VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB REPORT
on

Artificial Intelligence (23CS5PCAIN)

Submitted by

Prabhanjan Bhat (1BM22CS196)

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

Prof. Swathi Sridharan
Assistant 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

<u>CERTIFICATE</u>

This is to certify that the Lab work entitled "Artificial Intelligence (23CS5PCAIN)" carried out by **Prabhanjan Bhat(1BM22CS196),** 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.

Prof. Sneha S Bagalkot	Dr. Kavitha Sooda
Assistant Professor	Professor & HOD
Department of CSE, BMSCE	Department of CSE, BMSCE

Index

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

Algorithm:

DATE: 24/9/24 PAGE: ①

(I) Tic - Taa - Toc Game Algorithm:

(1) Initialization of Game Board :
Create a 3×3 matrix using 2-D array to represent
the board, initialized to empty states.

(ii) Display the Board to show the current state of the board.

(iii) User Input : allow the user to input their move (row and
column)

(iv) Check for Win: check if a player has won the game.

(v) Check for Draw : check if the board is full and there is no
winner.

(vi) ~~Suitable~~ Algorithm ~~for~~ to be implemented for Computer's
move ⇒
  -). ① Winning Move
  → ② Blocking Move
  → ③ Center Move
  → ④ corner move
  → ⑤ Side Move

(vii) check for win or draw again : Once the computer moves,
check for a win or draw.

(viii) Repeat steps 3-6 alternating between user and computer
untile the game ends.

4

Code:
```python
def check_win(board, r, c):
    if board[r - 1][c - 1] == 'X':
        ch = "O"
    else:
        ch = "X"
    if ch not in board[r - 1] and '-' not in board[r - 1]:
        return True
    elif ch not in (board[0][c - 1], board[1][c - 1], board[2][c - 1]) and '-' not in (board[0][c - 1], board[1][c - 1], board[2][c - 1]):
        return True
    elif ch not in (board[0][0], board[1][1], board[2][2]) and '-' not in (board[0][0], board[1][1], board[2][2]):
        return True
    elif ch not in (board[0][2], board[1][1], board[2][0]) and '-' not in (board[0][2], board[1][1], board[2][0]):
        return True
    return False

def displayb(board):
    print(board[0])
    print(board[1])
    print(board[2])

board=[['-','-','-'],['-','-','-'],['-','-','-']]
displayb(board)
xo=1
flag=0
while '-' in board[0] or '-' in board[1] or '-' in board[2]:

    if xo==1:
        print("enter position to place X:")
        x=int(input())
        y=int(input())
        if(x>3 or y>3):
            print("invalid position")
            continue
        if(board[x-1][y-1]=='-'):
            board[x-1][y-1]='X'
            xo=0
            displayb(board)
        else:
            print("invalid position")
            continue
        if(check_win(board,x,y)):
```

```python
            print("X wins")
            flag=1
            break
    else :
      print("enter position to place O:")
      x=int(input())
      y=int(input())
      if(x>3 or y>3):
        print("invalid position")
        continue
      if(board[x-1][y-1]=='-'):
        board[x-1][y-1]='O'
        xo=1
        displayb(board)
      else:
        print("invalid position")
      continue
      if(check_win(board,x,y)):
          print("0 wins")
          flag=1
          break
if flag==0:
  print("Draw")
print("Game Over")
```

```
['-', '-', '-']
['-', '-', '-']
['-', '-', '-']
enter position to place X:
1
1
['X', '-', '-']
['-', '-', '-']
['-', '-', '-']
enter position to place O:
1
2
['X', 'O', '-']
['-', '-', '-']
['-', '-', '-']
enter position to place X:
2
1
['X', 'O', '-']
['X', '-', '-']
['-', '-', '-']
enter position to place O:
2
2
['X', 'O', '-']
['X', 'O', '-']
['-', '-', '-']
enter position to place X:
3
1
['X', 'O', '-']
['X', 'O', '-']
['X', '-', '-']
X wins
Game Over
```

```
['-', '-', '-']
['-', '-', '-']
['-', '-', '-']
enter position to place X:
1
1
['X', '-', '-']
['-', '-', '-']
['-', '-', '-']
enter position to place O:
2
2
['X', '-', '-']
['-', 'O', '-']
['-', '-', '-']
enter position to place X:
3
3
['X', '-', '-']
['-', 'O', '-']
['-', '-', 'X']
enter position to place O:
1
2
['X', 'O', '-']
['-', 'O', '-']
['-', '-', 'X']
enter position to place X:
3
2
['X', 'O', '-']
['-', 'O', '-']
['-', 'X', 'X']
enter position to place O:
3
1
['X', 'O', '-']
['-', 'O', '-']
['O', 'X', 'X']
```

```
enter position to place X:
2
1
['X', 'O', '-']
['X', 'O', '-']
['O', 'X', 'X']
enter position to place O:
2
3
['X', 'O', '-']
['X', 'O', 'O']
['O', 'X', 'X']
enter position to place X:
1
3
['X', 'O', 'X']
['X', 'O', 'O']
['O', 'X', 'X']
Draw
Game Over
```

Vacuum Cleaner

Algorithm:

Lab-2

Vacuum cleaner Algorithm

(2)

(i) A vacuum cleaner agent performs a suck operation or if the room is dirty and moves to other location and checks the status.

(ii) We consider 2 rooms A and B (a grid 2×1). The agent can percieve whether its current room is clean or dirty.

(iii) Thus the agent performs 3 operations: move left, move right, suck

(iv) Input: Boolean data (0 or 1) whether the current room is clean or dirty. Clean = 1, Dirty = 0

(v) while (environment room != clean)

```
if (current_room = dirty){
    clean();
}
else if (current_room = clean)
{    move_right();
}

if (agent in room A)
{    move_right();
}
else if (agent in room B)
{    move_left();
}
}
```

