AI Lab Assignments to be submitted for Final Lab Exam
Programs to be executed in Python

1. Breadth-First Search (BFS) Algorithm

Write a Python program to find the shortest path in an unweighted graph using BFS.

Solution

```
from collections import deque
def bfs_shortest_path(graph, start, goal):
    queue = deque([[start]])
    visited = set()

    while queue:
        path = queue.popleft()
        node = path[-1]

        if node == goal:
            return path
        elif node not in visited:
            for neighbor in graph.get(node, []):
                new_path = list(path)
                new_path.append(neighbor)
                queue.append(new_path)

            visited.add(node)
    return None
def create_graph():
    graph = {}
    num_edges = int(input("Enter the number of edges: "))

    print("Enter each edge in the format 'node1 node2':")
    for _ in range(num_edges):
        u, v = input().split()
        if u not in graph:
            graph[u] = []
        if v not in graph:
            graph[v] = []
        graph[u].append(v)
        graph[v].append(u)
    return graph

graph = create_graph()
start = input("Enter the start node: ")
goal = input("Enter the goal node: ")

shortest_path = bfs_shortest_path(graph, start, goal)

if shortest_path:
    print(f"Shortest path from {start} to {goal}: {' -> '.join(shortest_path)}")
```

```
    else:
        print(f"No path exists from {start} to {goal}.")
```

Output

```
PS C:\Users\anit4\OneDrive\Desktop\xyz> python -u "c:\Users\anit4\OneDrive\Desktop\xyz\temp.py"
Enter the number of edges: 5
Enter each edge in the format 'node1 node2':
1 2
1 3
2 4
3 4
4 5
Enter the start node: 1
Enter the goal node: 5
Shortest path from 1 to 5: 1 -> 2 -> 4 -> 5
PS C:\Users\anit4\OneDrive\Desktop\xyz>
```

2. Depth-First Search (DFS) Algorithm

Implement DFS in Python to explore a graph and find paths or cycles.

Solution

```
def dfs(graph, start, visited=None, path=None):

    if visited is None:
        visited = set()
    if path is None:
        path = []

    visited.add(start)
    path.append(start)

    for neighbor in graph[start]:
        if neighbor not in visited:
            dfs(graph, neighbor, visited, path)
        elif neighbor in path:
            print("Cycle detected!")
            return

    path.pop()

def find_path_dfs(graph, start, goal, visited=None, path=None):
    if visited is None:
        visited = set()
    if path is None:
        path = []

    visited.add(start)
    path.append(start)

    if start == goal:
        return path[:]

    for neighbor in graph[start]:
        if neighbor not in visited:
```

```
            result = find_path_dfs(graph, neighbor, goal, visited, path)
            if result:
                return result

    path.pop()
    visited.remove(start)
    return None

def build_graph():
    graph = {}
    edges = int(input("Enter the number of edges: "))

    for _ in range(edges):
        u, v = input("Enter edge : ").split()

        if u not in graph:
            graph[u] = []
        if v not in graph:
            graph[v] = []

        graph[u].append(v)
        graph[v].append(u)

    return graph

graph = build_graph()
print("\nGraph:", graph)

start_node = input("\nEnter the starting node for DFS traversal: ")
dfs(graph, start_node)

goal_node = input("\nEnter the goal node for finding a path from the start node: ")
path = find_path_dfs(graph, start_node, goal_node)
if path:
    print(f"Path from {start_node} to {goal_node}: {path}")
else:
    print(f"No path found from {start_node} to {goal_node}.")
```

```
PROBLEMS    OUTPUT    DEBUG CONSOLE    TERMINAL    PORTS    SQL CONSOLE

PS C:\Users\anit4\OneDrive\Desktop\xyz> python -u "c:\Users\anit4\OneDrive\Desktop\xyz\temp.py"
Enter the number of edges: 5
Enter edge : 1 2
Enter edge : 1 3
Enter edge : 2 3
Enter edge : 2 5
Enter edge : 4 5

Graph: {'1': ['2', '3'], '2': ['1', '3', '5'], '3': ['1', '2'], '5': ['2', '4'], '4': ['5']}

Enter the starting node for DFS traversal: 1
Cycle detected!
Cycle detected!

Enter the goal node for finding a path from the start node: 5
Path from 1 to 5: ['1', '2', '5']
PS C:\Users\anit4\OneDrive\Desktop\xyz>
```

3. A Search Algorithm*

Develop a Python program to solve the shortest path problem using the A* algorithm with heuristics.

Solution

```
import heapq

class Graph:
    def __init__(self):
        self.edges = {}
        self.h = {}

    def add_edge(self, u, v, cost):
        if u not in self.edges:
            self.edges[u] = []
        if v not in self.edges:
            self.edges[v] = []
        self.edges[u].append((v, cost))
        self.edges[v].append((u, cost))

    def set_heuristic(self, node, heuristic_value):
        self.h[node] = heuristic_value

    def a_star(self, start, goal):
        open_set = []
        heapq.heappush(open_set, (0, start))
        came_from = {}
        g_score = {start: 0}
        f_score = {start: self.h.get(start, 0)}

        while open_set:
            current_f_score, current = heapq.heappop(open_set)

            if current == goal:
                path = []
```

```
            while current in came_from:
                path.append(current)
                current = came_from[current]
            path.append(start)
            return path[::-1]

        for neighbor, cost in self.edges.get(current, []):
            tentative_g_score = g_score[current] + cost

            if tentative_g_score < g_score.get(neighbor, float('inf')):
                came_from[neighbor] = current
                g_score[neighbor] = tentative_g_score
                f_score[neighbor] = tentative_g_score + self.h.get(neighbor, 0)
                heapq.heappush(open_set, (f_score[neighbor], neighbor))

    return None

def create_graph():
    graph = Graph()
    num_edges = int(input("Enter the number of edges: "))
    print("Enter each edge in the format 'node1 node2 cost':")
    for _ in range(num_edges):
        u, v, cost = input().split()
        graph.add_edge(u, v, float(cost))

    num_heuristics = int(input("Enter the number of heuristic values: "))
    print("Enter heuristic values in the format 'node heuristic_value':")
    for _ in range(num_heuristics):
        node, heuristic_value = input().split()
        graph.set_heuristic(node, float(heuristic_value))

    return graph

graph = create_graph()
start = input("Enter the start node: ")
goal = input("Enter the goal node: ")
shortest_path = graph.a_star(start, goal)

if shortest_path:
    print(f"Shortest path from {start} to {goal}: {' -> '.join(shortest_path)}")
else:
    print(f"No path exists from {start} to {goal}.")
```

```
PS C:\Users\anit4\OneDrive\Desktop\xyz> python -u "c:\Users\anit4\OneDrive\Des

Enter the number of edges: 5
Enter each edge in the format 'node1 node2 cost':
1 2 1
1 3 4
2 4 2
3 4 5
4 5 1
Enter the number of heuristic values: 5
Enter heuristic values in the format 'node heuristic_value':
1 7
2 6
3 2
4 1
5 0
Enter the start node: 1
Enter the goal node: 5
Shortest path from 1 to 5: 1 -> 2 -> 4 -> 5
PS C:\Users\anit4\OneDrive\Desktop\xyz>
```

---

4. Tic-Tac-Toe Game (AI vs Player using Minimax)

Build a Tic-Tac-Toe game where the AI uses the Minimax algorithm to play against a human player.

