SRM Institute of Science and Technology

Delhi – Meerut Road, Sikri Kalan, Ghaziabad, Uttar Pradesh – 201204

Department of Computer Applications

Circular – 2023-24

B.Sc. CS 5th Sem

ARTIFICIAL INTELLIGENCE (UCS23D01J)- Lab Manual

This manual provides the structure for each program in the Artificial Intelligence lab, including its title, aim, procedure, a placeholder for source code, input examples, and expected output.

Lab 1: Program for solving a water jug problem.

Title: Water Jug Problem

Aim: To implement a program to solve the Water Jug Problem using a suitable search algorithm.

Procedure:

1. **Define State Space:** Represent the state of the two jugs as a tuple `(amount_in_jug1, amount_in_jug2)`.
2. **Define Operations:** List all possible actions:
   - Fill Jug 1 (e.g., `(x, y) -> (Cap1, y)`)
   - Fill Jug 2 (e.g., `(x, y) -> (x, Cap2)`)
   - Empty Jug 1 (e.g., `(x, y) -> (0, y)`)
   - Empty Jug 2 (e.g., `(x, y) -> (x, 0)`)
   - Pour Jug 1 to Jug 2 (e.g., `(x, y) -> (x - delta, y + delta)`)
   - Pour Jug 2 to Jug 1 (e.g., `(x, y) -> (x + delta, y - delta)`)
3. **Choose Search Algorithm:** Select a search algorithm (e.g., Breadth-First Search or Depth-First Search) to explore the state space.
4. **Implement Search:** Write code to traverse states, keeping track of visited states to avoid cycles.
5. **Goal Test:** Define the goal condition (e.g., one jug contains a specific target amount).
6. **Path Reconstruction:** If the goal is found, reconstruct the sequence of operations that led to it.

Source Code:

```
# Placeholder for Python code to solve the Water Jug Problem
# (e.g., using BFS or DFS)

def solve_water_jug(jug1_capacity, jug2_capacity, target_amount):
    # Your implementation here
    # Example:
    # queue = [(0, 0, [])] # (jug1, jug2, path)
    # visited = set()
    # while queue:
    #     current_jug1, current_jug2, path = queue.pop(0) # For BFS
    #     if (current_jug1, current_jug2) in visited:
    #         continue
    #     visited.add((current_jug1, current_jug2))
    #     if current_jug1 == target_amount or current_jug2 == target_amount:
```

```
    #         return path + [(current_jug1, current_jug2)]
    #     # Generate next states and add to queue/stack
    # return "No solution found"

# Example usage:
# path = solve_water_jug(4, 3, 2)
# print(path)
```

Input:

```
Jug 1 Capacity: 4 liters
Jug 2 Capacity: 3 liters
Target Amount: 2 liters
```

Expected Output:

```
Initial State: (0, 0)
Fill Jug 1: (4, 0)
Pour Jug 1 to Jug 2: (1, 3)
Empty Jug 2: (1, 0)
Pour Jug 1 to Jug 2: (0, 1)
Fill Jug 1: (4, 1)
Pour Jug 1 to Jug 2: (2, 3)
Goal Reached: (2, 3)
```

Lab 2: Program for solving a water jug problem using Depth first search

Title: Water Jug Problem using Depth-First Search (DFS)

Aim: To implement a program to solve the Water Jug Problem specifically using the Depth-First Search (DFS) algorithm.

Procedure:

1. **Define State Space and Operations:** As in Lab 1.
2. **Implement DFS:** Use a recursive function or an explicit stack to implement DFS.
   - Start from the initial state.
   - Explore as far as possible along each branch before backtracking.
   - Maintain a set of `visited` states to avoid infinite loops and redundant computations.
   - When a new state is generated, add it to the stack if not visited.
3. **Goal Test:** Check if the current state satisfies the target amount.
4. **Path Reconstruction:** Store the path taken to reach the current state.

