (or 'done' to finish): ").strip()
        if rule.lower() == "done":
            break
        if "=>" in rule:
            condition_str, result = rule.split("=>")
            conditions = set(condition_str.strip().split(" AND "))
            result = result.strip()
            rules.append((conditions, result))
    return rules
```

```
def get_input_facts():
    """Get input for facts from the user."""
    facts = set()
    while True:
        fact = input("Enter fact (or 'done' to finish): ").strip()
        if fact.lower() == "done":
            break
        facts.add(fact)
    return facts

def get_input_query():
    """Get input for the query from the user."""
    return input("Enter the query: ").strip()

# Main program to run the forward reasoning
def main():
    print("Enter the rules:")
    rules = get_input_rules()
    print("\nEnter the facts:")
    facts = get_input_facts()
    print("\nEnter the query:")
    query = get_input_query()

    # Initialize and run forward reasoning
    reasoner = ForwardReasoning(rules, facts)
    result = reasoner.infer(query)

    # Output results
    print("\nFinal facts:")
    print(reasoner.facts)
    print(f"\nQuery '{query}' inferred: {result}")

# Call the main function to start
main()
```

Output:

```
Does John like peanuts? Yes
```

Program 10

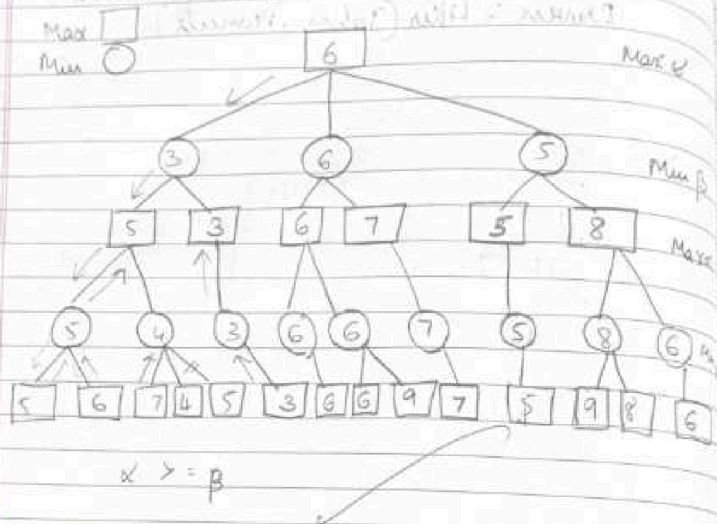
Implement Alpha-Beta Pruning.

Algorithm:

GoodLuck Page No.
Date 26/11/24

Lab-10.

Implement Alpha Beta Pruning Algorithm.



Algorithm

function MINIMAX-DECISION(state) returns an action
 return arg max a ∈ ACTIONS(s) MIN-VALUE(RESULT(state, a))

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

function MIN-VALUE(state) returns a utility value
 if TERMINAL-TEST(state) then return UTILITY(state)
 v ← ∞

for each a in ACTIONS(state) do
    v ← MIN(v, MAX-VALUE(RESULT(s,a)))
return v

Output :-
Enter depth of the tree : 3
Enter leaf values : -1, 8, -3, -1, 2, 1, 3, 4.
Optimal value : 2.

5/12-24

Code:

```
import math

def minimax(node, depth, is_maximizing):
    """
    Implement the Minimax algorithm to solve the decision tree.

    Parameters:
    node (dict): The current node in the decision tree, with the following structure:
    {
        'value': int,
        'left': dict or None,
        'right': dict or None
    }
    depth (int): The current depth in the decision tree.
    is_maximizing (bool): Flag to indicate whether the current player is the maximizing player.

    Returns:
    int: The utility value of the current node.
    """
    # Base case: Leaf node
    if node['left'] is None and node['right'] is None:
```

```
        return node['value']

    # Recursive case
    if is_maximizing:
        best_value = -math.inf
        if node['left']:
            best_value = max(best_value, minimax(node['left'], depth + 1, False))
        if node['right']:
            best_value = max(best_value, minimax(node['right'], depth + 1, False))
        return best_value
    else:
        best_value = math.inf
        if node['left']:
            best_value = min(best_value, minimax(node['left'], depth + 1, True))
        if node['right']:
            best_value = min(best_value, minimax(node['right'], depth + 1, True))
        return best_value

# Example usage
decision_tree = {
    'value': 5,
    'left': {
        'value': 6,
        'left': {
            'value': 7,
            'left': {
                'value': 4,
                'left': None,
                'right': None
            },
            'right': {
                'value': 5,
                'left': None,
                'right': None
            }
        },
        'right': {
            'value': 3,
            'left': {
                'value': 6,
                'left': None,
                'right': None
            },
            'right': {
                'value': 9,
                'left': None,
                'right': None
```

```
            }
        }
    },
    'right': {
        'value': 8,
        'left': {
            'value': 7,
            'left': {
                'value': 6,
                'left': None,
                'right': None
            },
            'right': {
                'value': 9,
                'left': None,
                'right': None
            }
        },
        'right': {
            'value': 8,
            'left': {
                'value': 6,
                'left': None,
                'right': None
            },
            'right': None
        }
    }
}

# Find the best move for the maximizing player
best_value = minimax(decision_tree, 0, True)
print(f"The best value for the maximizing player is: {best_value}")
```

Output:

```
The best value for the maximizing player is: 6
```