Code:
```
count = 0
def rec(state, loc):
    global count
```

```python
    if state['A'] == 0 and state['B'] == 0:
        print("Turning vacuum off")
        return

    if state[loc] == 1:
        state[loc] = 0
        count += 1
        print(f"Cleaned {loc}.")
        next_loc = 'B' if loc == 'A' else 'A'
        state[loc] = int(input(f"Is {loc} clean now? (0 if clean, 1 if dirty): "))
        if(state[next_loc]!=1):
            state[next_loc]=int(input(f"Is {next_loc} dirty? (0 if clean, 1 if dirty): "))
    if(state[loc]==1):
 rec(state,loc)else:
        next_loc = 'B' if loc == 'A' else 'A'
        dire="left" if loc=="B" else "right"
        print(loc,"is clean")
        print(f"Moving vacuum {dire}")
        if state[next_loc] == 1:
            rec(state, next_loc)

    state = {}
    state['A'] = int(input("Enter state of A (0 for clean, 1 for dirty): "))
    state['B'] = int(input("Enter state of B (0 for clean, 1 for dirty): "))
    loc = input("Enter location (A or B): ")
    rec(state, loc)
    print("Cost:",count)
    print(state)
```

```
Enter state of A (0 for clean, 1 for dirty): 0
Enter state of B (0 for clean, 1 for dirty): 0
Enter location (A or B): A
Turning vacuum off
Cost: 0
{'A': 0, 'B': 0}
```

```
Enter state of A (0 for clean, 1 for dirty): 0
Enter state of B (0 for clean, 1 for dirty): 1
Enter location (A or B): A
A is clean
Moving vacuum right
Cleaned B.
Is B clean now? (0 if clean, 1 if dirty): 0
Is A dirty? (0 if clean, 1 if dirty): 0
B is clean
Moving vacuum left
Cost: 1
{'A': 0, 'B': 0}
```

```
Enter state of A (0 for clean, 1 for dirty): 1
Enter state of B (0 for clean, 1 for dirty): 0
Enter location (A or B): A
Cleaned A.
Is A clean now? (0 if clean, 1 if dirty): 0
Is B dirty? (0 if clean, 1 if dirty): 0
A is clean
Moving vacuum right
Cost: 1
{'A': 0, 'B': 0}
```

```
Enter state of A (0 for clean, 1 for dirty): 1
Enter state of B (0 for clean, 1 for dirty): 1
Enter location (A or B): A
Cleaned A.
Is A clean now? (0 if clean, 1 if dirty): 0
A is clean
Moving vacuum right
Cleaned B.
Is B clean now? (0 if clean, 1 if dirty): 0
Is A dirty? (0 if clean, 1 if dirty): 0
B is clean
Moving vacuum left
Cost: 2
{'A': 0, 'B': 0}
```
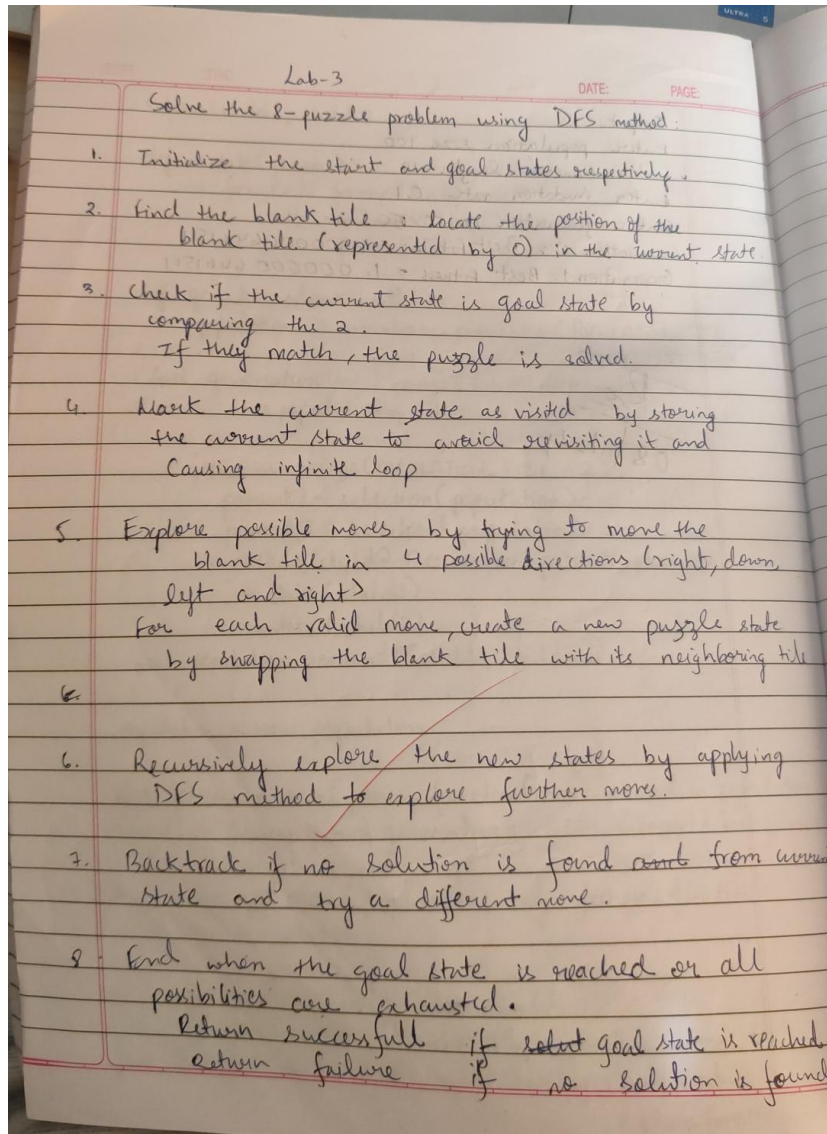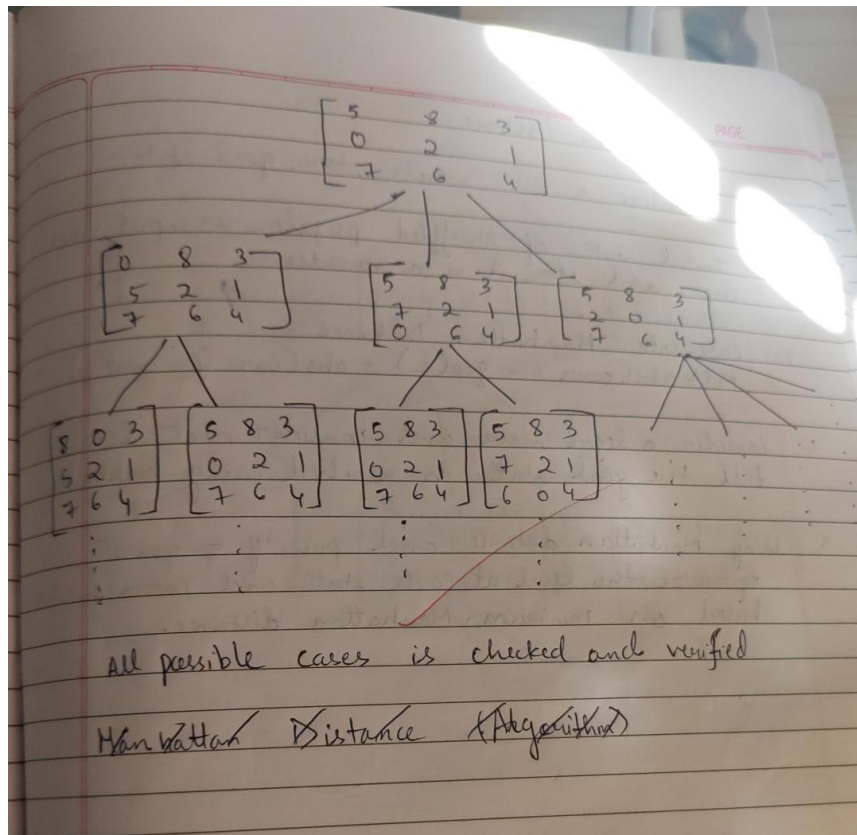
Program 2
Implement 8 puzzle problems using Depth First Search (DFS)
Implement Iterative deepening search algorithm

8 puzzle using DFS
Algorithm:

Lab-3

Solve the 8-puzzle problem using DFS method:

1. Initialize the start and goal states respectively.

2. Find the blank tile : locate the position of the blank tile (represented by 0) in the current state

3. Check if the current state is goal state by comparing the 2.
   If they match, the puzzle is solved.

4. Mark the current state as visited by storing the current state to avoid revisiting it and causing infinite loop

5. Explore possible moves by trying to move the blank tile in 4 possible directions (right, down, left and right)
   For each valid move, create a new puzzle state by swapping the blank tile with its neighboring tile

6. Recursively explore the new states by applying DFS method to explore further moves.

7. Backtrack if no solution is found from current state and try a different move.

8. End when the goal state is reached or all possibilities are exhausted.
   Return successfull if solot goal state is reached
   return failure if no solution is found

All possible cases is checked and verified

Manhattan Distance (Algorithm)

Code:
```
def dfs(initial_board, zero_pos):
    stack = [(initial_board, zero_pos, [])]
    visited = set()

    while stack:
        current_board, zero_pos, moves = stack.pop()

        if is_goal(current_board):
            return moves, len(moves) # Return moves and their count

        visited.add(tuple(current_board))

        for neighbor_board, neighbor_pos in get_neighbors(current_board, zero_pos):
            if tuple(neighbor_board) not in visited:
                stack.append((neighbor_board, neighbor_pos, moves + [neighbor_board]))
```

```
    return None, 0 # No solution found, return count as 0

# Initial state of the puzzle
initial_board = [1, 2, 3, 0, 4, 6, 7, 5, 8]
zero_position = (1, 0) # Position of the empty tile (0)

# Solve the puzzle using DFS
solution, move_count = dfs(initial_board, zero_position)

if solution:
    print("Solution found with moves ({} moves):".format(move_count))
    for move in solution:
        print_board(move)
        print() # Print an empty line between moves
else:
    print("No solution found.")
```