Source Code:

```
# Placeholder for Python code to solve the Water Jug Problem using DFS

def dfs_water_jug(jug1_capacity, jug2_capacity, target_amount):
    # Your DFS implementation here
    # Example:
    # stack = [(0, 0, [])] # (jug1, jug2, path)
    # visited = set()
    # while stack:
    #     current_jug1, current_jug2, path = stack.pop() # For DFS (LIFO)
    #     if (current_jug1, current_jug2) in visited:
    #         continue
    #     visited.add((current_jug1, current_jug2))
    #     if current_jug1 == target_amount or current_jug2 == target_amount:
    #         return path + [(current_jug1, current_jug2)]
    #     # Generate next states and push to stack (in reverse order for
specific path)
    # return "No solution found"

# Example usage:
# path = dfs_water_jug(4, 3, 2)
# print(path)
```

Input:

```
Jug 1 Capacity: 4 liters
Jug 2 Capacity: 3 liters
Target Amount: 2 liters
```

Expected Output:

```
A sequence of steps found by DFS to reach the target amount.
(Note: DFS does not guarantee the shortest path)
Example:
Initial State: (0, 0)
Fill Jug 1: (4, 0)
Pour Jug 1 to Jug 2: (1, 3)
Empty Jug 2: (1, 0)
```

```
Fill Jug 1: (4, 0) # Backtrack and explore another path
...
Goal Reached: (2, 3)
```

Lab 3: Program for solving a water jug problem using Breadth first search

Title: Water Jug Problem using Breadth-First Search (BFS)

Aim: To implement a program to solve the Water Jug Problem specifically using the Breadth-First Search (BFS) algorithm.

Procedure:

1. **Define State Space and Operations:** As in Lab 1.
2. **Implement BFS:** Use a queue to implement BFS.
   - Start from the initial state and add it to the queue.
   - Explore all neighbor nodes at the current depth level before moving to the next depth level.
   - Maintain a set of `visited` states to avoid redundant computations.
   - When a new state is generated, add it to the queue if not visited.
3. **Goal Test:** Check if the current state satisfies the target amount.
4. **Path Reconstruction:** Store the path taken to reach the current state. BFS guarantees the shortest path in terms of number of steps.

Source Code:

```
# Placeholder for Python code to solve the Water Jug Problem using BFS

from collections import deque

def bfs_water_jug(jug1_capacity, jug2_capacity, target_amount):
    # Your BFS implementation here
    # Example:
    # queue = deque([(0, 0, [])]) # (jug1, jug2, path)
    # visited = set()
    # while queue:
    #     current_jug1, current_jug2, path = queue.popleft() # For BFS (FIFO)
    #     if (current_jug1, current_jug2) in visited:
    #         continue
    #     visited.add((current_jug1, current_jug2))
    #     if current_jug1 == target_amount or current_jug2 == target_amount:
    #         return path + [(current_jug1, current_jug2)]
    #     # Generate next states and append to queue
    # return "No solution found"

# Example usage:
# path = bfs_water_jug(4, 3, 2)
# print(path)
```

Input:

```
Jug 1 Capacity: 4 liters
Jug 2 Capacity: 3 liters
Target Amount: 2 liters
```

Expected Output:

```
A shortest sequence of steps found by BFS to reach the target amount.
Example:
Initial State: (0, 0)
Fill Jug 1: (4, 0)
```

```
Pour Jug 1 to Jug 2: (1, 3)
Empty Jug 2: (1, 0)
Pour Jug 1 to Jug 2: (0, 1)
Fill Jug 1: (4, 1)
Pour Jug 1 to Jug 2: (2, 3)
Goal Reached: (2, 3)
```

Lab 4: Program to find out route distance between two cities

Title: Route Distance Calculation between Two Cities

Aim: To implement a program that calculates the shortest route distance between two given cities using a graph-based approach (e.g., Dijkstra's algorithm).

Procedure:

1. **Represent Graph:** Model cities as nodes and roads as edges in a graph. Assign weights to edges representing distances.
   - Use an adjacency list or adjacency matrix to store the graph.
2. **Choose Algorithm:** Select a shortest path algorithm (e.g., Dijkstra's algorithm or A* search).
3. **Implement Algorithm:**
   - Initialize distances to all nodes as infinity, except for the start node (0).
   - Use a priority queue to efficiently select the unvisited node with the smallest known distance.
   - Relax edges: Update the distance to a neighbor if a shorter path is found through the current node.
4. **User Input:** Prompt the user to enter the starting and destination cities.
5. **Display Result:** Print the shortest distance and the path (sequence of cities) from the start to the destination.

