VISVESVARAYA TECHNOLOGICAL UNIVERSITY

“JnanaSangama”, Belgaum -590014, Karnataka.

LAB REPORT
on

Artificial Intelligence (23CS5PCAIN)

Submitted by

Preeti T Korishettar (1BM22CS208)

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

Swathi Sridharan
Professor
Department of 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 outby **Preeti T Korishettar (1BM22CS208),** 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.

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

Index

Sl. No.	Date	Experiment Title	Page No.
1	24-9-2024	Implement Tic –Tac –Toe Game Implement vacuum cleaner agent	4-10
2	8-10-2024	Implement 8 puzzle problems using Depth First Search (DFS) Implement Iterative deepening search algorithm	11-17
3	15-10-2024	Implement A* search algorithm	18-24
4	22-10-2024	Implement Hill Climbing search algorithm to solve N-Queens problem	25-27
5	29-10-2024	Simulated Annealing to Solve 8-Queens problem	28-31
6	12-11-2024	Create a knowledge base using propositional logic and show that the given query entails the knowledge base or not.	32-35
7	19-11-2024	Implement unification in first order logic	36-39
8	3-12-2024	Create a knowledge base consisting of first order logic statements and prove the given query using forward reasoning.	40-43
9	3-12-2024	Create a knowledge base consisting of first order logic statements and prove the given query using Resolution	44-47
10	17-12-2024	Implement Alpha-Beta Pruning.	48-52

Github Link :https://github.com/Preeti-T-Korishettar/1BM22CS208_AI_LAB

Program 1

Implement Tic –Tac –Toe Game
Implement vacuum cleaner agent
Tic-Tac-Toe

Algorithm:

B1.
09/24

Bafna Gold
Date: Page:

Tic Tac Toe Game

step 1 : start

step 2 : creation of board 3×3
b[3][3] ;

step 3 : Allocate AI with O and player with x

step 4 : ~~All~~ Take alternate input from AI and player.

step 5 : To win()

```
for i in range(3):
    if b[i][0] == b[i][1] == b[i][2] != ' ' :
    return b[i][0]
    if b[0][i] == b[1][i] == b[2][i] != ' '
    return b[0][i]
    if b[0][0] == b[1][1] == b[2][2] != ' '
    return b[0][0]
    if b[0][2] == b[1][2] == b[2][2] != ' '
    return b[0][2].

return -1
```

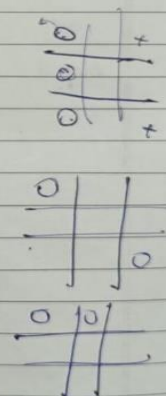
step 6 : Draw()

```
if ( is_b_full() && either of players not won)
Draw has occurred
else
continuing playing
```

step 7 : If any of the two cells are filled and are same then return the middle position which is empty. return position. else call to_win

step 8 : take input from AI and player alternatively.
if (pos == -1)
print (~~win~~ AI / player win)

step 9 : stop.



CODE:

```
# 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 a player's move
def player_move(icon):
    if icon == 'X':
        number = 1
    elif icon == 'O':
        number = 2
    print("Your turn player {}".format(number))
    while True:
        try:
            choice = int(input("Enter your move (1-9): "))
            if choice < 1 or choice > 9:
                print("Invalid input! Choose a number between 1 and 9.")
            elif board[choice - 1] != ' ':
                print("That space is taken! Choose another.")
            else:
                board[choice - 1] = icon
                break
        except ValueError:
            print("Invalid input! Please enter a number between 1 and 9.")

# 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
    return False

# Function to check for draw
def is_draw():
    return ' ' not in board

# Main game function
def play_game():
    draw_board()
    while True:
        # Player 1 move
        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 2 move
        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:

```
Your turn player 2
Enter your move (1-9): 4

| X |   |   |
| O |   |   |
|   |   |   |

Your turn player 1
Enter your move (1-9): 2

| X | X |   |
| O |   |   |
|   |   |   |

Your turn player 2
Enter your move (1-9): 5

| X | X |   |
| O | O |   |
|   |   |   |

Your turn player 1
Enter your move (1-9): 3

| X | X | X |
| O | O |   |
|   |   |   |

Player 1 wins! Congratulations!

=== Code Execution Successful ===
```

o

Vacuum Cleaner

01/10/24 LAB-2

Vacuum cleaner algorithm

step1 : start
step2 : Take location and status of both the rooms (A and B)
s_A = input ("Enter status of A")
s_B = input ("Enter status of B")
step3 : Declare agent table (Dictonary)
agent_table = {
'clean, A' : Moveright
'clean, B' : Moveleft
'dirty, A' : Suck
'dirty, B' : Suck
} // give starting room.
step4 : check the status of both rooms whether they are dirty
If dirty then clean room A and update status to clean and then moveright.
else move right
continue untill both the rooms are clean.

step 5 : end

1/10/24

CODE:

```
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

def goal(a_status, b_status):
    if a_status == 0 and b_status == 0:
        print("Goal reached")
    else:
        print("Goal not reached")

loc = input("Enter the starting location of the vacuum (A or B): ")
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): "))

if loc == "A":
    cost, loc, a_status = reflex(loc, a_status, cost)
elif loc == "B":
    cost, loc, b_status = reflex(loc, b_status, cost)

cost, loc, b_status = reflex(loc, b_status, cost) if loc == "A" else reflex(loc, a_status, cost)

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

=== Code Execution Successful ===
```

Program 2

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

8 puzzle using DFS

Algorithm:

Bafna Gold
Date: Page:

8 Puzzle problem using DFS

Algorithm:

Step 1 start
step 2. Assign initial state and goal states:

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

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

step3: Calc_manhantlan distance

```
initial_pos = x1, y1
goal_pos = x2, y2

for i in range(3):
    for j in ran (3):
        title = self.state [i][j]
        if title != 0:
            total_dist = (x2 - x1) + (y2 - y1)
```

step4: get_possible moves.

swap up, down, left, right

first find the position of empty space and return the position.

```
for dx, dy in directions:
    new_x, new_y = empty_x + dx, empty_y + dy
    if 0 <= new_x < 3 and 0 <= new_y < 3:

return rows.
```

Step5: dfs_with_manhanttan function.

Implement DFS approach using a stack.