Solution

```
import math

board = [' ' for _ in range(9)]

def print_board():
    for row in [board[i:i+3] for i in range(0, 9, 3)]:
        print('| ' + ' | '.join(row) + ' |')
    print()

def check_winner(board, player):
    win_conditions = [
        [0, 1, 2], [3, 4, 5], [6, 7, 8],
        [0, 3, 6], [1, 4, 7], [2, 5, 8],
        [0, 4, 8], [2, 4, 6]
    ]
    for condition in win_conditions:
        if all(board[i] == player for i in condition):
            return True
    return False

def is_board_full(board):
    return ' ' not in board

def minimax(board, depth, is_maximizing):
```

```
    if check_winner(board, 'O'):
        return 1
    if check_winner(board, 'X'):
        return -1
    if is_board_full(board):
        return 0

    if is_maximizing:
        best_score = -math.inf
        for i in range(9):
            if board[i] == ' ':
                board[i] = 'O'
                score = minimax(board, depth + 1, False)
                board[i] = ' '
                best_score = max(score, best_score)
        return best_score
    else:
        best_score = math.inf
        for i in range(9):
            if board[i] == ' ':
                board[i] = 'X'
                score = minimax(board, depth + 1, True)
                board[i] = ' '
                best_score = min(score, best_score)
        return best_score

def best_move():
    best_score = -math.inf
    move = -1
    for i in range(9):
        if board[i] == ' ':
            board[i] = 'O'
            score = minimax(board, 0, False)
            board[i] = ' '
            if score > best_score:
                best_score = score
                move = i
    return move

def player_move():
    while True:
        try:
            move = int(input("Enter your move (1-9): ")) - 1
            if move < 0 or move >= 9 or board[move] != ' ':
                raise ValueError
            board[move] = 'X'
            break
        except ValueError:
            print("Invalid move. Please enter a number between 1 and 9 corresponding to
an empty spot.")
```

```
def play_game():
    print("Welcome to Tic-Tac-Toe! You are 'X' and the AI is 'O'.")
    print_board()
    while True:
        player_move()
        print_board()
        if check_winner(board, 'X'):
            print("Congratulations! You win!")
            break
        if is_board_full(board):
            print("It's a draw!")
            break

        print("AI is making a move...")
        ai_move = best_move()
        board[ai_move] = 'O'
        print_board()
        if check_winner(board, 'O'):
            print("AI wins! Better luck next time.")
            break
        if is_board_full(board):
            print("It's a draw!")
            break

play_game()
```

```
Welcome to Tic-Tac-Toe! You are 'X' and the AI is 'O'.
|   |   |   |
|   |   |   |
|   |   |   |

Enter your move (1-9): 5
|   |   |   |
|   | X |   |
|   |   |   |

AI is making a move...
| O |   |   |
|   | X |   |
|   |   |   |

Enter your move (1-9): 7
| O |   |   |
|   | X |   |
| X |   |   |

AI is making a move...
| O |   | O |
|   | X |   |
| X |   |   |

Enter your move (1-9): 2
| O | X | O |
|   | X |   |
| X |   |   |

AI is making a move...
| O | X | O |
|   | X |   |
| X | O |   |

Enter your move (1-9): 4
| O | X | O |
| X | X |   |
| X | O |   |

AI is making a move...
| O | X | O |
| X | X | O |
| X | O |   |

Enter your move (1-9): 9
| O | X | O |
| X | X | O |
| X | O | X |

It's a draw!
```

5. <u>Sudoku Solver Using Backtracking</u>

Write a Python program that solves a Sudoku puzzle using a backtracking algorithm.

<u>Solution</u>

```python
def print_board(board):
    for i in range(9):
        for j in range(9):
            print(board[i][j], end=" ")
        print()

def is_valid(board, row, col, num):
    for x in range(9):
        if board[row][x] == num:
            return False

    for x in range(9):
        if board[x][col] == num:
            return False

    start_row = row - row % 3
    start_col = col - col % 3
    for i in range(3):
        for j in range(3):
            if board[i + start_row][j + start_col] == num:
                return False

    return True

def solve_sudoku(board):
    empty = find_empty_location(board)
    if not empty:
        return True
    row, col = empty

    for num in range(1, 10):
        if is_valid(board, row, col, num):
            board[row][col] = num

            if solve_sudoku(board):
                return True

            board[row][col] = 0  # Backtrack

    return False

def find_empty_location(board):
    for i in range(9):
        for j in range(9):
            if board[i][j] == 0:
                return (i, j)
    return None

def input_board():
    print("Enter the Sudoku puzzle row by row (use 0 for empty cells):")
    board = []
```

```
    for i in range(9):
        row = list(map(int, input(f"Row {i+1}: ").strip().split()))
        if len(row) != 9:
            print("Please enter exactly 9 numbers for each row.")
            return None
        board.append(row)

    return board

def main():
    board = input_board()
    if board is None:
        return
    print("\nSudoku Puzzle:")
    print_board(board)

    if solve_sudoku(board):
        print("\nSolved Sudoku:")
        print_board(board)
    else:
        print("\nNo solution exists for the provided Sudoku puzzle.")

main()
```

```
Enter the Sudoku puzzle row by row (use 0 for empty cells):
Row 1: 5 3 0 0 7 0 0 0 0
Row 2: 6 0 0 1 9 5 0 0 0
Row 3: 0 9 8 0 0 0 0 6 0
Row 4: 8 0 0 0 6 0 0 0 3
Row 5: 4 0 0 8 0 3 0 0 1
Row 6: 7 0 0 0 2 0 0 0 6
Row 7: 0 6 0 0 0 0 2 8 0
Row 8: 0 0 0 4 1 9 0 0 5
Row 9: 0 0 0 0 8 0 0 7 9

Sudoku Puzzle:
5 3 0 0 7 0 0 0 0
6 0 0 1 9 5 0 0 0
0 9 8 0 0 0 0 6 0
8 0 0 0 6 0 0 0 3
4 0 0 8 0 3 0 0 1
7 0 0 0 2 0 0 0 6
0 6 0 0 0 0 2 8 0
0 0 0 4 1 9 0 0 5
0 0 0 0 8 0 0 7 9

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

6. 8-Puzzle Solver Using A* Algorithm

Solve the 8-puzzle problem using A* algorithm with a heuristic function like Manhattan distance.

Solution

```
import heapq
```

```
class PuzzleState:
    def __init__(self, board, g, h, parent=None):
        self.board = board
        self.g = g
        self.h = h
        self.f = g + h
        self.parent = parent

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

def manhattan_distance(board, goal):
    distance = 0
    for i in range(3):
        for j in range(3):
            if board[i][j] != 0:
                x, y = divmod(goal.index(board[i][j]), 3)
                distance += abs(x - i) + abs(y - j)
    return distance

def flatten(board):
    return [num for row in board for num in row]

def goal_state():
    return [1, 2, 3, 4, 5, 6, 7, 8, 0]