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

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

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

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

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

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

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

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

[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]
```
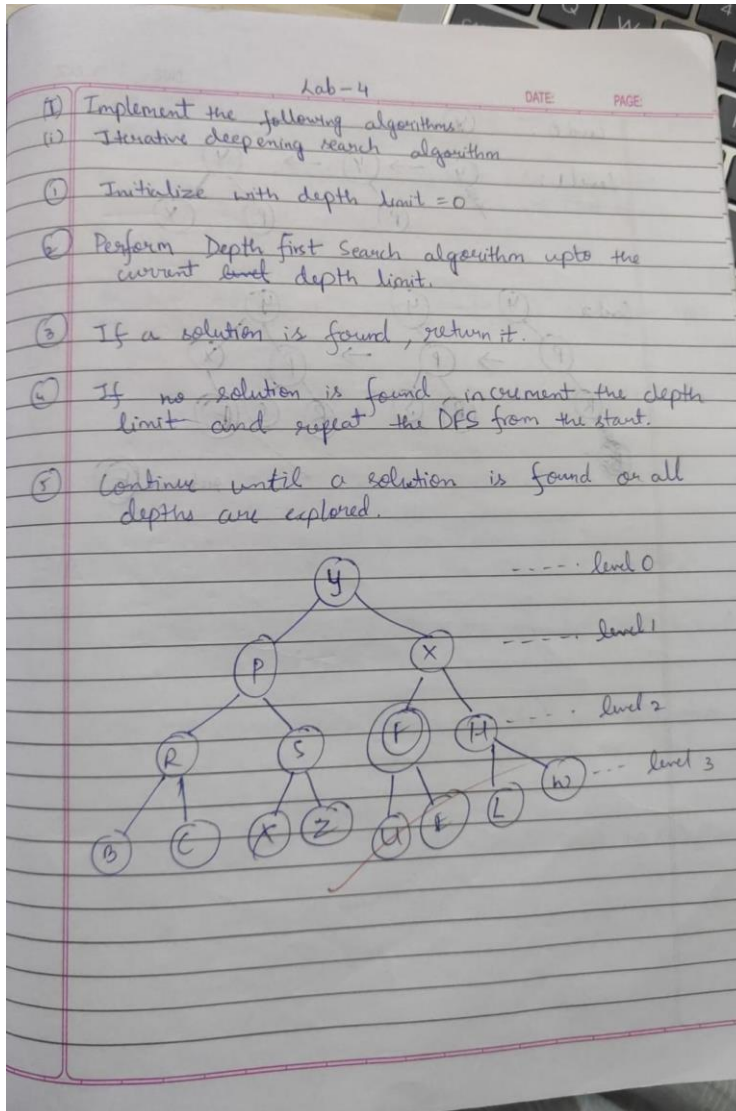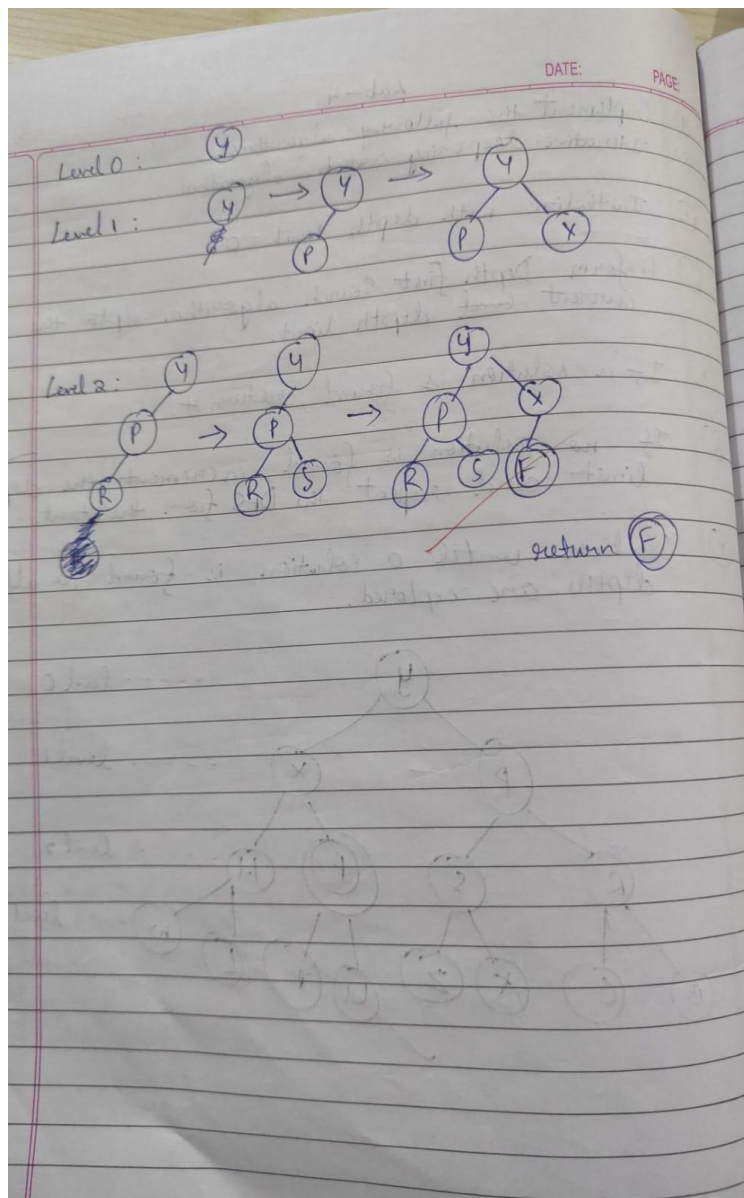
Implement Iterative deepening search algorithm
Algorithm:

Code:
from collections import deque

```
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:
                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():
            new_state = PuzzleState(new_board, new_zero_pos, state.moves + 1, state)
            if tuple(map(tuple, new_board)) not in visited:
                result = dls(new_state, goal_state, depth - 1, visited)
                if result:
                    return result
        visited.remove(tuple(map(tuple, state.board))) # Unmark this state
        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)
```

```python
        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("
```

```
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]
```
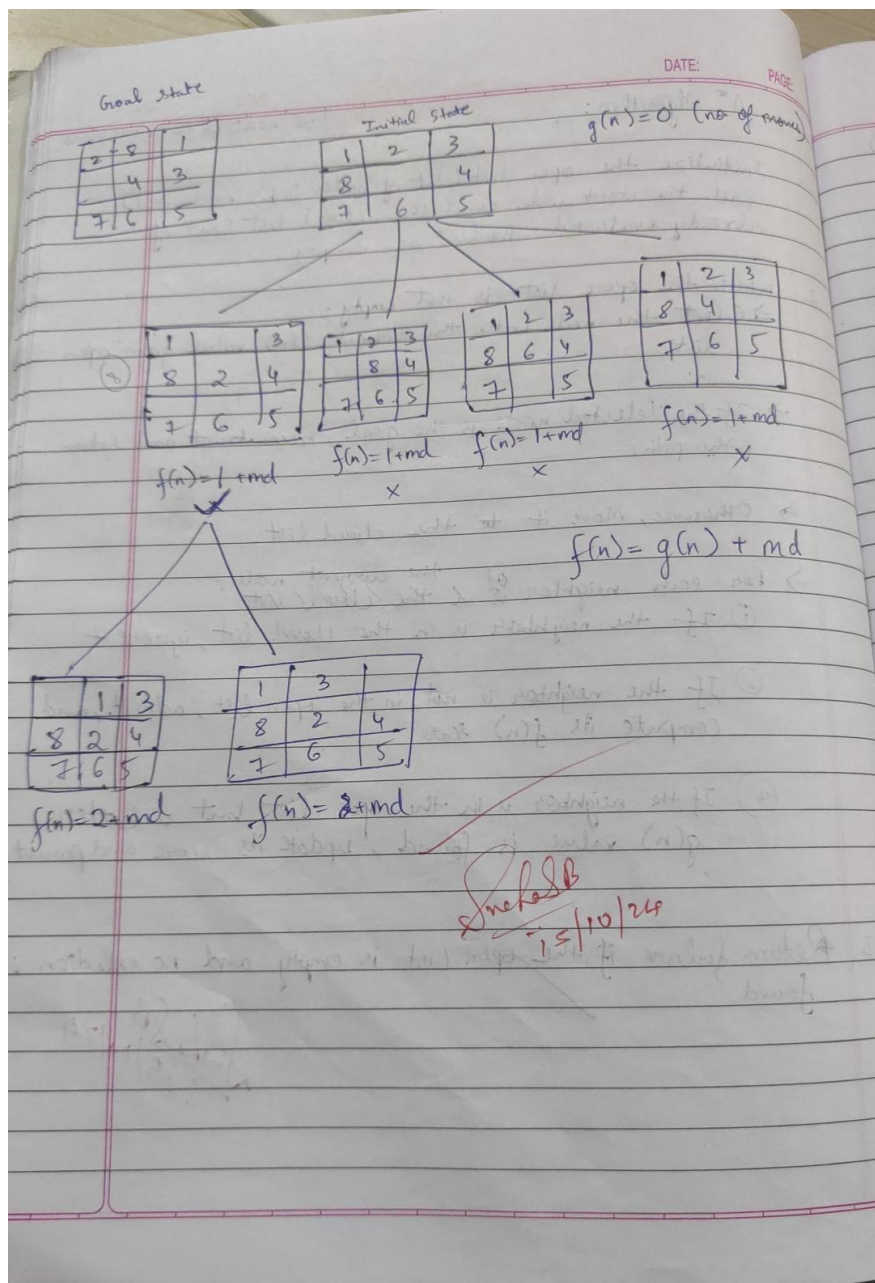
Program 3

Implement A* search algorithm
Algorithm:

1. Initialize the open list (set of nodes to be evaluated) with the start node and the closed list (set of already evaluated nodes) as empty.

2. while the open list is not empty:
   ⇒ Select the node with the lowest f(n) value from open list.

   ⇒ If the selected node is the goal, reconstruct and return the path.

   ⇒ Otherwise, move it to the closed list.

   ⇒ for each neighbor of the current node:
   ① If the neighbor is in the closed list, ignore it

   ② If the neighbor is not in the open list, add it, and compute its f(n) score.

   ③ If the neighbor is in the open list but a better g(n) value is found, update its score and parent

3. Return failure if the open list is empty and no solution found.

15/10/24

Goal state

2	8	1
	4	3
7	6	5

Initial State

1	2	3
8		4
7	6	5

$g(n) = 0$ (no of moves)

1		3
8	2	4
7	6	5

(8)

$f(n) = 1 + md$ ✗

1	2	3
	8	4
7	6	5

$f(n) = 1 + md$ ✗

1	2	3
8	6	4
7		5

$f(n) = 1 + md$ ✗

1	2	3
8	4	
7	6	5

$f(n) = 1 + md$ ✗

$f(n) = 1 + md$ ✓

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

1		3
8	2	4
7	6	5

$f(n) = 2 = md$

1		3
8	2	4
7	6	5

$f(n) = 2 + md$

Snehal B
15/10/24

Code:
Misplaced Tiles