```
while stack:
    node = stack.pop()

    if node.state == goal_state:
        return construct_solution (node)

    visited.add (tuple (map (tuple, node.state)))

return None
```

CODE:

```
count = 0

def print_state(in_array):
    global count
    count += 1
    for row in in_array:
        print(' '.join(str(num) for num in row))
    print()

def helper(goal, in_array, row, col, vis):
    global count
    # Marking current position as visited
    vis[row][col] = 1

    # Directions for movement: up, right, down, left
    drow = [-1, 0, 1, 0]
    dcol = [0, 1, 0, -1]
    dchange = ['Up', 'Right', 'Down', 'Left']

    # Print current state
    print("Current state:")
    print_state(in_array)

    # Check if the current state is the goal state
    if in_array == goal:
        print(f"Number of states: {count}")
        return True

    # Explore all possible directions
    for i in range(4):
        nrow = row + drow[i]
        ncol = col + dcol[i]

        # Check if the new position is within bounds and not visited
        if 0 <= nrow < len(in_array) and 0 <= ncol < len(in_array[0]) and not vis[nrow][ncol]:
            # Make the move (swap the empty space with the adjacent tile)
            print(f"Took a {dchange[i]} move")
            in_array[row][col], in_array[nrow][ncol] = in_array[nrow][ncol], in_array[row][col]

            # Recursive call
            if helper(goal, in_array, nrow, ncol, vis):
                return True

            # Backtrack (undo the move)
            in_array[row][col], in_array[nrow][ncol] = in_array[nrow][ncol], in_array[row][col]
```

```
    # Mark the position as unvisited before returning
    vis[row][col] = 0
    return False

# Example usage
initial_state = [[1, 2, 3], [0, 4, 6], [7, 5, 8]]  # 0 represents the empty space
goal_state = [[1, 2, 3], [4, 5, 6], [7, 8, 0]]
visited = [[0] * 3 for _ in range(3)]  # 3x3 visited matrix
empty_row, empty_col = 1, 0  # Initial position of the empty space

found_solution = helper(goal_state, initial_state, empty_row, empty_col, visited)
print("Solution found:", found_solution)
```

OUTPUT:

```
Took a Down move
Current state:
1 2 3
4 6 8
7 5 0

Took a Left move
Current state:
1 2 3
4 6 8
7 0 5

Took a Left move
Current state:
1 2 3
4 6 8
0 7 5

Took a Down move
Current state:
1 2 3
4 5 6
7 0 8

Took a Right move
Current state:
1 2 3
4 5 6
7 8 0

Number of states: 41
Solution found: True
```

Implement Iterative deepening search algorithm

Algorithm:

15/10/24 LAB - 3

Iterative depenning depth search.

Algorithm

```
bool IDDFS (src, target, max_depth)

    for l from 0 to max
        if DLS( src, target, limit ) = = true
            return true.

    return false.

bool DLS ( src, target, l)
    if (src = = target)
        return true;

    if ( l <= 0 )
        return false;

    foreach adjacent i of src
        if DLS( i, target, l-1)
            return true

    return false;
```

CODE:

```
class PuzzleState:
    def __init__(self, board, zero_pos, moves=0, previous=None):
        self.board = board
        self.zero_pos = zero_pos  # Position of the zero tile
        self.moves = moves  # Number of moves taken to reach this state
        self.previous = previous  # For tracking the path

    def is_goal(self, goal_state):
        return self.board == goal_state

    def get_possible_moves(self):
        moves = []
        x, y = self.zero_pos
        directions = [(-1, 0), (1, 0), (0, -1), (0, 1)]  # Up, Down, Left, Right
        for dx, dy in directions:
            new_x, new_y = x + dx, y + dy
            if 0 <= new_x < 3 and 0 <= new_y < 3:
                # Create a new board configuration
                new_board = [row[:] for row in self.board]
                # Swap the zero tile with the adjacent tile
                new_board[x][y], new_board[new_x][new_y] = new_board[new_x][new_y], new_board[x][y]
                moves.append((new_board, (new_x, new_y)))
        return moves

def ids(initial_state, goal_state, max_depth):
    for depth in range(max_depth):
        visited = set()
        result = dls(initial_state, goal_state, depth, visited)
        if result:
            return result
    return None

def dls(state, goal_state, depth, visited):
    if state.is_goal(goal_state):
        return state
    if depth == 0:
        return None
    visited.add(tuple(map(tuple, state.board)))  # Mark this state as visited
    for new_board, new_zero_pos in state.get_possible_moves():
        if tuple(map(tuple, new_board)) not in visited:
            new_state = PuzzleState(new_board, new_zero_pos, state.moves + 1, state)
            result = dls(new_state, goal_state, depth - 1, visited)
            if result:
                return result
    # Unmark this state
    visited.remove(tuple(map(tuple, state.board)))
```

```
    return None

def print_solution(solution):
    path = []
    while solution:
        path.append(solution.board)
        solution = solution.previous
    for board in reversed(path):
        for row in board:
            print(row)
        print()

# Define the initial state and goal state
initial_state = PuzzleState(
    board=[
        [1, 2, 3],
        [4, 0, 5],
        [7, 8, 6]
    ],
    zero_pos=(1, 1)
)
goal_state = [
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 0]
]

# Perform Iterative Deepening Search
max_depth = 20  # You can adjust this value
solution = ids(initial_state, goal_state, max_depth)