Source Code:

```
# Placeholder for Python code to find route distance using Dijkstra's
algorithm

import heapq

def dijkstra(graph, start_node, end_node):
    # Your Dijkstra's implementation here
    # Example:
    # distances = {node: float('infinity') for node in graph}
    # distances[start_node] = 0
    # priority_queue = [(0, start_node)] # (distance, node)
    # predecessors = {} # To reconstruct path
    #
    # while priority_queue:
    #     current_distance, current_node = heapq.heappop(priority_queue)
    #
    #     if current_distance > distances[current_node]:
    #         continue
    #
    #     for neighbor, weight in graph[current_node].items():
    #         distance = current_distance + weight
    #         if distance < distances[neighbor]:
    #             distances[neighbor] = distance
    #             predecessors[neighbor] = current_node
    #             heapq.heappush(priority_queue, (distance, neighbor))
    #
    # # Reconstruct path
    # path = []
    # current = end_node
    # while current is not None:
    #     path.insert(0, current)
    #     current = predecessors.get(current)
    # if path[0] != start_node: # No path found
    #     return float('infinity'), []
```

```
    # return distances[end_node], path

# Example usage:
# graph = {
#     'A': {'B': 1, 'C': 4},
#     'B': {'A': 1, 'C': 2, 'D': 5},
#     'C': {'A': 4, 'B': 2, 'D': 1},
#     'D': {'B': 5, 'C': 1}
# }
# distance, path = dijkstra(graph, 'A', 'D')
# print(f"Shortest distance: {distance}, Path: {path}")
```

Input:

```
Graph Definition:
Cities: A, B, C, D
Roads (and distances):
A-B: 10 km
A-C: 15 km
B-C: 5 km
B-D: 20 km
C-D: 8 km

Start City: A
Destination City: D
```

Expected Output:

```
Shortest distance from A to D: 23 km
Path: A -> C -> D
```

Lab 5: Program for Tic Tac Toe game played by Single player against automated Computer player

Title: Tic-Tac-Toe Game (Single Player vs. Computer)

Aim: To develop a single-player Tic-Tac-Toe game where a human player competes against an automated computer opponent, implementing a basic AI strategy.

Procedure:

1. **Game Board:** Create a 3x3 grid to represent the Tic-Tac-Toe board.
2. **Game State:** Maintain the current state of the board, tracking 'X', 'O', and empty cells.
3. **Player Turns:** Implement a loop for alternating turns between the human player and the computer.
4. **Human Player Move:**
   - Prompt the human player for their move (e.g., row and column).
   - Validate the move (ensure cell is empty and within bounds).
   - Update the board.
5. **Computer Player AI:**
   - Implement a simple AI strategy for the computer (e.g., Minimax algorithm for optimal play, or a rule-based approach):
     - Check for immediate winning moves.
     - Check for immediate blocking moves (prevent human from winning).
     - Prioritize center square.
     - Prioritize corner squares.
     - Choose any available edge square.
   - Update the board with the computer's move.
6. **Win/Draw Condition:** After each move, check if the game has ended (win for 'X', win for 'O', or a draw).
7. **Display Board:** Render the current state of the board after each move.

Source Code:

```
# Placeholder for Python code for Tic-Tac-Toe (Single Player vs. Computer)

def print_board(board):
    # Your board printing logic
    pass

def check_win(board, player):
    # Your win checking logic
    pass

def get_computer_move(board):
    # Your AI logic (e.g., simple rules or minimax)
    pass

def play_game():
    # Your game loop
    # board = [[' ' for _ in range(3)] for _ in range(3)]
    # current_player = 'X' # Human starts
    # while True:
    #     print_board(board)
    #     if current_player == 'X':
    #         # Get human move
    #         pass
    #     else: # Computer's turn
    #         # Get computer move
```

```
    #         pass
    #     # Check win/draw
    #     # Switch player
    pass

# play_game()
```