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

def print_board(board):
    for row in board:
        print(" ".join(map(str, row)))
    print()

def get_neighbors(board):
    neighbors = []
    zero_pos = flatten(board).index(0)
    i, j = divmod(zero_pos, 3)

    moves = {
        'up': (i - 1, j),
        'down': (i + 1, j),
        'left': (i, j - 1),
        'right': (i, j + 1)
    }

    for direction, (x, y) in moves.items():
```

```
        if 0 <= x < 3 and 0 <= y < 3:
            new_board = [row[:] for row in board]
            new_board[i][j], new_board[x][y] = new_board[x][y], new_board[i][j]
            neighbors.append(new_board)
    return neighbors

def a_star(initial_board):
    goal = goal_state()
    open_set = []
    g = 0
    h = manhattan_distance(initial_board, goal)
    initial_state = PuzzleState(initial_board, g, h)
    heapq.heappush(open_set, initial_state)
    closed_set = set()

    while open_set:
        current = heapq.heappop(open_set)

        if flatten(current.board) == goal:
            return reconstruct_path(current)

        closed_set.add(tuple(flatten(current.board)))
        for neighbor in get_neighbors(current.board):
            if tuple(flatten(neighbor)) in closed_set:
                continue
            g = current.g + 1
            h = manhattan_distance(neighbor, goal)
            neighbor_state = PuzzleState(neighbor, g, h, current)
            heapq.heappush(open_set, neighbor_state)

    return None

def input_board():
    print("Enter the initial 8-puzzle board configuration as a 3x3 grid (use 0 for the empty
space):")
    board = []
    for i in range(3):
        row = list(map(int, input(f"Row {i + 1}: ").strip().split()))
        if len(row) != 3:
            print("Each row must have exactly 3 numbers.")
            return None
        board.append(row)
    return board

def main():
    initial_board = input_board()
    if initial_board is None:
        return

    print("\nInitial Board:")
    print_board(initial_board)
```

```
    solution = a_star(initial_board)
    if solution:
        print("Solution found:")
        for step, board in enumerate(solution):
            print(f"Step {step}:")
            print_board(board)
    else:
        print("No solution exists for this configuration.")

main()
```

Output

```
Enter the initial 8-puzzle board configuration as a 3x3 grid (use 0 for the empty space):
Row 1: 1 2 3
Row 2: 4 0 5
Row 3: 7 8 6

Initial Board:
1 2 3
4 0 5
7 8 6

Solution found:
Step 0:
1 2 3
4 0 5
7 8 6

Step 1:
1 2 3
4 5 0
7 8 6

Step 2:
1 2 3
4 5 6
7 8 0
```

7. Hill Climbing Algorithm

Implement the Hill Climbing algorithm to solve an optimization problem, such as maximizing a function.

Solution

```
import random

def objective_function(x):
    return -x**2 + 5*x + 20

def hill_climbing(initial_x, step_size=0.1, max_iterations=1000):
    current_x = initial_x
    current_value = objective_function(current_x)

    for iteration in range(max_iterations):
        neighbors = [current_x + step_size, current_x - step_size]
        next_x = max(neighbors, key=objective_function)
        next_value = objective_function(next_x)
```

```
        if next_value > current_value:
            current_x, current_value = next_x, next_value
        else:
            break

        print(f"Iteration {iteration + 1}: x = {current_x:.4f}, f(x) = {current_value:.4f}")

    return current_x, current_value

initial_x = float(input("Enter an initial x value: "))
solution_x, solution_value = hill_climbing(initial_x)

print("\nFinal Solution:")
print(f"x = {solution_x:.4f}, f(x) = {solution_value:.4f}")
```

Output

```
PS C:\Users\anit4\OneDrive\Desktop\xyz> python -u "c:\Users\a

Enter an initial x value: 0.0
Iteration 1: x = 0.1000, f(x) = 20.4900
Iteration 2: x = 0.2000, f(x) = 20.9600
Iteration 3: x = 0.3000, f(x) = 21.4100
Iteration 4: x = 0.4000, f(x) = 21.8400
Iteration 5: x = 0.5000, f(x) = 22.2500
Iteration 6: x = 0.6000, f(x) = 22.6400
Iteration 7: x = 0.7000, f(x) = 23.0100
Iteration 8: x = 0.8000, f(x) = 23.3600
Iteration 9: x = 0.9000, f(x) = 23.6900
Iteration 10: x = 1.0000, f(x) = 24.0000
Iteration 11: x = 1.1000, f(x) = 24.2900
Iteration 12: x = 1.2000, f(x) = 24.5600
Iteration 13: x = 1.3000, f(x) = 24.8100
Iteration 14: x = 1.4000, f(x) = 25.0400
Iteration 15: x = 1.5000, f(x) = 25.2500
Iteration 16: x = 1.6000, f(x) = 25.4400
Iteration 17: x = 1.7000, f(x) = 25.6100
Iteration 18: x = 1.8000, f(x) = 25.7600
Iteration 19: x = 1.9000, f(x) = 25.8900
Iteration 20: x = 2.0000, f(x) = 26.0000
Iteration 21: x = 2.1000, f(x) = 26.0900
Iteration 22: x = 2.2000, f(x) = 26.1600
Iteration 23: x = 2.3000, f(x) = 26.2100
Iteration 24: x = 2.4000, f(x) = 26.2400
Iteration 25: x = 2.5000, f(x) = 26.2500

Final Solution:
x = 2.5000, f(x) = 26.2500
PS C:\Users\anit4\OneDrive\Desktop\xyz>
```

8. Genetic Algorithm for Optimization

Write a Python program to solve an optimization problem using Genetic Algorithms.

Solution

```
import random

def objective_function(x):
    return -x**2 + 5*x + 20
```

```python
def generate_population(size, x_min, x_max):
    return [random.uniform(x_min, x_max) for _ in range(size)]

def selection(population, fitnesses, num_parents):
    selected_parents = []
    for _ in range(num_parents):
        i, j = random.sample(range(len(population)), 2)
        if fitnesses[i] > fitnesses[j]:
            selected_parents.append(population[i])
        else:
            selected_parents.append(population[j])
    return selected_parents

def crossover(parent1, parent2):
    return (parent1 + parent2) / 2

def mutate(individual, mutation_rate, x_min, x_max):
    if random.random() < mutation_rate:
        return max(min(individual + random.uniform(-1, 1), x_max), x_min)
    return individual

def genetic_algorithm(pop_size, generations, x_min, x_max, mutation_rate):
    population = generate_population(pop_size, x_min, x_max)

    for generation in range(generations):
        fitnesses = [objective_function(x) for x in population]
        parents = selection(population, fitnesses, pop_size // 2)

        next_generation = []
        while len(next_generation) < pop_size:
            parent1, parent2 = random.sample(parents, 2)
            offspring = crossover(parent1, parent2)
            offspring = mutate(offspring, mutation_rate, x_min, x_max)
            next_generation.append(offspring)

        population = next_generation

        best_individual = max(population, key=objective_function)
        best_fitness = objective_function(best_individual)
        print(f"Generation {generation + 1}: x = {best_individual:.4f}, f(x) =
{best_fitness:.4f}")

    best_individual = max(population, key=objective_function)
    return best_individual, objective_function(best_individual)

population_size = int(input("Enter population size: "))
num_generations = int(input("Enter number of generations: "))
x_min = float(input("Enter minimum x value: "))
x_max = float(input("Enter maximum x value: "))
mutation_rate = float(input("Enter mutation rate (0-1): "))
```

```
solution_x, solution_value = genetic_algorithm(population_size, num_generations,
x_min, x_max, mutation_rate)

print("\nBest solution found:")
print(f"x = {solution_x:.4f}, f(x) = {solution_value:.4f}")
```

Output