```
def mistil(state, goal):
    count = 0
    for i in range(3):
        for j in range(3):
            if state[i][j] != goal[i][j]:
                count += 1
    return count


def findmin(open_list, goal):
    minv = float('inf')
    best_state = None
    for state in open_list:
        h = mistil(state['state'], goal)
        f = state['g'] + h
        if f < minv:
            minv = f
            best_state = state
    open_list.remove(best_state)
    return best_state


def operation(state):
    next_states = []
    blank_pos = find_blank_position(state['state'])
    for move in ['up', 'down', 'left', 'right']:
        new_state = apply_move(state['state'], blank_pos, move)
        if new_state:
            next_states.append({
                'state': new_state,
                'parent': state,
                'move': move,
                'g': state['g'] + 1
            })
    return next_states


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


def apply_move(state, blank_pos, move):
    i, j = blank_pos
    new_state = [row[:] for row in state]
    if move == 'up' and i > 0:
```

```python
            new_state[i][j], new_state[i - 1][j] = new_state[i - 1][j], new_state[i][j]
        elif move == 'down' and i < 2:
            new_state[i][j], new_state[i + 1][j] = new_state[i + 1][j], new_state[i][j]
        elif move == 'left' and j > 0:
            new_state[i][j], new_state[i][j - 1] = new_state[i][j - 1], new_state[i][j]
        elif move == 'right' and j < 2:
            new_state[i][j], new_state[i][j + 1] = new_state[i][j + 1], new_state[i][j]
        else:
            return None
        return new_state

def print_state(state):
    for row in state:
        print(' '.join(map(str, row)))

initial_state = [[2,8,3], [1,6,4], [7,0,5]]
goal_state = [[1,2,3], [8,0,4], [7,6,5]]
open_list = [{'state': initial_state, 'parent': None, 'move': None, 'g': 0}]
visited_states = []

while open_list:
    best_state = findmin(open_list, goal_state)
    print("Current state:")
    print_state(best_state['state'])
    h = mistil(best_state['state'], goal_state)
    f = best_state['g'] + h
    print(f"g(n): {best_state['g']}, h(n): {h}, f(n): {f}")
    if best_state['move'] is not None:
        print(f"Move: {best_state['move']}")
    print()
    if mistil(best_state['state'], goal_state) == 0:
        goal_state_reached = best_state
        break
    visited_states.append(best_state['state'])
    next_states = operation(best_state)
    for state in next_states:
        if state['state'] not in visited_states:
            open_list.append(state)

moves = []
while goal_state_reached['move'] is not None:
    moves.append(goal_state_reached['move'])
    goal_state_reached = goal_state_reached['parent']
moves.reverse()

print("\nMoves to reach the goal state:", moves)
print("\nGoal state reached:")
```

print_state(goal_state)

```
Current state:
2 8 3
1 6 4
7 0 5
g(n): 0, h(n): 5, f(n): 5

Current state:
2 8 3
1 0 4
7 6 5
g(n): 1, h(n): 3, f(n): 4
Move: up

Current state:
2 0 3
1 8 4
7 6 5
g(n): 2, h(n): 4, f(n): 6
Move: up

Current state:
2 8 3
0 1 4
7 6 5
g(n): 2, h(n): 4, f(n): 6
Move: left

Current state:
0 2 3
1 8 4
7 6 5
g(n): 3, h(n): 3, f(n): 6
Move: left

Current state:
1 2 3
0 8 4
7 6 5
g(n): 4, h(n): 2, f(n): 6
Move: down
```

```
Current state:
1 2 3
8 0 4
7 6 5
g(n): 5, h(n): 0, f(n): 5
Move: right


Moves to reach the goal state: ['up', 'up', 'left', 'down', 'right']

Goal state reached:
1 2 3
8 0 4
7 6 5
```

Manhattan Distance

```
def manhattan_distance(state, goal):
    distance = 0
    for i in range(3):
        for j in range(3):
            tile = state[i][j]
            if tile != 0: # Ignore the blank space (0)
                # Find the position of the tile in the goal state
                for r in range(3):
                    for c in range(3):
                        if goal[r][c] == tile:
                            target_row, target_col = r, c
                            break
                # Add the Manhattan distance (absolute difference in rows and columns)
                distance += abs(target_row - i) + abs(target_col - j)
    return distance

def findmin(open_list, goal):
    minv = float('inf')
    best_state = None
    for state in open_list:
        h = manhattan_distance(state['state'], goal) # Use Manhattan distance here
        f = state['g'] + h
        if f < minv:
            minv = f
            best_state = state
    open_list.remove(best_state)
    return best_state

def operation(state):
    next_states = []
    blank_pos = find_blank_position(state['state'])
    for move in ['up', 'down', 'left', 'right']:
        new_state = apply_move(state['state'], blank_pos, move)
        if new_state:
            next_states.append({
                'state': new_state,
                'parent': state,
                'move': move,
                'g': state['g'] + 1
            })
    return next_states

def find_blank_position(state):
    for i in range(3):
        for j in range(3):
            if state[i][j] == 0:
```

```python
            return i, j
    return None

def apply_move(state, blank_pos, move):
    i, j = blank_pos
    new_state = [row[:] for row in state]
    if move == 'up' and i > 0:
        new_state[i][j], new_state[i - 1][j] = new_state[i - 1][j], new_state[i][j]
    elif move == 'down' and i < 2:
        new_state[i][j], new_state[i + 1][j] = new_state[i + 1][j], new_state[i][j]
    elif move == 'left' and j > 0:
        new_state[i][j], new_state[i][j - 1] = new_state[i][j - 1], new_state[i][j]
    elif move == 'right' and j < 2:
        new_state[i][j], new_state[i][j + 1] = new_state[i][j + 1], new_state[i][j]
    else:
        return None
    return new_state

def print_state(state):
    for row in state:
        print(' '.join(map(str, row)))

# Initial state and goal state
initial_state = [[2,8,3], [1,6,4], [7,0,5]]
goal_state = [[1,2,3], [8,0,4], [7,6,5]]

# Open list and visited states
open_list = [{'state': initial_state, 'parent': None, 'move': None, 'g': 0}]
visited_states = []

while open_list:
    best_state = findmin(open_list, goal_state)

    print("Current state:")
    print_state(best_state['state'])

    h = manhattan_distance(best_state['state'], goal_state) # Using Manhattan distance here
    f = best_state['g'] + h
    print(f"g(n): {best_state['g']}, h(n): {h}, f(n): {f}")

    if best_state['move'] is not None:
        print(f"Move: {best_state['move']}")
    print()
    if h == 0: # Goal is reached if h == 0
        goal_state_reached = best_state
        break
```

```
        visited_states.append(best_state['state'])
        next_states = operation(best_state)

        for state in next_states:
            if state['state'] not in visited_states:
                open_list.append(state)

# Reconstruct the path of moves
moves = []
while goal_state_reached['move'] is not None:
    moves.append(goal_state_reached['move'])
    goal_state_reached = goal_state_reached['parent']
moves.reverse()

print("\nMoves to reach the goal state:", moves)
print("\nGoal state reached:")
print_state(goal_state)
```

```
Current state:
2 8 3
1 6 4
7 0 5
g(n): 0, h(n): 5, f(n): 5

Current state:
2 8 3
1 0 4
7 6 5
g(n): 1, h(n): 4, f(n): 5
Move: up

Current state:
2 0 3
1 8 4
7 6 5
g(n): 2, h(n): 3, f(n): 5
Move: up

Current state:
0 2 3
1 8 4
7 6 5
g(n): 3, h(n): 2, f(n): 5
Move: left

Current state:
1 2 3
0 8 4
7 6 5
g(n): 4, h(n): 1, f(n): 5
Move: down
```

```
Current state:
1 2 3
8 0 4
7 6 5
g(n): 5, h(n): 0, f(n): 5
Move: right


Moves to reach the goal state: ['up', 'up', 'left', 'down', 'right']