if solution:
    print("Solution found:")
    print_solution(solution)
else:
    print("No solution found.")
```

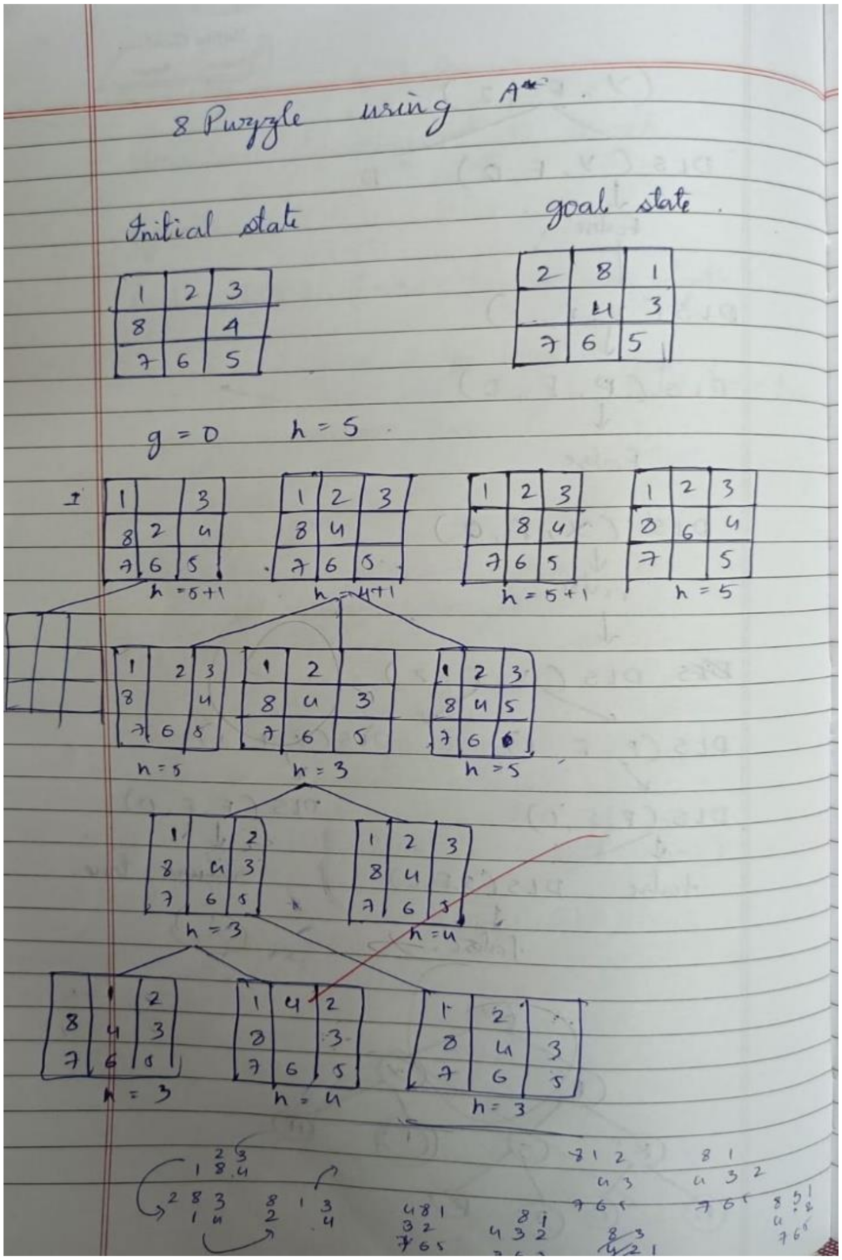
OUTPUT:

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

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

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

Program 3

Implement A* search algorithm

Algorithm:



CODE:

```
from collections import deque

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

def find_empty(state):
    return state.index(0)

def get_neighbors(state):
    neighbors = []
    empty_index = find_empty(state)
    row, col = divmod(empty_index, 3)
    directions = [(-1, 0), (1, 0), (0, 1), (0, -1)]  # Up, Down, Right, Left
    for dr, dc in directions:
        new_row, new_col = row + dr, col + dc
        if 0 <= new_row < 3 and 0 <= new_col < 3:
            new_index = new_row * 3 + new_col
            new_state = list(state)
            # Swap the empty space with the neighbor tile
            new_state[empty_index], new_state[new_index] = new_state[new_index],
new_state[empty_index]
            neighbors.append(tuple(new_state))
    return neighbors

def bfs(initial_state):
    queue = deque([(initial_state, [])])
    visited = set()
    visited.add(initial_state)
    visited_count = 1  # Initialize visited count

    while queue:
        current_state, path = queue.popleft()
        if current_state == GOAL_STATE:
            return path, visited_count  # Return path and count
        for neighbor in get_neighbors(current_state):
            if neighbor not in visited:
                queue.append((neighbor, path + [neighbor]))
                visited.add(neighbor)
                visited_count += 1  # Increment visited count
    return None, visited_count  # Return count if no solution found

def input_start_state():
    while True:
        print("Enter the starting state as 9 numbers (0 for the empty space):")
        input_state = input("Format: 1 2 3 4 5 6 7 8 0\n")
        try:
```

```
            numbers = list(map(int, input_state.split()))
            if len(numbers) != 9 or set(numbers) != set(range(9)):
                raise ValueError
            return tuple(numbers)
        except ValueError:
            print("Invalid input. Please enter numbers from 0 to 8 with no duplicates.")

def print_matrix(state):
    for i in range(0, 9, 3):
        print(state[i:i + 3])

if __name__ == "__main__":  # Corrected main check
    initial_state = input_start_state()
    print("Initial state:")
    print_matrix(initial_state)
    print()

    solution, visited_count = bfs(initial_state)
    print(f"Number of states visited: {visited_count}")
    if solution:
        print("\nSolution found with the following steps:")
        for step in solution:
            print_matrix(step)
            print()
```

OUTPUT:

```
else:
    print("\nNo solution found.")
```

```
Enter the starting state as 9 numbers (0 for the empty space):
Format: 1 2 3 4 5 6 7 8 0
1 2 3 0 4 6  7 5 8
Initial state:
(1, 2, 3)
(0, 4, 6)
(7, 5, 8)

Number of states visited: 29

Solution found with the following steps:
(1, 2, 3)
(4, 0, 6)
(7, 5, 8)