```
Enter population size: 10
Enter number of generations: 20
Enter minimum x value: -10
Enter maximum x value: 10
Enter mutation rate (0-1): 0.1
Generation 1: x = 5.6080, f(x) = 16.5902
Generation 2: x = 5.5512, f(x) = 16.9401
Generation 3: x = 5.6334, f(x) = 16.4319
Generation 4: x = 5.6745, f(x) = 16.1728
Generation 5: x = 5.6745, f(x) = 16.1728
Generation 6: x = 5.7067, f(x) = 15.9669
Generation 7: x = 4.7297, f(x) = 21.2786
Generation 8: x = 4.7297, f(x) = 21.2786
Generation 9: x = 4.7297, f(x) = 21.2786
Generation 10: x = 4.7297, f(x) = 21.2786
Generation 11: x = 4.7297, f(x) = 21.2786
Generation 12: x = 4.6425, f(x) = 21.6595
Generation 13: x = 4.7297, f(x) = 21.2786
Generation 14: x = 4.7297, f(x) = 21.2786
Generation 15: x = 4.7297, f(x) = 21.2786
Generation 16: x = 4.7297, f(x) = 21.2786
Generation 17: x = 4.7297, f(x) = 21.2786
Generation 18: x = 4.7297, f(x) = 21.2786
Generation 19: x = 4.2671, f(x) = 23.1273
Generation 20: x = 4.2671, f(x) = 23.1273

Best solution found:
x = 4.2671, f(x) = 23.1273
```

9. N-Queens Problem

Solve the N-Queens problem using a backtracking algorithm in Python.

Solution

```
def print_board(board):
  for row in board:
    print(" ".join("Q" if col else "." for col in row))
  print()
def is_safe(board, row, col, N):
for i in range(row):
    if board[i][col] == 1:
      return False

  # Check the upper left diagonal for any queens
  for i, j in zip(range(row, -1, -1), range(col, -1, -1)):
    if board[i][j] == 1:
      return False

  # Check the upper right diagonal for any queens
  for i, j in zip(range(row, -1, -1), range(col, N)):
    if board[i][j] == 1:
      return False
```

```
    return True

def solve_n_queens(board, row, N):
    if row >= N:
        return True

    for col in range(N):
        if is_safe(board, row, col, N):
            board[row][col] = 1  # Place the queen

            if solve_n_queens(board, row + 1, N):
                return True

            board[row][col] = 0  # Backtrack

    return False

def n_queens(N):
    board = [[0 for _ in range(N)] for _ in range(N)]

    if solve_n_queens(board, 0, N):
        print("Solution found:")
        print_board(board)
    else:
        print("No solution exists for this board size.")

N = int(input("Enter the number of queens (N): "))
n_queens(N)
```

Output

```
PS C:\Users\anit4\OneDrive\Desktop\xyz> python

Enter the number of queens (N): 4
Solution found:
. Q . .
. . . Q
Q . . .
. . Q .

PS C:\Users\anit4\OneDrive\Desktop\xyz>
```

---

10. Bayesian Inference for Probabilistic Reasoning

Implement Bayesian inference in Python to predict the probability of a condition, like disease diagnosis.

Solution

```
p_disease = float(input("Enter the prior probability of having the disease (P(D)): "))
```

```
p_positive_given_disease = float(input("Enter the probability of testing positive given the disease (P(T|D)): "))

p_positive_given_no_disease = float(input("Enter the probability of testing positive given no disease (P(T|~D)): "))
p_no_disease = 1 - p_disease
p_test_positive = (p_positive_given_disease * p_disease) + (p_positive_given_no_disease * p_no_disease)

p_disease_given_positive = (p_positive_given_disease * p_disease) / p_test_positive

print(f"\nProbability of having the disease given a positive test result (P(D|T)): {p_disease_given_positive:.4f}")
```

```
Enter the prior probability of having the disease (P(D)): 0.01
Enter the probability of testing positive given the disease (P(T|D)): 0.95
Enter the probability of testing positive given no disease (P(T|~D)): 0.05

Probability of having the disease given a positive test result (P(D|T)): 0.1610
```

11. Simple Chatbot Using Rule-Based Logic

Create a rule-based chatbot using Python that can respond to simple queries.

Solution

```
import datetime
import requests

def get_weather(city_name):
   try:
      city = city_name.replace(" ", "+")
      url = f"https://wttr.in/{city}?format=%l:+%c+%t"
      response = requests.get(url)
      if response.status_code == 200:
         print(response.text)
      else:
         print("Could not fetch weather data. Please try again.")
   except Exception as e:
      print("An error occurred:", e)

def chatbot_response(user_input):
   user_input = user_input.lower()
   if "hello" in user_input or "hi" in user_input:
      return "Hello! How can I assist you today?"
   elif "how are you" in user_input:
      return "I'm just a bot, but thanks for asking! How can I help you?"
   elif "time" in user_input:
      current_time = datetime.datetime.now().strftime("%I:%M %p")
      return f"The current time is {current_time}."
   elif "date" in user_input:
      current_date = datetime.datetime.now().strftime("%B %d, %Y")
      return f"Today’s date is {current_date}."
   elif "thank you" in user_input:
```

```
        return "You are most welcome :)"
    elif "weather" in user_input:
        city = input("Enter the Name of City -> ")
        get_weather(city)
        return "Have a Nice Day :)"
    elif "bye" in user_input or "exit" in user_input:
        return "Goodbye! Have a great day."
    else:
        return "I'm sorry, I didn't understand that. Could you please rephrase?"

print("Chatbot: Hello! Type 'bye' to exit the chat.")

while True:
    user_input = input("You: ")
    if user_input.lower() in ["bye", "exit"]:
        print("Chatbot: Goodbye! Have a great day.")
        break
    response = chatbot_response(user_input)
    print(f"Chatbot: {response}")
```

```
Chatbot: Hello! Type 'bye' to exit the chat.
You: hello
Chatbot: Hello! How can I assist you today?
You: weather today
Enter the Name of City -> kolkata
kolkata: 🌫 +31°C
Chatbot: Have a Nice Day :)
You: bye
Chatbot: Goodbye! Have a great day.
```

12. <u>Decision Tree Classifier (Using Scikit-learn)</u>

<u>Solution</u>

```
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split from sklearn.tree import
DecisionTreeClassifier
from sklearn.metrics import accuracy_score, classification_report, confusion_matrix
iris = load_iris()
X = iris.data y = iris.target

X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.3, random_state=42)

clf = DecisionTreeClassifier(random_state=42)

clf.fit(X_train, y_train) y_pred = clf.predict(X_test)
accuracy = accuracy_score(y_test, y_pred)
```

```
print(f"Accuracy: {accuracy:.2f}")

print("\nClassification Report:") print(classification_report(y_test, y_pred, target_names=iris.target_names))

print("\nConfusion Matrix:") print(confusion_matrix(y_test, y_pred))
```

```
Accuracy: 1.00

Classification Report:
              precision    recall  f1-score   support

      setosa       1.00      1.00      1.00        19
  versicolor       1.00      1.00      1.00        13
   virginica       1.00      1.00      1.00        13

    accuracy                           1.00        45
   macro avg       1.00      1.00      1.00        45
weighted avg       1.00      1.00      1.00        45


Confusion Matrix:
[[19  0  0]
 [ 0 13  0]
 [ 0  0 13]]
```

Programs to be executed in Prolog

1. Family Relationship Expert System

Write a Prolog program to represent a family tree and allow queries about relationships (e.g., parent, sibling, cousin).

Solution:

```
parent(john, mary).
parent(john, michael).
parent(susan, mary).
parent(susan, michael).
parent(mary, linda).
parent(mary, james).
parent(michael, alex).
parent(michael, emma).
parent(alex, sophia).
parent(alex, tom).

male(john).
male(michael).
male(alex).
male(james).
male(tom).
female(susan).
female(mary).
female(linda).
```

```
female(emma).
female(sophia).
```

Output:

```
?- parent(john, mary).
true .