Input:

```
(Human player's moves, e.g., entering row and column numbers)
Human Player (X) chooses: 1 1
Computer Player (O) chooses: 0 0
Human Player (X) chooses: 2 2
Computer Player (O) chooses: 0 1
...
```

Expected Output:

```
(Visual representation of the board after each move, and the final game
result)

Initial Board:
  |   |
--+---+--
  |   |
--+---+--
  |   |

Human (X) moves to (1,1):
  |   |
--+---+--
  | X |
--+---+--
  |   |

Computer (O) moves to (0,0):
O |   |
--+---+--
  | X |
--+---+--
  |   |

...

Final Board:
O | X | O
--+---+--
X | X | O
--+---+--
O |   | X

X wins!
```

Lab 6: Program for Tic Tac Toe game played by two different human players.

Title: Tic-Tac-Toe Game (Two Human Players)

Aim: To develop a two-player Tic-Tac-Toe game where two human players can compete against each other.

Procedure:

1. **Game Board:** Create a 3x3 grid to represent the Tic-Tac-Toe board.
2. **Game State:** Maintain the current state of the board, tracking 'X', 'O', and empty cells.
3. **Player Turns:** Implement a loop for alternating turns between Player 1 ('X') and Player 2 ('O').
4. **Player Move:**
   - Prompt the current player for their move (e.g., row and column).
   - Validate the move (ensure cell is empty and within bounds).
   - Update the board.
5. **Win/Draw Condition:** After each move, check if the game has ended (win for 'X', win for 'O', or a draw).
6. **Display Board:** Render the current state of the board after each move.

Source Code:

```
# Placeholder for Python code for Tic-Tac-Toe (Two Human Players)

def print_board(board):
    # Your board printing logic
    pass

def check_win(board, player):
    # Your win checking logic
    pass

def play_two_player_game():
    # Your game loop
    # board = [[' ' for _ in range(3)] for _ in range(3)]
    # current_player = 'X'
    # while True:
    #     print_board(board)
    #     print(f"Player {current_player}'s turn.")
    #     # Get player move
    #     # Validate and update board
    #     # Check win/draw
    #     # Switch player
    pass

# play_two_player_game()
```

Input:

```
(Player X's moves, then Player O's moves)
Player X chooses: 0 0
Player O chooses: 1 1
Player X chooses: 0 1
Player O chooses: 2 2
Player X chooses: 0 2
```

Expected Output:

```
(Visual representation of the board after each move, and the final game
result)

Initial Board:
  |   |
--+---+--
  |   |
--+---+--
  |   |

Player X moves to (0,0):
X |   |
--+---+--
  |   |
--+---+--
  |   |

Player O moves to (1,1):
X |   |
--+---+--
  | O |
--+---+--
  |   |

...

Final Board:
X | X | X
--+---+--
O | O |
--+---+--
  |   |

Player X wins!
```

Lab 7: Program to implement Tower of Hanoi

Title: Tower of Hanoi

Aim: To implement a program that solves the Tower of Hanoi puzzle for a given number of disks using recursion.

Procedure:

1. **Understand Rules:** Recall the rules:
   - Only one disk can be moved at a time.
   - Each move consists of taking the upper disk from one of the stacks and placing it on top of another stack or on an empty rod.
   - No larger disk may be placed on top of a smaller disk.
2. **Recursive Logic:** Define a recursive function `hanoi(n, source, auxiliary, destination)`:
   - **Base Case:** If `n = 1` (only one disk), move it directly from `source` to `destination`.
   - **Recursive Step:**
     - Move `n-1` disks from `source` to `auxiliary` using `destination` as temporary.
     - Move the `n`th disk from `source` to `destination`.
     - Move `n-1` disks from `auxiliary` to `destination` using `source` as temporary.
3. **Print Moves:** In each move step, print the action taken (e.g., "Move disk 1 from A to C").