Goal state reached:
1 2 3
8 0 4
7 6 5
```
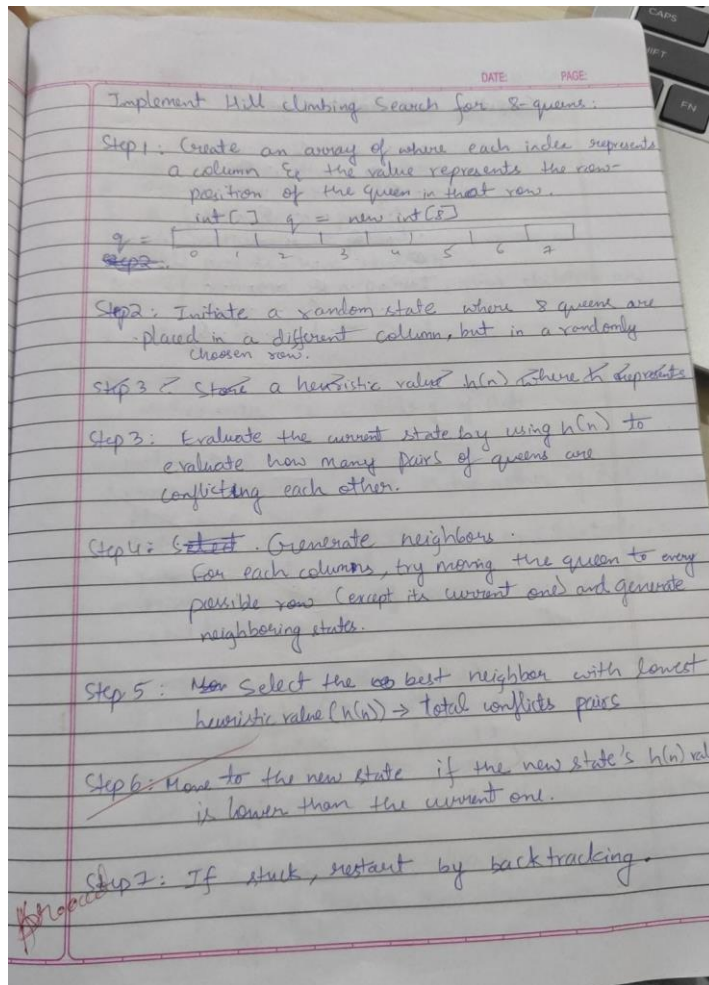
Program 4

Implement Hill Climbing search algorithm to solve N-Queens problem

Algorithm:

Code:
```
import random

def calculate_conflicts(board):
    conflicts = 0
    n = len(board)
    for i in range(n):
        for j in range(i + 1, n):
            if board[i] == board[j] or abs(board[i] - board[j]) == abs(i - j):
                conflicts += 1
```

```python
        return conflicts

def hill_climbing(n):
    cost=0
    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
                        neighbor_conflicts = calculate_conflicts(neighbor_board)
                        if neighbor_conflicts < current_conflicts:
                            print_board(current_board)
                            print(current_conflicts)
                            print_board(neighbor_board)
                            print(neighbor_conflicts)
                            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, current_conflicts, cost

def print_board(board):
    n = len(board)
    for i in range(n):
        row = ['.'] * n
        row[board[i]] = 'Q' # Place a queen
        print(' '.join(row))
    print()
```

```
print("================")
# Example Usage
n = 4
solution, conflicts, cost = hill_climbing(n)
print("Final Board Configuration:")
print_board(solution)
print("Number of Cost:", cost)
```

```
================
Q . . .
. . . Q
. . Q .
. Q . .

4
Q . . .
Q . . .
. . Q .
. Q . .

3
Q . . .
Q . . .
. . Q .
. Q . .

3
. . Q .
Q . . .
. . Q .
. Q . .

2
. . Q .
Q . . .
. . Q .
. Q . .

2
. . . Q
Q . . .
. . Q .
. Q . .

1
Final Board Configuration:
. Q . .
. . . Q
Q . . .
. . Q .
```
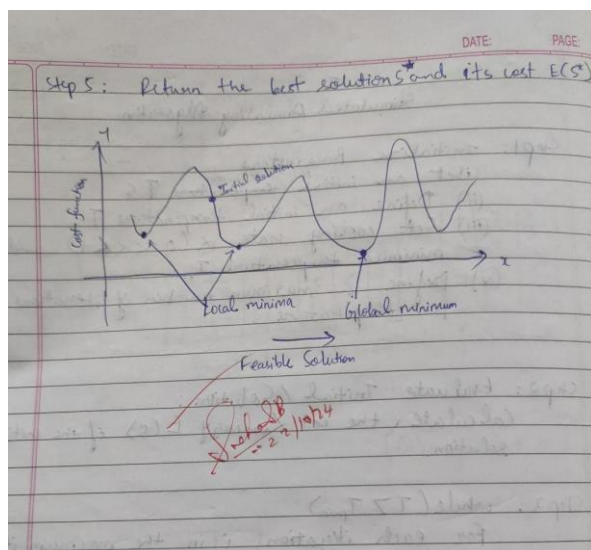
Program 5
Simulated Annealing Algorithm
Algorithm:

Code:

```python
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) # Round and convert to integers for queen positions
    n = len(position)
    queen_not_attacking = 0

    for i in range(n - 1):
        no_attack_on_j = 0
        for j in range(i + 1, n):
            # Check if queens are on the same row or on the same diagonal
            if position[i] != position[j] and abs(position[i] - position[j]) != (j - i):
                no_attack_on_j += 1
        if no_attack_on_j == n - 1 - i:
            queen_not_attacking += 1
    if queen_not_attacking == n - 1:
        queen_not_attacking += 1
    return -queen_not_attacking # Negative because we want to maximize this value

# Bounds for each queen's position (0 to 7 for an 8x8 chessboard)
bounds = [(0, 8) 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 # Flip sign to get the number of non-attacking queens

print('The best position found is:', best_position)
print('The number of queens that are not attacking each other is:', best_objective)
```
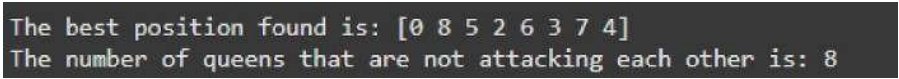
```
The best position found is: [0 8 5 2 6 3 7 4]
The number of queens that are not attacking each other is: 8
```

Program 6

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

Algorithm:

Lab-7
Entailment Process

DATE: 12/11/24 PAGE:

Q. KB:
1. Alice is the mother of Bob
2. Bob is the father of Charlie.
3. A father is a parent
4. A mother is a parent
5. All parents have children
6. If someone is a parent, their children are siblings.
7. Alice is married to David

Hypothesis:
→ "Charlie is a sibling of Bob

Entailment Process:
- From the premise, Alice is the mother of Bob, so Alice is a parent.
- Bob is the father of Charlie, so Charlie's parent is Bob.
- From the premise, If someone is a parent, their child

A → B (Alice is mother of Bob)
B → C (Bob is father of Charlie)
F → P (A father is a parent)
M → P (A mother is a parent)
P → S (If someone is a parent, their children are siblings)
A∧B → Q (If Alice is mother of Bob & Bob is father of Charlie then Charlie is a sibling of Bob)

→ check for entailment :

1. If A(Alice is mother of Bob) is true then
   B (Bob is the father of Charlie) must also be true
   (A → B)

2. If B is true then C (Bob is a parent) must be
   true (F → F) & M (Alice is a parent) must also
   be true (M → P)

3. If both Alice & Charlie are parents (i.e M & F
   are true) then S (their children are sibling)
   must be true. (P → S)

4. Since S is true, the hypothesis Q (" Charlie is
   sibling of Bob") is true.

Conclusion:

Using Propositional logic, we can conclude the
hypothesis "Charlie is a sibling of Bob" is
not entailed by KB (Knowledge Base).

Code:

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

```
import itertools


# Function to evaluate an expression
def evaluate_expression(a, b, c, expression):
    # Use eval() to evaluate the logical expression
    return eval(expression)


# Function to generate the truth table and evaluate a logical expression
def truth_table_and_evaluation(kb, query):
    # All possible combinations of truth values for a, b, and c
    truth_values = [True, False]
    combinations = list(itertools.product(truth_values, repeat=3))

    # Reverse the combinations to start from the bottom (False -> True)
    combinations.reverse()

    # Header for the full truth table
    print(f"{'a':<5}  {'b':<5} {'c':<5} {'KB':<20}{'Query':<20}")

    # Evaluate the expressions for each combination
```

```python
    for combination in combinations:
        a, b, c = combination

        # Evaluate the knowledge base (KB) and query expressions
        kb_result = evaluate_expression(a, b, c, kb)
        query_result = evaluate_expression(a, b, c, query)

        # Replace True/False with string "True"/"False"
        kb_result_str = "True" if kb_result else "False"
        query_result_str = "True" if query_result else "False"

        # Convert boolean values of a, b, c to "True"/"False"
        a_str = "True" if a else "False"
        b_str = "True" if b else "False"
        c_str = "True" if c else "False"

        # Print the results for the knowledge base and the query
        print(f"{a_str:<5} {b_str:<5} {c_str:<5} {kb_result_str:<20} {query_result_str:<20}")

    # Additional output for combinations where both KB and query are true
    print("\nCombinations where both KB and Query are True:")
    print(f"{'a':<5} {'b':<5} {'c':<5} {'KB':<20}{'Query':<20}")

    # Print only the rows where both KB and Query are True
    for combination in combinations:
        a, b, c = combination

        # Evaluate the knowledge base (KB) and query expressions
        kb_result = evaluate_expression(a, b, c, kb)
        query_result = evaluate_expression(a, b, c, query)

        # If both KB and query are True, print the combination
        if kb_result and query_result:
            a_str = "True" if a else "False"
            b_str = "True" if b else "False"
            c_str = "True" if c else "False"
            kb_result_str = "True" if kb_result else "False"
            query_result_str = "True" if query_result else "False"
            print(f"{a_str:<5} {b_str:<5} {c_str:<5} {kb_result_str:<20} {query_result_str:<20}")