?- sibling(mary, michael).
true .

?- cousin(linda, emma).
true .

?- parent(susan, X).
X = mary .

?- sibling(Z, linda).
Z = james .

?- cousin(Y, alex).
Y = linda .
```

2. Expert System for Medical Diagnosis

Develop a simple expert system in Prolog that diagnoses a disease based on symptoms provided by the user.

Solution:

```
% Define diseases and their symptoms
disease(flu) :-
   symptom(fever),
   symptom(headache),
   symptom(body_ache),
   symptom(chills),
   symptom(sore_throat),
   symptom(cough),
   symptom(fatigue).

disease(cold) :-
   symptom(sneezing),
   symptom(runny_nose),
   symptom(sore_throat),
   symptom(cough),
   symptom(congestion).

disease(allergy) :-
   symptom(sneezing),
   symptom(runny_nose),
   symptom(itchy_eyes),
   symptom(watery_eyes),
   symptom(congestion).

disease(covid_19) :-
```

```
  symptom(fever),
  symptom(cough),
  symptom(shortness_of_breath),
  symptom(fatigue),
  symptom(loss_of_taste_or_smell).

% Diagnose the disease based on symptoms
diagnose(Disease) :-
  disease(Disease),
  !.

% Ask the user about symptoms
ask(Symptom) :-
  write('Do you have '), write(Symptom), write('? (yes/no): '),
  read(Response), nl,
  ((Response == yes) -> assert(symptom(Symptom)) ; true).

% Start the diagnosis process
start_diagnosis :-
  retractall(symptom(_)),
  write('Medical Diagnosis Expert System'), nl,
  write('Please answer the following questions:'), nl,
  ask(fever),
  ask(headache),
  ask(body_ache),
  ask(chills),
  ask(sore_throat),
  ask(cough),
  ask(fatigue),
  ask(sneezing),
  ask(runny_nose),
  ask(itchy_eyes),
  ask(watery_eyes),
  ask(congestion),
  ask(shortness_of_breath),
  ask(loss_of_taste_or_smell),
  (diagnose(Disease) ->
    write('You may have: '), write(Disease), nl
  ;
    write('Diagnosis inconclusive. Please consult a doctor.'), nl).
```

Output:

```
?- start_diagnosis.
Medical Diagnosis Expert System
Please answer the following questions:
Do you have fever? (yes/no): yes.

Do you have headache? (yes/no): |: yes.

Do you have body_ache? (yes/no): |: yes.

Do you have chills? (yes/no): |: no.

Do you have sore_throat? (yes/no): |: yes.

Do you have cough? (yes/no): |: no.

Do you have fatigue? (yes/no): |: yes.

Do you have sneezing? (yes/no): |: yes.

Do you have runny_nose? (yes/no): |: no.

Do you have itchy_eyes? (yes/no): |: no.

Do you have watery_eyes? (yes/no): |: no.

Do you have congestion? (yes/no): |: no.

Do you have shortness_of_breath? (yes/no): |: yes.

Do you have loss_of_taste_or_smell? (yes/no): |: no.

Diagnosis inconclusive. Please consult a doctor.
true.
```

3. Solve the Monkey and Banana Problem

Implement the monkey and banana problem where the monkey needs to figure out how to get the banana using Prolog's logical reasoning.

Solution:

```
% Initial state and goal state definitions
initial_state(state(at_door, at_window, no, no)).
goal_state(state(_, _, _, yes)).

% Define the moves
move(state(at_door, Box, no, HasBanana), walk(door, window), state(at_window, Box, no, HasBanana)).
move(state(at_window, Box, no, HasBanana), walk(window, door), state(at_door, Box, no, HasBanana)).
move(state(Monkey, at_window, no, HasBanana), push_box(Monkey, window), state(window, window, no, HasBanana)) :-
    Monkey \= at_window.
move(state(Monkey, at_door, no, HasBanana), push_box(Monkey, door), state(door, door, no, HasBanana)) :-
    Monkey \= at_door.
move(state(window, window, no, HasBanana), climb_box, state(window, window, yes, HasBanana)).
move(state(window, window, yes, no), grab_banana, state(window, window, yes, yes)).

% Define the planning mechanism
plan(State, [], _) :-
```

```
   goal_state(State).

plan(State, [Action | RestOfPlan], VisitedStates) :-
   move(State, Action, NewState),
   \+ member(NewState, VisitedStates),
   plan(NewState, RestOfPlan, [NewState | VisitedStates]).

% Solve the problem
solve(Plan) :-
   initial_state(InitialState),
   plan(InitialState, Plan, [InitialState]).
```

Output:

```
?- solve(Plan).
Plan = [push_box(at_door, window), climb_box, grab_banana] .
```

4. 8-Puzzle Problem

Create a Prolog program that solves the 8-puzzle problem using a search strategy like depth-first or breadth-first search.

Solution:

```
% Initial state and goal state definitions
initial_state([[1, 2, 3],
            [4, 5, 6],
            [7, 8, 0]]).

goal_state([[1, 2, 3],
          [4, 5, 6],
          [7, 8, 0]]).

% Define the move mechanism for the puzzle
move(State, NewState) :-
   append(State, [], FlatState),
   select(0, FlatState, TempState),
   length(State, Size),
   index_of(0, FlatState, BlankIndex), (
      (UpIndex is BlankIndex - Size, valid_index(UpIndex, Size),
       replace(TempState, BlankIndex, 0, UpIndex, NewState));
      (DownIndex is BlankIndex + Size, valid_index(DownIndex, Size),
       replace(TempState, BlankIndex, 0, DownIndex, NewState));
      (LeftIndex is BlankIndex - 1, valid_index(LeftIndex, Size),
       replace(TempState, BlankIndex, 0, LeftIndex, NewState));
      (RightIndex is BlankIndex + 1, valid_index(RightIndex, Size),
       replace(TempState, BlankIndex, 0, RightIndex, NewState))
   ).

% Check if an index is valid
valid_index(Index, Size) :-
```

```
    Index >= 0,
    Index < Size.

% Find the index of an element in the list
index_of(X, List, Index) :-
    nth0(Index, List, X).

% Replace an element in the list
replace(List, Old, New, Index, NewList) :-
    nth0(Index, List, Old),
    replace_helper(List, Index, New, NewList).

replace_helper([_|T], 0, New, [New|T]).
replace_helper([H|T], Index, New, [H|R]) :-
    Index > 0,
    NewIndex is Index - 1,
    replace_helper(T, NewIndex, New, R).

% BFS for the 8-puzzle
bfs([[State|Path]|_], Path, State) :-
    goal_state(State).

bfs([Current|Rest], Path, Goal) :-
    Current = [State|Path],
    findall([NewState, State|Path],
            (move(State, NewState),
             \+ member(NewState, [State|Path])), NewStates),
    append(Rest, NewStates, NewQueue),
    bfs(NewQueue, Path, Goal).

% Solve the puzzle
solve :-
    initial_state(InitialState),
    bfs([[InitialState]], Path, GoalState),
    reverse(Path, SolutionPath),
    print_solution(SolutionPath, GoalState).