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

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

MISPLACED TILES

CODE:

```
from collections import deque

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

def find_empty(state):
    return state.index(0)

def get_neighbors(state):
    neighbors = []
    empty_index = find_empty(state)
    row, col = divmod(empty_index, 3)
    directions = [(-1, 0), (1, 0), (0, 1), (0, -1)]  # Up, Down, Right, Left
    for dr, dc in directions:
        new_row, new_col = row + dr, col + dc
        if 0 <= new_row < 3 and 0 <= new_col < 3:
            new_index = new_row * 3 + new_col
            new_state = list(state)
            # Swap the empty space with the neighbor tile
            new_state[empty_index], new_state[new_index] = new_state[new_index],
new_state[empty_index]
            neighbors.append(tuple(new_state))
    return neighbors

def misplaced_tiles(state):
    """Count the number of tiles that are not in their goal position."""
    return sum(1 for i in range(9) if state[i] != GOAL_STATE[i] and state[i] != 0)

def bfs_with_heuristic(initial_state):
    queue = deque([(initial_state, [], 0)])  # (state, path, heuristic cost)
    visited = set()
    visited.add(initial_state)
    visited_count = 1  # Initialize visited count

    while queue:
        # Sort the queue by heuristic value (misplaced tiles)
        queue = deque(sorted(queue, key=lambda x: x[2]))
        current_state, path, _ = queue.popleft()

        if current_state == GOAL_STATE:
            return path, visited_count  # Return path and count

        for neighbor in get_neighbors(current_state):
            if neighbor not in visited:
                heuristic_cost = misplaced_tiles(neighbor)
```

```
                queue.append((neighbor, path + [neighbor], heuristic_cost))
                visited.add(neighbor)
                visited_count += 1  # Increment visited count

    return None, visited_count  # Return count if no solution found

def input_start_state():
    while True:
        print("Enter the starting state as 9 numbers (0 for the empty space):")
        input_state = input("Format: 1 2 3 4 5 6 7 8 0\n")
        try:
            numbers = list(map(int, input_state.split()))
            if len(numbers) != 9 or set(numbers) != set(range(9)):
                raise ValueError
            return tuple(numbers)
        except ValueError:
            print("Invalid input. Please enter numbers from 0 to 8 with no duplicates.")

def print_matrix(state):
    for i in range(0, 9, 3):
        print(state[i:i + 3])

if __name__ == "__main__":
    initial_state = input_start_state()
    print("Initial state:")
    print_matrix(initial_state)
    print()

    solution, visited_count = bfs_with_heuristic(initial_state)
    print(f"Number of states visited: {visited_count}")
    if solution:
        print("\nSolution found with the following steps:")
        for step in solution:
            print_matrix(step)
            print()
    else:
        print("\nNo solution found.")
```

OUTPUT:

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

Number of states visited: 9

Solution found with the following steps:
(1, 2, 3)
(4, 0, 6)
(7, 5, 8)

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

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

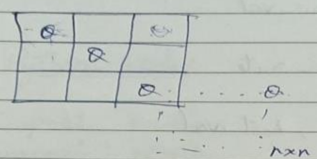
Program 4
Implement Hill Climbing search algorithm to solve N-Queens problem

Algorithm:

LAB 6 29/10/24

A* algorithm.

(i) Hill climbing



Step 1: Initialize:
1/ Generate random initial state.
→ index indicates the row no.
→ column value indicates the column no.

state = {0, 4, 7, 5, 2, 6, 1, 3}

step2. Calculate no. of pairs of attacking queens and assign it to cost.

```
for i in range 8.
    for j in range 8.
        if (state [i] = = state [j])
            cost ++
        if (abs(state (i) - state[j]) = = (j-i))
            cost ++.
    end for
end for
```

Step 3: If the curr_cost is less than the new_cost update the cost

step 4: check the cost for all the queens. if the cost is not minimum then the first states cost return failure

steps: Return solution.

CODE:

```
import random

def calculate_conflicts(board):
    """Calculate the number of conflicts in the current board state."""
    conflicts = 0
    n = len(board)
    for i in range(n):
        for j in range(i + 1, n):
            # Check if queens are in the same column or on the same diagonal
            if board[i] == board[j] or abs(board[i] - board[j]) == abs(i - j):
                conflicts += 1
    return conflicts

def hill_climbing(n):
    """Solve the N-Queens problem using the hill climbing algorithm."""
    cost = 0  # Tracks the number of steps taken
    while True:
        # Initialize a random board
        current_board = list(range(n))
        random.shuffle(current_board)
        current_conflicts = calculate_conflicts(current_board)

        while True:
            # Generate neighbors by moving each queen to a different position
            found_better = False
            for i in range(n):
                for j in range(n):
                    if j != current_board[i]:  # Only consider different positions
                        neighbor_board = list(current_board)
                        neighbor_board[i] = j  # Move queen to a new row
                        neighbor_conflicts = calculate_conflicts(neighbor_board)

                        if neighbor_conflicts < current_conflicts:
                            # Update to the better neighbor
                            current_board = neighbor_board
                            current_conflicts = neighbor_conflicts
                            cost += 1
                            found_better = True
                            break
                if found_better:
                    break

            # If no better neighbor found, stop searching
            if not found_better:
                break

        # If a solution is found (zero conflicts), return the board
        if current_conflicts == 0:
            return current_board, cost

def print_board(board):
```

```
    """Print the board in a human-readable format."""
    n = len(board)
    for i in range(n):
        row = ['.'] * n
        row[board[i]] = 'Q'  # Place a queen
        print(' '.join(row))
    print()

if __name__ == "__main__":
    n = int(input("Enter the value of N (size of the board): "))
    solution, cost = hill_climbing(n)
    print("\nSolution found:")
    print_board(solution)
    print(f"Number of steps taken: {cost}")
```

OUTPUT:

```
Enter the value of N (size of the board): 8


Solution found:
. . Q . . . . .
. . . . Q . . .
. Q . . . . . .
. . . . . . . Q
. . . . . Q . .
. . . Q . . . .
. . . . . . Q .
Q . . . . . . .


Number of steps taken: 9
```

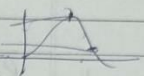
Program 5

Simulated Annealing to Solve 8-Queens problem

Algorithm:

LAB 5:

Simulated annealing



Step 1: Set $T = T\_initial$

step 2: set x_best = x_current

// Initialize the best solution

step 3: while $T > T_{min}$

// Initialize T_{min} in begining.

step 4: For i = 1 to max_iterations

// Initialize max_iterations.

step 5: Generate a new solution x_new by slightly modifying x_current

Step 6: Calculate $\Delta E = f(x\_new) - f(x\_current)$

step 7: If $\Delta E < 0$ (x_new is better).

Step 8. Accept x_new : x_current = x_new.

else :

Accept x_new with probability

$P = \exp(-\Delta E / T)$

- if accepted, set x_curr = x_new.

9. If f(x_curr) < f(x_best)

update x_best = current.