# Define the logical expressions as strings
kb = "(a or c) and (b or not c)" # Knowledge Base
query = "a or b" # Query to evaluate

# Generate the truth table and evaluate the knowledge base and query
truth_table_and_evaluation(kb, query)
```

```
a       b       c       KB                      Query
False   False   False   False                   False
False   False   True    False                   False
False   True    False   False                   True
False   True    True    True                    True
True    False   False   True                    True
True    False   True    False                   True
True    True    False   True                    True
True    True    True    True                    True

Combinations where both KB and Query are True:
a       b       c       KB                      Query
False   True    True    True                    True
True    False   False   True                    True
True    True    False   True                    True
True    True    True    True                    True
```
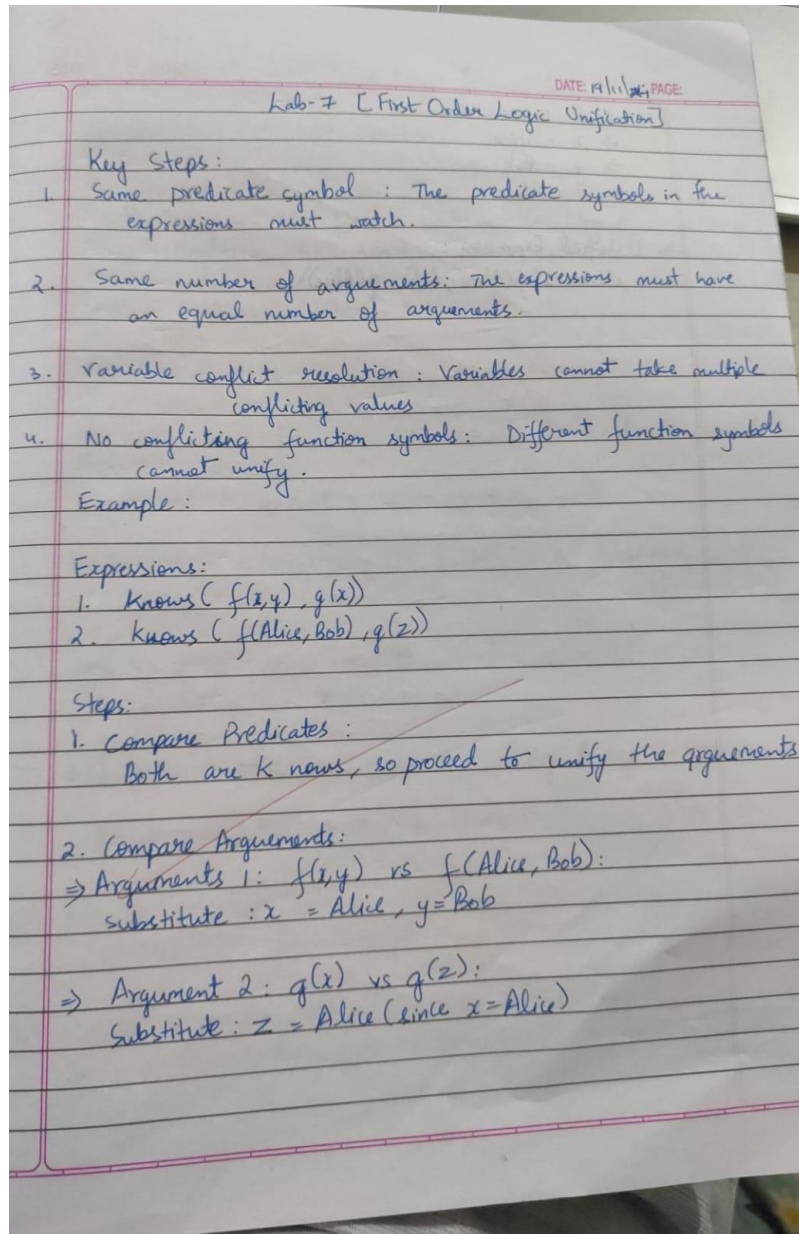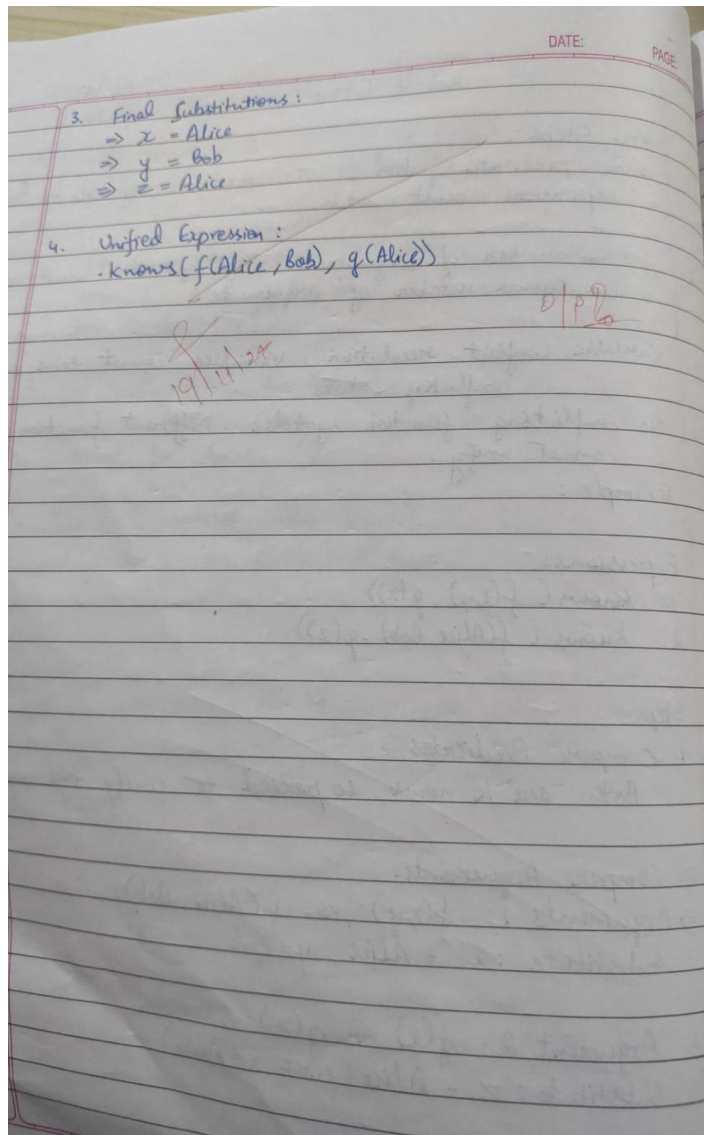
Program 7

Implement unification in first order logic

Algorithm:

DATE: 19/11/ PAGE:

Lab-7 [First Order Logic Unification]

Key Steps:

1. Same predicate symbol : The predicate symbols in the expressions must match.

2. Same number of arguments: The expressions must have an equal number of arguments.

3. Variable conflict resolution : Variables cannot take multiple conflicting values

4. No conflicting function symbols: Different function symbols cannot unify.

Example :

Expressions:
1. knows (f(x,y) , g(x))
2. knows (f(Alice, Bob) , g(z))

Steps:

1. Compare Predicates :
   Both are knows, so proceed to unify the arguments.

2. Compare Arguments:
   ⇒ Arguments 1: f(x,y) vs f(Alice, Bob):
      substitute : x = Alice, y = Bob

   ⇒ Argument 2: g(x) vs g(z):
      substitute : z = Alice (since x=Alice)

38

3. Final Substitutions :
   ⇒ x = Alice
   ⇒ y = Bob
   ⇒ z = Alice

4. Unified Expression :
   Knows (f(Alice, Bob), q(Alice))

Code:
import re

def occurs_check(var, x):
    """Checks if var occurs in x (to prevent circular substitutions)."""
    if var == x:
        return True
    elif isinstance(x, list): # If x is a compound expression (like a function or predicate)
        return any(occurs_check(var, xi) for xi in x)
    return False

def unify_var(var, x, subst):
    """Handles unification of a variable with another term."""