% Print the solution path
print_solution([], Goal) :-
    print(Goal), nl.

print_solution([State|Rest], Goal) :-
    print(State), nl,
    print_solution(Rest, Goal).
```

Output:

```
?- solve.
[[1,2,3],[4,5,6],[7,8,0]]
true .
```

5. Towers of Hanoi Problem

Write a Prolog program to solve the Towers of Hanoi puzzle with recursive logic.

Solution:

```
move(1, Source, Target, _) :-
format('Move disk from ~w to ~w~n', [Source, Target]).
move(N, Source, Target, Helper) :- N > 1,
M is N - 1,
move(M, Source, Helper, Target), move(1, Source, Target, _), move(M, Helper, Target,
Source).

solve_hanoi(N) :-
format('Solving Towers of Hanoi for ~d disks:~n', [N]),
move(N, source, target, helper).
```

Output:

```
?- solve_hanoi(3).
Solving Towers of Hanoi for 3 disks:
Move disk from source to target
Move disk from source to helper
Move disk from target to helper
Move disk from source to target
Move disk from helper to source
Move disk from helper to target
Move disk from source to target
true .
```

6. Traveling Salesman Problem (TSP)

Implement the Traveling Salesman Problem in Prolog using a brute-force search to find the shortest path.

Solution:

```
% Define distances between cities
distance(a, b, 10).
distance(a, c, 15).
distance(a, d, 20).
distance(b, a, 10).
distance(b, c, 35).
distance(b, d, 25).
distance(c, a, 15).
distance(c, b, 35).
distance(c, d, 30).
distance(d, a, 20).
distance(d, b, 25).
distance(d, c, 30).

% Define path distances
path_distance(X, Y, D) :- distance(X, Y, D).
path_distance(X, Y, D) :- distance(Y, X, D).

% Calculate total distance of a path
path_total_distance([_], 0).
path_total_distance([City1, City2 | Rest], TotalDistance) :-
    path_distance(City1, City2, D),
    path_total_distance([City2 | Rest], RestDistance),
    TotalDistance is D + RestDistance.

% Generate all possible paths starting from a given city
all_paths(Start, Cities, [Start | Path]) :-
    permutation(Cities, Path).

% Traveling Salesman Problem implementation
tsp(Start, Path, MinDistance) :-
    findall(City, (distance(Start, City, _); distance(City, Start, _)), Cities),
    list_to_set(Cities, UniqueCities),  % Remove duplicates
    findall([Start | P], all_paths(Start, UniqueCities, [Start | P]), Paths),
    findall(Distance-Path,
            (member(Path, Paths), path_total_distance(Path, Distance)),
            Distances),
    min_member(MinDistance-Path, Distances).

% Solve the TSP and print the result
solve_tsp(Start) :-
    tsp(Start, Path, MinDistance),
    format('The shortest path is: ~w~n', [Path]),
    format('With a total distance of: ~w~n', [MinDistance]).
```

Output:

```
?- solve_tsp(a).
The shortest path is: [a,b,d,c]
With a total distance of: 65
true.
```

7. Missionaries and Cannibals Problem

Solve the missionaries and cannibals problem in Prolog by ensuring that missionaries are never outnumbered by cannibals.

Solution:

```
% Define the start and goal states
start([3, 3, left, 0, 0]).
goal([0, 0, right, 3, 3]).

% Check if the state is legal
legal(CL, ML, CR, MR) :-
  ML >= 0, CL >= 0, MR >= 0, CR >= 0,
  (ML >= CL; ML = 0),   % Missionaries on the left must not be less than cannibals
  (MR >= CR; MR = 0).   % Missionaries on the right must not be less than cannibals

% Define possible moves from left to right
move([CL, ML, left, CR, MR], [CL, ML2, right, CR, MR2]) :-
  ML2 is ML - 2,
  MR2 is MR + 2,
  legal(CL, ML2, CR, MR2).

move([CL, ML, left, CR, MR], [CL2, ML, right, CR2, MR]) :-
  CL2 is CL - 2,
  CR2 is CR + 2,
  legal(CL2, ML, CR2, MR).

move([CL, ML, left, CR, MR], [CL2, ML2, right, CR2, MR2]) :-
  CL2 is CL - 1,
  ML2 is ML - 1,
  CR2 is CR + 1,
  MR2 is MR + 1,
  legal(CL2, ML2, CR2, MR2).

move([CL, ML, left, CR, MR], [CL, ML2, right, CR, MR2]) :-
  ML2 is ML - 1,
  MR2 is MR + 1,
  legal(CL, ML2, CR, MR2).

move([CL, ML, left, CR, MR], [CL2, ML, right, CR2, MR]) :-
  CL2 is CL - 1,
  CR2 is CR + 1,
  legal(CL2, ML, CR2, MR).
```

```
% Define possible moves from right to left
move([CL, ML, right, CR, MR], [CL, ML2, left, CR, MR2]) :-
   ML2 is ML + 2,
   MR2 is MR - 2,
   legal(CL, ML2, CR, MR2).

move([CL, ML, right, CR, MR], [CL2, ML, left, CR2, MR]) :-
   CR2 is CR - 2,
   CL2 is CL + 2,
   legal(CL2, ML, CR2, MR).

move([CL, ML, right, CR, MR], [CL2, ML2, left, CR2, MR2]) :-
   CL2 is CL + 1,
   ML2 is ML + 1,
   CR2 is CR - 1,
   MR2 is MR - 1,
   legal(CL2, ML2, CR2, MR2).

move([CL, ML, right, CR, MR], [CL, ML2, left, CR, MR2]) :-
   ML2 is ML + 1,
   MR2 is MR - 1,
   legal(CL, ML2, CR, MR2).

move([CL, ML, right, CR, MR], [CL2, ML, left, CR2, MR]) :-
   CL2 is CL + 1,
   CR2 is CR - 1,
   legal(CL2, ML, CR2, MR).

% Define the path finding logic
path([CL1, ML1, B1, CR1, MR1], [CL2, ML2, B2, CR2, MR2], Explored, MovesList) :-
   move([CL1, ML1, B1, CR1, MR1], [CL3, ML3, B3, CR3, MR3]),
   \+ member([CL3, ML3, B3, CR3, MR3], Explored),
   path([CL3, ML3, B3, CR3, MR3], [CL2, ML2, B2, CR2, MR2],
      [[CL3, ML3, B3, CR3, MR3] | Explored],
      [[[CL3, ML3, B3, CR3, MR3], [CL1, ML1, B1, CR1, MR1]] | MovesList]).

% Base case for path finding
path([CL, ML, B, CR, MR], [CL, ML, B, CR, MR], _, MovesList) :-
   output(MovesList).

% Output the moves
output([]) :- nl.
output([[A, B] | MovesList]) :-
   output(MovesList),
   write(B), write(' -> '), write(A), nl.

% Find the solution
find :-
   path([3, 3, left, 0, 0], [0, 0, right, 3, 3], [[3, 3, left, 0, 0]], _).
```

Output:

```
?- find.