CODE:

```
import random
import math

# Function to calculate the cost of the current state (number of conflicting pairs)
def calculate_cost(board):
    n = len(board)
    cost = 0
    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):
                cost += 1
    return cost

# Function to generate all valid neighbors of the current state
def get_neighbors(board):
    neighbors = []
    n = len(board)
    for row in range(n):
        for col in range(n):
            if col != board[row]:  # Ensure not moving to the current column
                new_board = board[:]
                new_board[row] = col  # Move the queen to the new column
                neighbors.append(new_board)
    return neighbors

# Function to simulate annealing process for solving N-Queens problem
def simulated_annealing(n, initial_temperature=1000, cooling_rate=0.99, max_iterations=10000):
    # Initialize a random state
    board = [random.randint(0, n - 1) for _ in range(n)]  # Random initial state
    cost = calculate_cost(board)
    temperature = initial_temperature

    print("Initial Board:")
    print_board(board)

    iteration = 0
    while temperature > 1 and iteration < max_iterations:  # While temperature is above threshold
        neighbors = get_neighbors(board)
        best_cost = cost
        best_neighbor = None

        # Evaluate all neighbors and choose the one with the least cost
        for neighbor in neighbors:
            neighbor_cost = calculate_cost(neighbor)
            if neighbor_cost < best_cost:
                best_cost = neighbor_cost
                best_neighbor = neighbor
```

```
        # If no better neighbor, accept a worse one based on temperature
        if best_cost > cost:
            probability = math.exp((cost - best_cost) / temperature)
            if random.random() < probability:
                board = best_neighbor
                cost = best_cost

        # Update the board with the best neighbor
        if best_neighbor is not None and best_cost < cost:
            board = best_neighbor
            cost = best_cost

        # Decrease temperature according to the cooling schedule
        temperature *= cooling_rate
        iteration += 1
        print(f"Iteration {iteration}: Cost = {cost}, Temperature = {temperature}")
        print_board(board)

        # If a solution is found, terminate
        if cost == 0:
            print("Solution Found!")
            print_board(board)
            return board

    print("Solution not found within the iteration limit.")
    return None

# Function to print the board
def print_board(board):
    n = len(board)
    for row in range(n):
        line = ['Q' if col == board[row] else '.' for col in range(n)]
        print(' '.join(line))
    print()

# Example usage:
n = 8  # N for N-Queens
solution = simulated_annealing(n)
if solution:
    print("Solution found:")
    print_board(solution)
else:
    print("No solution found.")
```

OUTPUT:

```
. . . . . . . Q
. . . Q . . . .
. . . . . . Q .
. . Q . . . . .

Iteration 20: Cost = 1, Temperature = 817.9069375972307
. . . . . Q . .
Q . . . . . . .
. . . . Q . . .
Q . . . . . . .
. . . . . . . Q
. . . Q . . . .
. . . . . . Q .
. . Q . . . . .

Iteration 21: Cost = 1, Temperature = 809.7278682212584
. . . . . Q . .
Q . . . . . . .
. . . . Q . . .
Q . . . . . . .
. . . . . . . Q
. . . Q . . . .
. . . . . . Q .
. . Q . . . . .
```

Program 6

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

Algorithm:

LAB-7 12/11/24

Bafna Gold Date: Page:

Knowledge Base:

Alice is the mother of Bob.
Bob is the father of charlie.
A father is a parent.
A mother is a parent.
All parents have children.
If someone is a parent, their children are siblings.
Alice is married to David.

Hypothesis:

Charlie is a sibling of Bob.

\# step - by - step solution:

1. Entailment Process:

- Alice is the mother of Bob and Bob is the father of Charlie.
- ~~It implies Alice and Bob are parents~~
- Alice is the mother of Bob implies alice is a parent.
- Bob is a father of charlie ~~implies~~ and a father is a parent → Bob is a parent.
- ~~All~~ Alice and bob are parents and all parents have children implies alice and bob have children.
- Alice and bob are parents and if someone is a parent, their children are siblings.

CODE:

```
import itertools

# Define symbols in the KB and query
symbols = ['A', 'B', 'C']

# Define the Knowledge Base (KB) as separate components
A_or_C = lambda A, B, C: A or C
B_or_not_C = lambda A, B, C: B or not C

# Combine the components to define KB
KB = lambda A, B, C: A_or_C(A, B, C) and B_or_not_C(A, B, C)

# Define the Query (alpha)
query = lambda A, B, C: A or B

# Function to print the truth tables
def print_truth_tables(symbols, A_or_C, B_or_not_C, KB, query):
    # Full truth table
    print(f"{'A':<6}{'B':<6}{'C':<6}{'A∨C':<8}{'B∨¬C':<8}{'KB':<8}{'α (A∨B)':<8}")
    print("-" * 56)

    # List to store combinations where both KB and α are true
    both_true = []

    # Generate all possible truth assignments for symbols
    for values in itertools.product([False, True], repeat=len(symbols)):
        # Create a dictionary for the current truth assignment
        assignment = dict(zip(symbols, values))

        # Evaluate each part of the table based on the current assignment
        A_val = assignment['A']
        B_val = assignment['B']
        C_val = assignment['C']
        A_or_C_val = A_or_C(A_val, B_val, C_val)
        B_or_not_C_val = B_or_not_C(A_val, B_val, C_val)
        KB_val = KB(A_val, B_val, C_val)
        query_val = query(A_val, B_val, C_val)

        # Print each row of the truth table
        print(f"{str(A_val):<6}{str(B_val):<6}{str(C_val):<6}"
              f"{str(A_or_C_val):<8}{str(B_or_not_C_val):<8}"
              f"{str(KB_val):<8}{str(query_val):<8}")

        # Store combinations where both KB and α are true
        if KB_val and query_val:
            both_true.append(assignment)
```

```python
    # Table for combinations where both KB and α are true
    print("\nCombinations where both KB and α (A∨B) are true:")
    print(f"{'A':<6}{'B':<6}{'C':<6}")
    print("-" * 18)
    for assignment in both_true:
        print(f"{assignment['A']:<6}{assignment['B']:<6}{assignment['C']:<6}")

# Run the function to print the truth tables
print_truth_tables(symbols, A_or_C, B_or_not_C, KB, query)
```

OUTPUT:

```
A     B     C     AvC     Bv¬C    KB      α (AvB)
--------------------------------------------------------
False False False False   True    False   False
False False True  True    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  True  False True    True    True    True
True  True  True  True    True    True    True

Combinations where both KB and α (AvB) are true:
A     B     C
------------------
0     1     1
1     0     0
1     1     0
1     1     1
```

Program 7

Implement unification in first order logic

Algorithm:

18/11/24

LAB - 7

Unification :

ψ_1 : Knows (x , Person (y , z))

ψ_2 : Knows (Sara , Person(Amith , Akash))

step 1 : ψ_1 and ψ_2 both sentences use same predicates

step 2 : $\psi_1 \rightarrow x$

$\psi_2 \rightarrow$ Sara

substitute (Sara / x)

step 3 : Second argument

Both sentences have same functor Person.