```python
        if var in subst: # If var is already substituted
            return unify(subst[var], x, subst)
        elif isinstance(x, (list, tuple)) and tuple(x) in subst: # Handle compound expressions
            return unify(var, subst[tuple(x)], subst)
        elif occurs_check(var, x):  # Check for circular references
            return "FAILURE"
        else:
            # Add the substitution to the set (convert list to tuple for hashability)
            subst[var] = tuple(x) if isinstance(x, list) else x
            return subst


def unify(x, y, subst=None):
    """

    Unifies two expressions x and y and returns the substitution set if they can be unified.
    Returns 'FAILURE' if unification is not possible.
    """

    if subst is None:
        subst = {} # Initialize an empty substitution set

    # Step 1: Handle cases where x or y is a variable or constant
    if x == y: # If x and y are identical
        return subst
    elif isinstance(x, str) and x.islower(): # If x is a variable
        return unify_var(x, y, subst)
    elif isinstance(y, str) and y.islower():  # If y is a variable
        return unify_var(y, x, subst)
    elif isinstance(x, list) and isinstance(y, list):  # If x and y are compound expressions (lists)
        if len(x) != len(y):  # Step 3: Different number of arguments
            return "FAILURE"

        # Step 2: Check if the predicate symbols (the first element) match
        if x[0] != y[0]:  # If the predicates/functions are different
            return "FAILURE"

        # Step 5: Recursively unify each argument
        for xi, yi in zip(x[1:], y[1:]): # Skip the predicate (first element)
            subst = unify(xi, yi, subst)
            if subst == "FAILURE":
                return "FAILURE"
        return subst
    else:  # If x and y are different constants or non-unifiable structures
        return "FAILURE"


def unify_and_check(expr1, expr2):
    """

    Attempts to unify two expressions and returns a tuple:
    (is_unified: bool, substitutions: dict or None)
```

```python
    """
    result = unify(expr1, expr2)
    if result == "FAILURE":
        return False, None
    return True, result

def display_result(expr1, expr2, is_unified, subst):
    print("Expression 1:", expr1)
    print("Expression 2:", expr2)
    if not is_unified:
        print("Result: Unification Failed")
    else:
        print("Result: Unification Successful")
        print("Substitutions:", {k: list(v) if isinstance(v, tuple) else v for k, v in subst.items()})

def parse_input(input_str):
    """Parses a string input into a structure that can be processed by the unification algorithm."""
    # Remove spaces and handle parentheses
    input_str = input_str.replace(" ", "")

    # Handle compound terms (like p(x, f(y)) -> ['p', 'x', ['f', 'y']])
    def parse_term(term):
        # Handle the compound term
        if '(' in term:
            match = re.match(r'([a-zA-Z0-9_]+)(.*)', term)
            if match:
                predicate = match.group(1)
                arguments_str = match.group(2)
                arguments = [parse_term(arg.strip()) for arg in arguments_str.split(',')]
                return [predicate] + arguments
        return term

    return parse_term(input_str)

# Main function to interact with the user
def main():
    while True:
        # Get the first and second terms from the user
        expr1_input = input("Enter the first expression (e.g., p(x, f(y))): ")
        expr2_input = input("Enter the second expression (e.g., p(a, f(z))): ")

        # Parse the input strings into the appropriate structures
        expr1 = parse_input(expr1_input)
        expr2 = parse_input(expr2_input)

        # Perform unification
        is_unified, result = unify_and_check(expr1, expr2)
```

```
        # Display the results
        display_result(expr1, expr2, is_unified, result)

        # Ask the user if they want to run another test
        another_test = input("Do you want to test another pair of expressions? (yes/no): ").strip().lower()
        if another_test != 'yes':
            break

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

```
Enter the first expression (e.g., p(x, f(y))): p(b,x,f(g(z)))
Enter the second expression (e.g., p(a, f(z))): p(z,f(y),f(y))
Expression 1: ['p', '(b', 'x', ['f', '(g(z)))']]
Expression 2: ['p', '(z', ['f', '(y)'], ['f', '(y))']]
Result: Unification Successful
Substitutions: {'(b': '(z', 'x': ['f', '(y)'], '(g(z)))': '(y))'}
Do you want to test another pair of expressions? (yes/no): yes
Enter the first expression (e.g., p(x, f(y))): p(x,h(y))
Enter the second expression (e.g., p(a, f(z))): p(a,f(z))
Expression 1: ['p', '(x', ['h', '(y))']]
Expression 2: ['p', '(a', ['f', '(z))']]
Result: Unification Failed
Do you want to test another pair of expressions? (yes/no): yes
Enter the first expression (e.g., p(x, f(y))): p(f(a),g(y))
Enter the second expression (e.g., p(a, f(z))): p(x,x)
Expression 1: ['p', '(f(a)', ['g', '(y))']]
Expression 2: ['p', '(x', 'x)']
Result: Unification Successful
Substitutions: {'(f(a)': '(x', 'x)': ['g', '(y))']}
Do you want to test another pair of expressions? (yes/no): no
```

Program 8

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

Algorithm:

3/12/24   Lab-8   First Order Logic : Forward Chaining   DATE: PAGE:

Consider the following problem:

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, and all the missiles were sold to it by Robert, who is an American citizen."

PT "Robert is criminal"

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

Let's see p, q and r are variables

$$American(p) \land Weapon(q) \land Sells(p,q,r) \land Hostile(r) \Rightarrow Criminal(p)$$

(i) Country A has some missiles
$$\exists x \; Owns(A,x) \land Missile(x)$$

(ii) All of the missiles were sold to country A by Robert
$$\forall x \; Missile(x) \land Own(A,x) \Rightarrow Sells(Robert, x, A)$$

(iii) Missiles are weapons
$$Missile(x) \Rightarrow weapon(x)$$

(iv) Enemy of America is known as hostile
$$\forall x \; Enemy(x, America) \Rightarrow Hostile(x)$$

43

Code:
```python
# Define the knowledge base (KB) as a set of facts
KB = set()

# Premises based on the provided FOL problem
KB.add('American(Robert)')
KB.add('Enemy(America, A)')
KB.add('Missile(T1)')
KB.add('Owns(A, T1)')

# Define inference rules
def modus_ponens(fact1, fact2, conclusion):
    """ Apply modus ponens inference rule: if fact1 and fact2 are true, then conclude conclusion """
    if fact1 in KB and fact2 in KB:
        KB.add(conclusion)
        print(f"Inferred: {conclusion}")

def forward_chaining():
    """ Perform forward chaining to infer new facts until no more inferences can be made """
    # 1. Apply: Missile(x) → Weapon(x)
    if 'Missile(T1)' in KB:
        KB.add('Weapon(T1)')
        print(f"Inferred: Weapon(T1)")

    # 2. Apply: Sells(Robert, T1, A) from Owns(A, T1) and Weapon(T1)
    if 'Owns(A, T1)' in KB and 'Weapon(T1)' in KB:
        KB.add('Sells(Robert, T1, A)')
        print(f"Inferred: Sells(Robert, T1, A)")

    # 3. Apply: Hostile(A) from Enemy(A, America)
    if 'Enemy(America, A)' in KB:
        KB.add('Hostile(A)')
        print(f"Inferred: Hostile(A)")

    # 4. Now, check if the goal is reached (i.e., if 'Criminal(Robert)' can be inferred)
    if 'American(Robert)' in KB and 'Weapon(T1)' in KB and 'Sells(Robert, T1, A)' in KB and 'Hostile(A)' in KB:
        KB.add('Criminal(Robert)')
        print("Inferred: Criminal(Robert)")

    # Check if we've reached our goal
    if 'Criminal(Robert)' in KB:
        print("Robert is a criminal!")
    else:
        print("No more inferences can be made.")

# Run forward chaining to attempt to derive the conclusion
```
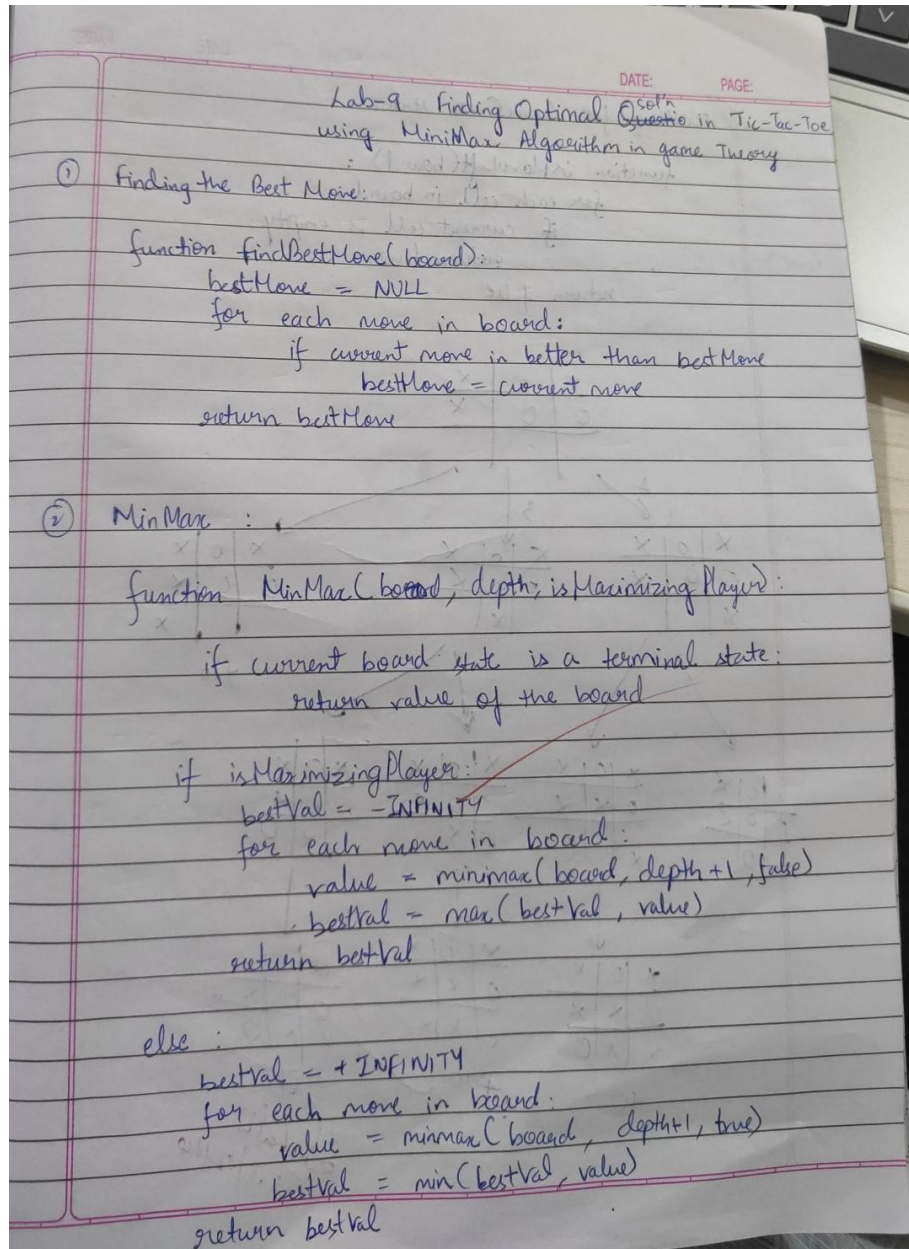
forward_chaining()