[3,3,left,0,0] -> [1,3,right,2,0]
[1,3,right,2,0] -> [2,3,left,1,0]
[2,3,left,1,0] -> [0,3,right,3,0]
[0,3,right,3,0] -> [1,3,left,2,0]
[1,3,left,2,0] -> [1,1,right,2,2]
[1,1,right,2,2] -> [2,2,left,1,1]
[2,2,left,1,1] -> [2,0,right,1,3]
[2,0,right,1,3] -> [3,0,left,0,3]
[3,0,left,0,3] -> [1,0,right,2,3]
[1,0,right,2,3] -> [1,1,left,2,2]
[1,1,left,2,2] -> [0,0,right,3,3]
true .
```

8. N-Queens Problem

Create a Prolog program to place N queens on an N×N chessboard such that no two queens threaten each other.

Solution:

```
n_queens(N, Solution) :-
    length(Solution, N),
    place_queens(Solution, N),
    safe(Solution).

place_queens([], _).
place_queens([Row | Others], N) :-
    place_queens(Others, N),
    between(1, N, Row),
    \+ threatened(Row, Others, 1).

threatened(Row, [Row1 | Others], Dist) :-
    Row =:= Row1;
    Row =:= Row1 + Dist;
    Row =:= Row1 - Dist,
    Dist1 is Dist + 1,
    threatened(Row, Others, Dist1).

safe([]).
safe([Row | Others]) :-
    safe(Others),
    \+ threatened(Row, Others, 1).

print_solution([]).
print_solution([Q | Rest]) :-
    print_row(Q),
    print_solution(Rest).

print_row(Q) :-
    length(Row, Q),
    maplist(=(.), Row),
```

```
        append(Row, ['Q'], DisplayRow),
        writeln(DisplayRow).
```

Output:

```
?- n_queens(4, Solution), print_solution(Solution).
[.,.,.,Q]
[.,Q]
[.,.,.,.,Q]
[.,.,Q]
Solution = [3, 1, 4, 2] .
```

9. Water Jug Problem

Write a Prolog program to solve the water jug problem where two jugs with different capacities must measure a specific amount of water.

Solution:

```
solve_water_jug(X, Y, Goal, Solution) :-
   empty_state(InitialState),
   search(InitialState, Goal, X, Y, [InitialState], Solution).
empty_state(state(0, 0)).
search(state(Goal, _), Goal, _, _, Path, Path).
search(state(_, Goal), Goal, _, _, Path, Path).
search(CurrentState, Goal, X, Y, Visited, Solution) :-
   action(CurrentState, NextState, X, Y),
   \+ member(NextState, Visited),
   search(NextState, Goal, X, Y, [NextState | Visited], Solution).
action(state(_, Y2), state(X, Y2), X, _).
action(state(X1, _), state(X1, Y), _, Y).
action(state(X, Y2), state(0, Y2), _, _).
action(state(X1, Y), state(X1, 0), _, _).
action(state(X1, Y2), state(NewX1, NewY2), X, Y) :-
   Total is X1 + Y2,
   (Total =< Y -> NewX1 = 0, NewY2 = Total
   ; NewX1 is Total - Y, NewY2 = Y).
action(state(X1, Y2), state(NewX1, NewY2), X, Y) :-
   Total is X1 + Y2,
   (Total =< X -> NewX1 = Total, NewY2 = 0
```

```
    ; NewX1 = X, NewY2 is Total - X).
print_solution([]).
print_solution([state(Jug1, Jug2) | Rest]) :-
    format('Jug1: ~w, Jug2: ~w~n', [Jug1, Jug2]),
    print_solution(Rest).
```

<u>Output:</u>

```
?- solve_water_jug(4, 3, 2, Solution), print_solution(Solution).
Jug1: 4, Jug2: 2
Jug1: 3, Jug2: 3
Jug1: 3, Jug2: 0
Jug1: 0, Jug2: 3
Jug1: 4, Jug2: 3
Jug1: 4, Jug2: 0
Jug1: 0, Jug2: 0
Solution = [state(4, 2), state(3, 3), state(3, 0), state(0, 3), state(4, 3), state(4, 0), state(0, 0)]
```

10. Prolog Chatbot

Develop a simple Prolog-based chatbot that can respond to user queries with predefined rule-based responses.

<u>Solution:</u>

```
chatbot :-
    write('Hello! I am a simple Prolog chatbot. How can I help you today?'), nl,
    repeat,
    write('> '),
    read_line_to_string(user_input, Input),
    respond(Input),
    (Input == "quit" -> ! ; fail).
 respond("hello") :-
    write('Hello! How are you?'), nl.
respond("hi") :-
    write('Hi there! How can I assist you today?'), nl.
respond("how are you") :-
    write('I am just a program, but thanks for asking! How can I help you?'), nl.
respond("what is prolog") :-
    write('Prolog is a logic programming language widely used in AI and computational
linguistics.'), nl.
respond("why use prolog") :-
    write('Prolog is useful for tasks that involve symbolic reasoning and rule-based queries.'),
nl.
respond("what can you do") :-
```

```
    write('I can answer basic questions about Prolog and respond to simple greetings. Ask me anything!'), nl.

respond("help") :-

    write('You can ask me about Prolog, say "hello" or "hi", or inquire about my capabilities.'), nl.

respond("goodbye") :-

    write('Goodbye! Have a wonderful day!'), nl.

respond("quit") :-

    write('Exiting the chatbot. Take care!'), nl.

respond(_) :-

    write('I am sorry, I do not understand. Could you try rephrasing?'), nl.
```

<u>Output:</u>

```
?- chatbot.
Hello! I am a simple Prolog chatbot. How can I help you today?
> hello
Hello! How are you?
> what is prolog
Prolog is a logic programming language widely used in AI and computational linguistics.
> goodbye
Goodbye! Have a wonderful day!
> quit
Exiting the chatbot. Take care!
true.
```

11. <u>Arithmetic Puzzle Solver</u>

Create a Prolog program that can solve puzzles like Sudoku, magic squares, or other arithmetic puzzles using constraint logic programming (CLP).

<u>Solution:</u>

```
:- use_module(library(clpfd)).

% Sudoku Solver

sudoku(Puzzle) :-

    length(Puzzle, 9),

    maplist(same_length(Puzzle), Puzzle),

    append(Puzzle, Vars),

    Vars ins 1..9,

    maplist(all_distinct, Puzzle),

    transpose(Puzzle, Columns),

    maplist(all_distinct, Columns),

    boxes(Puzzle),

    label(Vars).

boxes([]).
```

```
  boxes([A, B, C | Rest]) :-
      A = [A1, A2, A3 | _],
      B = [B1, B2, B3 | _],
      C = [C1, C2, C3 | _],
      all_distinct([A1, A2, A3, B1, B2, B3, C1, C2, C3]),
      boxes(Rest).
example_sudoku(Puzzle) :-
      Puzzle = [
          [5, 3, 0, 0, 7, 0, 0, 0, 0],
          [6, 0, 0, 1, 9, 5, 0, 0, 0],
          [0, 9, 8, 0, 0, 0, 0, 6, 0],
          [8, 0, 0, 0, 6, 0, 0, 0, 3],
          [4, 0, 0, 8, 0, 3, 0, 0, 1],
          [7, 0, 0, 0, 2, 0, 0, 0, 6],
          [0, 6, 0, 0, 0, 0, 2, 8, 0],
          [0, 0, 0, 4, 1, 9, 0, 0, 5],
          [0, 0, 0, 0, 8, 0, 0, 7, 9]
      ].
% Magic Square Solver
  magic_square(N, Square) :-
      length(Square, N),
      maplist(length_(N), Square),
      append(Square, Vars),
      Vars ins 1..(N*N),
      all_distinct(Vars),
      Rows = Square,
      transpose(Rows, Columns),
      maplist(all_distinct, Columns),
      diagonals(Rows, Diags),
      maplist(all_distinct, Diags),
      Total is N * (N*N + 1) // 2,
      maplist(sum_eq(Total), Rows),
      maplist(sum_eq(Total), Columns),
```