> Inside Person :

1st arguement ,

y = Amith

2nd arguement

z = Akash.

substitute (Amith / y)

substitute (Akash / z)

step4 : Unified form is Knows (Sara , Person (Amith , Akash)) ; and the unifier is { x = Sara , y = Amith , z = Akash }

19/11/24

CODE:

```
def unify(expr1, expr2):
    print(f"Unifying {expr1} with {expr2}")
    if expr1 == expr2:
        print("Result: Identical terms, no substitution needed.")
        return []  # Return NIL if expressions are identical
    elif is_variable(expr1):
        return failure_if_occurs_check(expr1, expr2)
    elif is_variable(expr2):
        return failure_if_occurs_check(expr2, expr1)
    elif is_compound(expr1) and is_compound(expr2):
        if get_predicate(expr1) != get_predicate(expr2):
            print("Failure: Predicates do not match.")
            return "FAILURE"
        return unify_args(get_arguments(expr1), get_arguments(expr2))
    else:
        print("Failure: Incompatible terms.")
        return "FAILURE"

def unify_args(args1, args2):
    if len(args1) != len(args2):
        print("Failure: Arguments have different lengths.")
        return "FAILURE"
    subst = []
    for a1, a2 in zip(args1, args2):
        s = unify(a1, a2)
        if s == "FAILURE":
            print(f"Failure: Could not unify {a1} with {a2}.")
            return "FAILURE"
        if s:
            subst.extend(s)
            args1 = apply_substitution(s, args1)
            args2 = apply_substitution(s, args2)
    return subst

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

def is_compound(expression):
    return isinstance(expression, str) and "(" in expression and ")" in expression

def get_predicate(expression):
    return expression.split("(")[0]

def get_arguments(expression):
    args_str = expression[expression.index("(") + 1 : expression.rindex(")")]
    return [arg.strip() for arg in args_str.split(",")]
```

```
def failure_if_occurs_check(variable, expression):
    if occurs_check(variable, expression):
        print(f"Failure: Occurs check failed for {variable} in {expression}.")
        return "FAILURE"
    print(f"Substitution: {variable} -> {expression}")
    return [(variable, expression)]

def occurs_check(variable, expression):
    if variable == expression:
        return True
    if is_compound(expression):
        return variable in get_arguments(expression)
    return False

def apply_substitution(subst, expression):
    if isinstance(expression, list):
        return [apply_substitution(subst, sub_expr) for sub_expr in expression]
    elif is_variable(expression):
        for var, value in subst:
            if expression == var:
                return value
    elif is_compound(expression):
        predicate = get_predicate(expression)
        arguments = get_arguments(expression)
        substituted_args = [apply_substitution(subst, arg) for arg in arguments]
        return f"{predicate}({', '.join(substituted_args)})"
    return expression

# Example usage:
expr1 = "P(b,X,f(g(Z)))"
expr2 = "P(Z,f(y),f(y))"

result = unify(expr1, expr2)

print("\nFinal Result:")
if result == "FAILURE":
    print("Unification failed!")
else:
    print("Unification successful!")
    print("Substitutions:", ', '.join(f"{var} -> {val}" for var, val in result))
```

OUTPUT:

```
Unifying P(b,X,f(g(Z))) with P(Z,f(y),f(y))
Unifying b with Z
Substitution: b -> Z
Unifying X with f(y)
Substitution: f(y) -> X
Unifying f(g(Z)) with f(y)
Substitution: f(y) -> f(g(Z))

Final Result:
Unification successful!
Substitutions: b -> Z, f(y) -> X, f(y) -> f(g(Z))

=== Code Execution Successful ===
```

Program 8

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

Algorithm:

LAB 8 3/12/24

First order logic (Forward chaining)

Q. It is prime crime for an American to sell weapons to hostile nations. Country A, an enemy of America, has some missiles, and all the missiles were sold to it by Robert, who is an American citizen.

To prove "Robert is criminal".

Representation:

p, q, r → variables

- It is a crime for an American to sell weapons to hostile nations

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

- Country A has some missiles

∃ x Owns (A, x) ∧ Missile (x)

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

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

- Robert is an American, American (Robert)

:

CODE:

```
class ForwardReasoning:
    def __init__(self, rules, facts):
        """
        Initializes the ForwardReasoning system.
        Parameters:
        rules (list): List of rules as tuples (condition, result),
        where 'condition' is a set of facts.
        facts (set): Set of initial known facts.
        """
        self.rules = rules  # List of rules (condition -> result)
        self.facts = set(facts)  # Known facts

    def infer(self, query):
        """
        Applies forward reasoning to infer new facts based on rules and initial facts.
        Parameters:
        query (str): The fact to verify if it can be inferred.
        Returns:
        bool: True if the query can be inferred, False otherwise.
        """
        applied_rules = True
        while applied_rules:
            applied_rules = False
            for condition, result in self.rules:
                if condition.issubset(self.facts) and result not in self.facts:
                    self.facts.add(result)
                    applied_rules = True
                    print(f"Applied rule: {condition} -> {result}")

        # After applying all rules, check if the query is in the facts
        if query in self.facts:
            return True
        return False


# Define the Knowledge Base (KB) with rules as (condition, result)
rules = [
    ({"American(Robert)"}, "Sells(Robert, m1, CountryA)"),  # Based on Owns(CountryA, m) ^
Missile(m)
    ({"Sells(Robert, m1, CountryA)", "American(Robert)", "Hostile(CountryA)"}, "Criminal(Robert)"),
# Final inference
]

# Define initial facts
facts = {
    "American(Robert)",
    "Hostile(CountryA)",
```

```
    "Missile(m1)",
    "Owns(CountryA, m1)",
}

# Query
alpha = "Criminal(Robert)"

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

print("\nFinal facts:")
print(reasoner.facts)
print(f"\nQuery '{alpha}' inferred: {result}")
```