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

Program 9
Min Max Algorithm in Tic-Tac-Toe
Algorithm:

Code:

import sys

# Function to check if there's a winner or the game is drawn
def check_winner(board):
    for row in board:

```python
        if row[0] == row[1] == row[2] and row[0] != '_':
            return row[0]
    for col in range(3):
        if board[0][col] == board[1][col] == board[2][col] and board[0][col] != '_':
            return board[0][col]
    if board[0][0] == board[1][1] == board[2][2] and board[0][0] != '_':
        return board[0][0]
    if board[0][2] == board[1][1] == board[2][0] and board[0][2] != '_':
        return board[0][2]
    if all(board[i][j] != '_' for i in range(3) for j in range(3)):
        return 'Draw'
    return None

# Minimax function
def minimax(board, depth, is_maximizing):
    winner = check_winner(board)
    if winner == 'O':
        return 10 - depth
    if winner == 'X':
        return depth - 10
    if winner == 'Draw':
        return 0

    if is_maximizing:
        max_eval = -float('inf')
        for i in range(3):
            for j in range(3):
                if board[i][j] == '_':
                    board[i][j] = 'O'
                    eval = minimax(board, depth + 1, False)
                    board[i][j] = '_'
                    max_eval = max(max_eval, eval)
        return max_eval
    else:
        min_eval = float('inf')
        for i in range(3):
            for j in range(3):
                if board[i][j] == '_':
                    board[i][j] = 'X'
                    eval = minimax(board, depth + 1, True)
                    board[i][j] = '_'
                    min_eval = min(min_eval, eval)
        return min_eval

# Function to find the best move
def find_best_move(board):
    best_move = None
    best_value = -float('inf')
    for i in range(3):
        for j in range(3):
            if board[i][j] == '_':
                board[i][j] = 'O'
```

```
        move_value = minimax(board, 0, False)
        board[i][j] = '_'
        if move_value > best_value:
            best_value = move_value
            best_move = (i, j)
    return best_move

# Input and testing
if __name__ == "__main__":
    # Example board input from command line
    # e.g., python script.py "_ _ _" "_ X O" "O X _"
    board = [sys.argv[i + 1].split() for i in range(3)]
    move = find_best_move(board)
    print("Best Move:", move)
```

```
Tic Tac Toe!
You are 'O'. The AI is 'X'.
   |   |
---------
   |   |
---------
   |   |
---------
Enter your move (row and column: 0, 1, or 2): 2 2
Your move:
   |   |
---------
   |   |
---------
   |   | O
---------
AI is making its move...
AI's move:
   |   |
---------
   | X |
---------
   |   | O
---------
Enter your move (row and column: 0, 1, or 2): 0 0
Your move:
O |   |
---------
   | X |
```
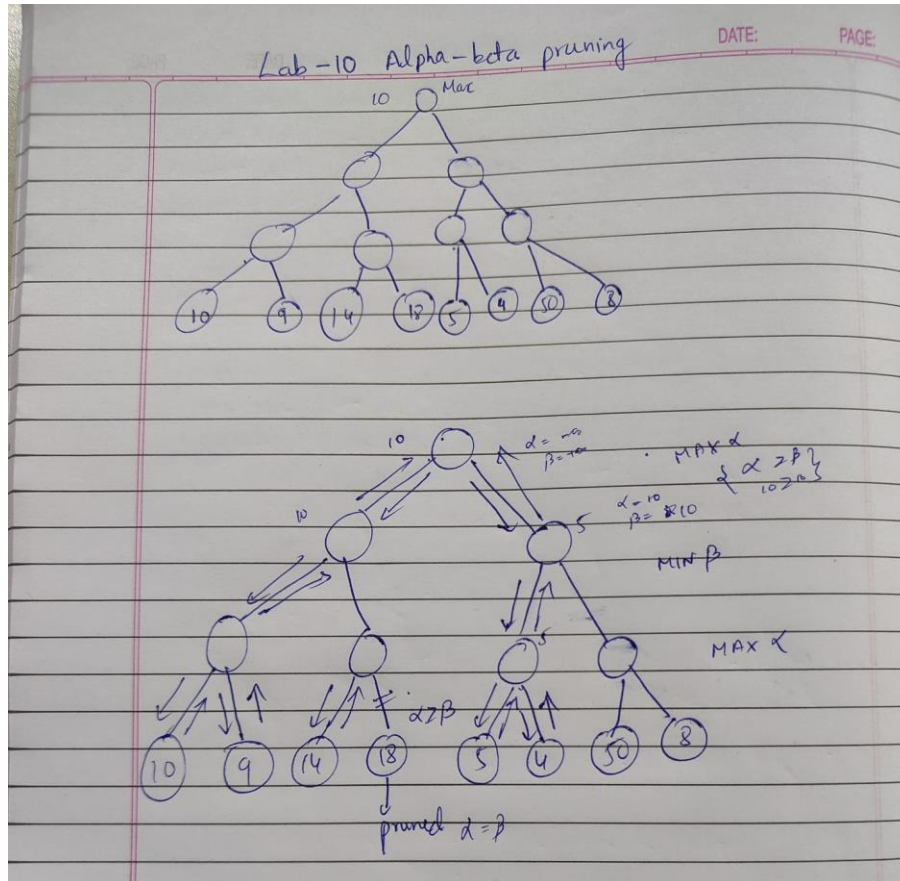
Program 10
Implement Alpha-Beta Pruning.

Algorithm:

Code:

```
# Alpha-Beta Pruning Implementation
def alpha_beta_pruning(node, alpha, beta, maximizing_player):
    # Base case: If it's a leaf node, return its value (simulating evaluation of the node)
    if type(node) is int:
        return node

    # If not a leaf node, explore the children
    if maximizing_player:
        max_eval = -float('inf')
        for child in node: # Iterate over children of the maximizer node
            eval = alpha_beta_pruning(child, alpha, beta, False)
            max_eval = max(max_eval, eval)
            alpha = max(alpha, eval) # Maximize alpha
            if beta <= alpha: # Prune the branch
                break
        return max_eval
    else:
        min_eval = float('inf')
        for child in node: # Iterate over children of the minimizer node
```

```python
        eval = alpha_beta_pruning(child, alpha, beta, True)
        min_eval = min(min_eval, eval)
        beta = min(beta, eval) # Minimize beta
        if beta <= alpha: # Prune the branch
            break
    return min_eval

# Function to build the tree from a list of numbers
def build_tree(numbers):
    # We need to build a tree with alternating levels of maximizers and minimizers
    # Start from the leaf nodes and work up
    current_level = [[n] for n in numbers]

    while len(current_level) > 1:
        next_level = []
        for i in range(0, len(current_level), 2):
            if i + 1 < len(current_level):
                next_level.append(current_level[i] + current_level[i + 1]) # Combine two nodes
            else:
                next_level.append(current_level[i]) # Odd number of elements, just carry forward
        current_level = next_level

    return current_level[0] # Return the root node, which is a maximizer

# Main function to run alpha-beta pruning
def main():
    # Input: User provides a list of numbers
    numbers = list(map(int, input("Enter numbers for the game tree (space-separated): ").split()))

    # Build the tree with the given numbers
    tree = build_tree(numbers)

    # Parameters: Tree, initial alpha, beta, and the root node is a maximizing player
    alpha = -float('inf')
    beta = float('inf')
    maximizing_player = True  # The root node is a maximizing player

    # Perform alpha-beta pruning and get the final result
    result = alpha_beta_pruning(tree, alpha, beta, maximizing_player)

    print("Final Result of Alpha-Beta Pruning:", result)

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

```
Enter numbers for the game tree (space-separated): 10 9 14 18 5 4 50 3
Final Result of Alpha-Beta Pruning: 50
```