```
    maplist(sum_eq(Total), Diags),
    label(Vars).
length_(Length, List) :- length(List, Length).
diagonals(Rows, [D1, D2]) :-
    findall(E, (nth1(I, Rows, Row), nth1(I, Row, E)), D1),
    findall(E, (nth1(I, Rows, Row), N is length(Rows) - I + 1, nth1(N, Row, E)), D2).
sum_eq(Sum, List) :- sum(List, #=, Sum).
example_magic_square(Square) :- Square = [
    [8, 1, 6],
    [3, 5, 7],
    [4, 9, 2]
 ]
```

Output

```
?- example_sudoku(Puzzle).
Puzzle = [[5, 3, 0, 0, 7, 0, 0, 0|...], [6, 0, 0, 1, 9, 5, 0|...], [0, 9, 8, 0, 0, 0|...], [8, 0, 0, 0, 6|...], [4, 0, 0, 8|...], [7, 0, 0|...], [0, 6|...]
, [0|...], [...|...]].

?- sudoku(Puzzle).
Puzzle = [[1, 2, 3, 4, 5, 6, 7, 8|...], [4, 5, 6, 1, 2, 3, 8|...], [7, 8, 9, 2, 1, 4|...], [2, 1, 4, 3, 6|...], [3, 6, 5, 7|...], [8, 9, 7|...], [5, 3|...]
, [6|...], [...|...]] .

?- maplist(writeln, Puzzle).
Puzzle = [] .
```

12. Logic Circuit Design

Write a Prolog program to simulate a basic logic circuit (e.g., AND, OR, NOT gates) and verify its truth table.

Solution:

```
% AND Gate Truth Table

and_gate(0, 0, 0).

and_gate(0, 1, 0).

and_gate(1, 0, 0).

and_gate(1, 1, 1).

% OR Gate Truth Table

or_gate(0, 0, 0).

or_gate(0, 1, 1).

or_gate(1, 0, 1).

or_gate(1, 1, 1).

% NOT Gate Truth Table

not_gate(0, 1).

not_gate(1, 0).

% Verify AND Gate Truth Table

verify_and_truth_table :-

   and_gate(A, B, Output),

   format('AND(~w, ~w) = ~w~n', [A, B, Output]),

   fail.

verify_and_truth_table. % To terminate after all calls

% Verify OR Gate Truth Table

verify_or_truth_table :-

   or_gate(A, B, Output),

   format('OR(~w, ~w) = ~w~n', [A, B, Output]),

   fail.

verify_or_truth_table. % To terminate after all calls
```

```
% Verify NOT Gate Truth Table
verify_not_truth_table :-
    not_gate(A, Output),
    format('NOT(~w) = ~w~n', [A, Output]),
    fail.
verify_not_truth_table. % To terminate after all calls


% Verify all Truth Tables
verify_all :-
    writeln('--- AND Gate Truth Table ---'),
    verify_and_truth_table,
    nl,
    writeln('--- OR Gate Truth Table ---'),
    verify_or_truth_table,
    nl,
    writeln('--- NOT Gate Truth Table ---'),
    verify_not_truth_table.
```

<u>Output:</u>

```
?- verify_all.
--- AND Gate Truth Table ---
AND(0,0) = 0
AND(0,1) = 0
AND(1,0) = 0
AND(1,1) = 1

--- OR Gate Truth Table ---
OR(0,0) = 0
OR(0,1) = 1
OR(1,0) = 1
OR(1,1) = 1

--- NOT Gate Truth Table ---
NOT(0) = 1
NOT(1) = 0
true .
```

13. Route Finding System

Create a Prolog system to find routes between cities (nodes) using depth-first or breadth-first search algorithms.

Solution:

```
% Define the connections between cities
connected(cityA, cityB).
connected(cityB, cityC).
connected(cityA, cityD).
connected(cityD, cityE).
connected(cityC, cityF).
connected(cityE, cityF).
connected(cityF, cityG).

% Define the path predicate
path(X, Y) :- connected(X, Y).
path(Y, X) :- connected(X, Y).  % Bi-directional paths

% Depth-First Search (DFS)
dfs(Start, Goal, Path) :-
    dfs_recursive(Start, Goal, [Start], Path).

dfs_recursive(Goal, Goal, Visited, Path) :-
    reverse(Visited, Path).  % Reverse to get the correct path order
dfs_recursive(Start, Goal, Visited, Path) :-
    path(Start, Next),
    \+ member(Next, Visited),  % Avoid cycles
    dfs_recursive(Next, Goal, [Next|Visited], Path).

% Breadth-First Search (BFS)
bfs(Start, Goal, Path) :-
```

```
  bfs_queue([[Start]], Goal, Path).

bfs_queue([[Goal|Rest]|_], Goal, Path) :-
  reverse([Goal|Rest], Path).  % Reverse to get the correct path order
bfs_queue([CurrentPath|Paths], Goal, Path) :-
  extend_path(CurrentPath, NewPaths),
  append(Paths, NewPaths, UpdatedPaths),  % Update the queue
  bfs_queue(UpdatedPaths, Goal, Path).

extend_path([Node|RestOfPath], NewPaths) :-
  findall([Next, Node|RestOfPath],
      (path(Node, Next), \+ member(Next, [Node|RestOfPath])),
      NewPaths).
```

Output:

```
?- dfs(cityA, cityG, Path).
Path = [cityA, cityB, cityC, cityF, cityG] .

?- bfs(cityA, cityG, Path).
Path = [cityA, cityB, cityC, cityF, cityG] .
```

14. Natural Language Processing (NLP)

Implement a simple natural language processor in Prolog that can parse and understand basic English sentences, focusing on grammar and syntax.

Solution:

```
% Define articles
article(a).
article(the).

% Define nouns
noun(cat).
noun(dog).
noun(man).
noun(woman).
```

```
noun(ball).
noun(apple).

% Define verbs
verb(eats).
verb(chases).
verb(sees).
verb(likes).

% Define a sentence structure
sentence(S) :-
   noun_phrase(NP, S, S1),  % Get a noun phrase
   verb_phrase(VP, S1, []),  % Get a verb phrase
   append(NP, VP, S).  % Combine both phrases to form a sentence

% Define a noun phrase structure
noun_phrase([Article, Noun], [Article, Noun | Rest], Rest) :-
   article(Article),
   noun(Noun).  % Article followed by a noun

% Define a verb phrase structure
verb_phrase([Verb | NP], [Verb | Rest], Remaining) :-
   verb(Verb),
   noun_phrase(NP, Rest, Remaining).  % Verb followed by a noun phrase

% Parse a sentence and print if it's correct
parse(Sentence) :-
   sentence(Sentence),
   write('The sentence is grammatically correct.'),
   !.
```

```
% Check the sentence and print if it's correct or incorrect
parse_sentence(Sentence) :-
    ( parse(Sentence) -> true
    ; write('The sentence is grammatically incorrect.') ).
```

Output:

```
?- parse_sentence([the, cat, chases, the, dog]).
The sentence is grammatically correct.
true.

?- parse_sentence([cat, the, chases]).
The sentence is grammatically incorrect.
true.
```