OUTPUT:

```
Applied rule: {'American(Robert)'} -> Sells(Robert, m1, CountryA)
Applied rule: {'American(Robert)', 'Sells(Robert, m1, CountryA)', 'Hostile(CountryA)'}
    -> Criminal(Robert)

Final facts:
{'American(Robert)', 'Owns(CountryA, m1)', 'Hostile(CountryA)', 'Criminal(Robert)',
    'Sells(Robert, m1, CountryA)', 'Missile(m1)'}

Query 'Criminal(Robert)' inferred: True

=== Code Execution Successful ===
```

Program 9

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

Algorithm:

Python code:

```
class KnowledgeBase:
    def_init_(self):
    self.rules = []
    self.facts = set()

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

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

    def infer (self):
        new_inferences = True
        while new_inferences:
            new_inferences = False
                for premise, conclusion in self.rules:
                    if all (fact in self.facts for fact
                        in premise):
```

Bafna Gold
Date: Page:

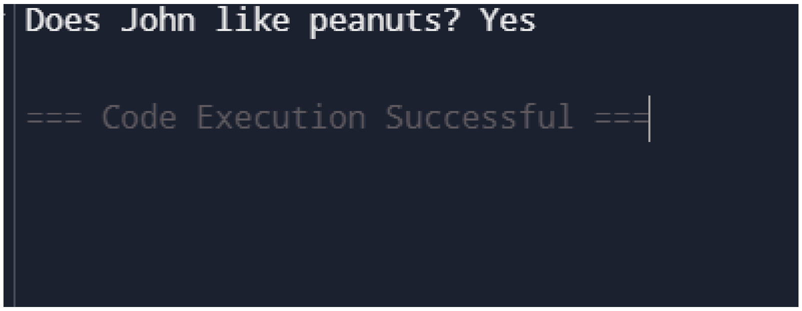
```
    if conclusion is not in self.facts:
        self.facts.add(conclusion)
        new_inferences = True

    def entails(self, hypothesis):
        return hypothesis in self.facts.

Kb = KnowledgeBase()

Kb.add_fact("Alice is the mother of Bob")
Kb.add_fact("Bob is the father of charlie")
Kb.add_fact("A father is a parent")
Kb.add_fact("A mother is a parent").
Kb.add_fact("All parents have children")
Kb.add_fact("Alice is married to David")

Kb.add_rule(["Bob is the father of charlie",
 "A father is a parent"], "Bob is a parent")
Kb.add_rule(["Alice is the mother of Bob",
 "A mother is a parent"], "Alice is a parent")
Kb.add_rule(["Bob is a parent", "All parents
have children"], "Charlie and Bob are siblings")

Kb.infer()

hypothesis = "charlie and Bob are siblings"
if Kb.entails(hypothesis):
    print(f'The hypothesis is entailed by the
          Knowledge base").
else:
    print(f" the hypothesis '{hypothesis}' is not
entailed by the Kb").
```

CODE:

```
# Define the knowledge base (KB)
KB = {
    "food(Apple)": True,
    "food(vegetables)": True,
    "eats(Anil, Peanuts)": True,
    "alive(Anil)": True,
    "likes(John, X)": "food(X)", # Rule: John likes all food
    "food(X)": "eats(Y, X) and not killed(Y)",  # Rule: Anything eaten and not killed is food
    "eats(Harry, X)": "eats(Anil, X)",  # Rule: Harry eats what Anil eats
    "alive(X)": "not killed(X)", # Rule: Alive implies not killed
    "not killed(X)": "alive(X)", # Rule: Not killed implies alive
}

# Function to evaluate if a predicate is true based on the KB
def resolve(predicate):
    # If it's a direct fact in KB
    if predicate in KB and isinstance(KB[predicate], bool):
        return KB[predicate]

    # If it's a derived rule
    if predicate in KB:
        rule = KB[predicate]
        if " and " in rule: # Handle conjunction
            sub_preds = rule.split(" and ")
            return all(resolve(sub.strip()) for sub in sub_preds)
        elif " or " in rule: # Handle disjunction
            sub_preds = rule.split(" or ")
            return any(resolve(sub.strip()) for sub in sub_preds)
        elif "not " in rule: # Handle negation
            sub_pred = rule[4:] # Remove "not "
            return not resolve(sub_pred.strip())
        else: # Handle single predicate
            return resolve(rule.strip())

    # If the predicate is a specific query (e.g., likes(John, Peanuts))
    if "(" in predicate:
        func, args = predicate.split("(")
        args = args.strip(")").split(", ")
        if func == "food" and args[0] == "Peanuts":
            return resolve("eats(Anil, Peanuts)") and not resolve("killed(Anil)")
        if func == "likes" and args[0] == "John" and args[1] == "Peanuts":
            return resolve("food(Peanuts)")

    # Default to False if no rule or fact applies
    return False
```

```
# Query to prove: John likes Peanuts
query = "likes(John, Peanuts)"
result = resolve(query)

# Print the result
print(f"Does John like peanuts? {'Yes' if result else 'No'}")
```

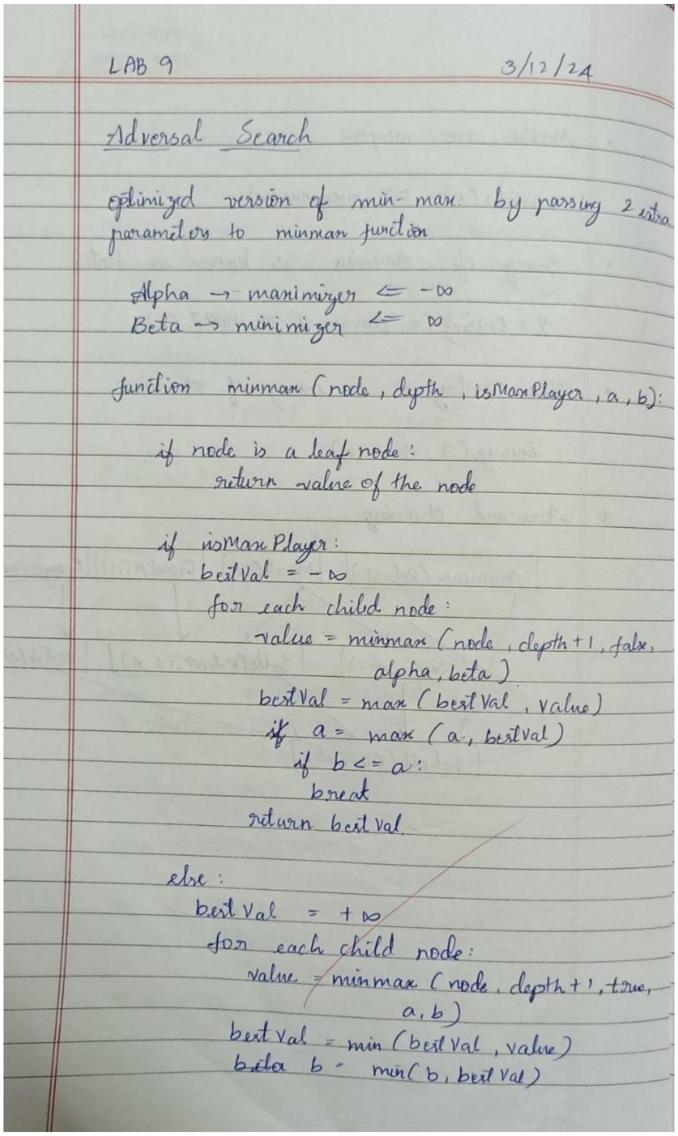
OUTPUT:



Program 10

Implement Alpha-Beta Pruning.

Algorithm:



LAB 9 3/12/24

Adversal Search

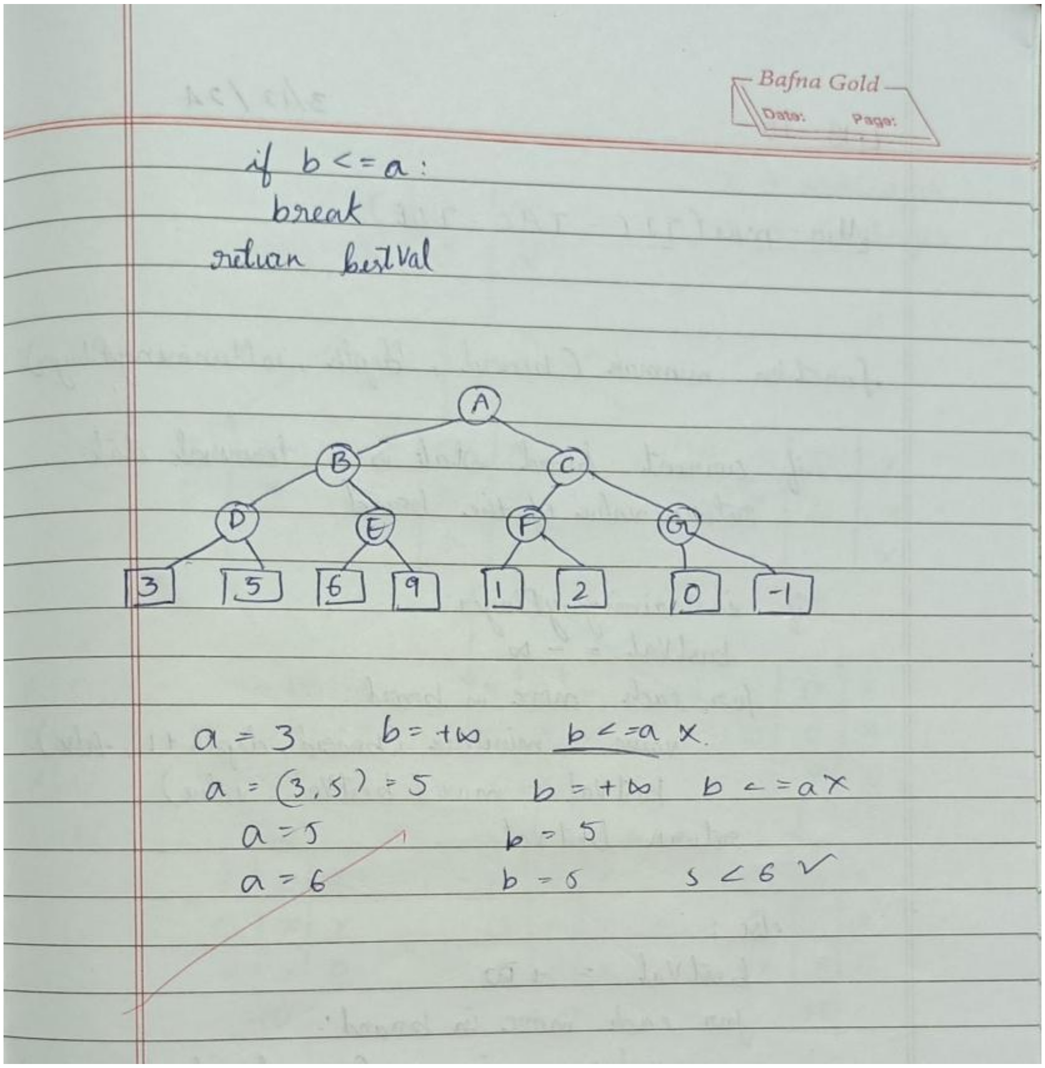
optimized version of min-max by passing 2 extra parameters to minmax function

Alpha → maximizer ⇐ $-\infty$

Beta → minimizer ⇐ ∞

function minmax (node, depth, isMaxPlayer, a, b):

```
    if node is a leaf node:
        return value of the node

    if isMaxPlayer:
        bestVal = -∞
        for each child node:
            value = minmax (node, depth+1, false,
                            alpha, beta)
            bestVal = max (bestVal, value)
            if a = max (a, bestVal)
               if b <= a:
                  break
        return bestVal

    else:
        bestVal = +∞
        for each child node:
            value = minmax (node, depth+1, true,
                            a, b)
            best Val = min (bestVal, value)
            beta b = min(b, bestVal)
```



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


=== Code Execution Successful ===
```