Source Code:

```
# Placeholder for Python code to solve Tower of Hanoi recursively

def tower_of_hanoi(n, source, auxiliary, destination):
    # Your recursive implementation here
    # Example:
    # if n == 1:
    #     print(f"Move disk 1 from {source} to {destination}")
    #     return
    # tower_of_hanoi(n - 1, source, destination, auxiliary)
    # print(f"Move disk {n} from {source} to {destination}")
    # tower_of_hanoi(n - 1, auxiliary, source, destination)

# Example usage:
# tower_of_hanoi(3, 'A', 'B', 'C')
```

Input:

```
Number of disks: 3
```

Expected Output:

```
Move disk 1 from A to C
Move disk 2 from A to B
Move disk 1 from C to B
Move disk 3 from A to C
Move disk 1 from B to A
Move disk 2 from B to C
```

```
Move disk 1 from A to C
```

Lab 8: Program for building a magic square of Odd number of Rows and columns.

Title: Magic Square Generation (Odd Order)

Aim: To implement a program that generates a magic square for a given odd number of rows and columns.

Procedure:

1. **Understand Magic Square:** A magic square is a square grid where the sum of numbers in each row, each column, and both main diagonals is the same (the "magic constant").
2. **Siamese Method (De la Loubère):** This method is commonly used for odd-ordered magic squares.
   - Start by placing `1` in the middle cell of the top row (i.e., `[0][n//2]`).
   - For subsequent numbers (`k = 2, 3, ..., n*n`):
     - Move diagonally up-right from the current position `(row, col)` to `(row-1, col+1)`.
     - **Boundary Conditions:**
       - If `row` becomes `-1`, wrap around to `n-1`.
       - If `col` becomes `n`, wrap around to `0`.
     - **Occupied Cell/Top-Right Corner:** If the new position is already occupied, or if the diagonal move goes out of bounds at the top-right corner, move directly down one cell from the *original* position (`(row+1, col)`) instead.
     - Place the number `k` in the determined cell.
3. **Print Square:** Display the generated magic square.

Source Code:

```
# Placeholder for Python code to generate an odd-ordered magic square

def generate_odd_magic_square(n):
    # Your implementation of the Siamese method
    # Example:
    # magic_square = [[0 for _ in range(n)] for _ in range(n)]
    # i, j = 0, n // 2
    # num = 1
    # while num <= n * n:
    #     if i < 0 and j == n: # Wrap around top-right corner
    #         i = n - 1
    #         j = n - 1
    #     elif i < 0: # Wrap around top
    #         i = n - 1
    #     elif j == n: # Wrap around right
    #         j = 0
    #
    #     if magic_square[i][j] != 0: # Cell occupied, move down
    #         i += 2
    #         j -= 1 # Revert last diagonal move
    #         if i >= n: # Handle wrap around for i
    #             i -= n
    #         if j < 0: # Handle wrap around for j
    #             j += n
    #
    #     magic_square[i][j] = num
    #     num += 1
    #     i -= 1
```

```
    #     j += 1
    # return magic_square

# Example usage:
# square = generate_odd_magic_square(3)
# for row in square:
#     print(row)
```

Input:

```
Order of Magic Square (n): 3
```

Expected Output:

```
8 1 6
3 5 7
4 9 2
```

Lab 9: Program for building a magic square of Even number of Rows and columns

Title: Magic Square Generation (Even Order)

Aim: To implement a program that generates a magic square for a given even number of rows and columns.

Procedure:

1. **Understand Magic Square:** As in Lab 8.
2. **Even Order Methods:** Even-ordered magic squares are more complex.
   - **Doubly Even Order (**n=4k**):**
     - Fill the square naturally from 1 to n2.
     - Identify "unchanged" cells (e.g., a central square of size n/2×n/2 and four corner n/4×n/4 squares).
     - For the "changed" cells, replace the number `x` with `n*n + 1 - x`.
   - **Singly Even Order (**n=4k+2**):** This involves dividing the square into four quadrants, filling them using a modified Siamese method, and then swapping elements between specific quadrants. This is significantly more involved.
3. **Print Square:** Display the generated magic square.

Source Code:

```
# Placeholder for Python code to generate an even-ordered magic square
# This example focuses on a doubly even square (n = 4k)

def generate_doubly_even_magic_square(n):
    # Your implementation for doubly even squares
    # Example for n=4:
    # magic_square = [[0 for _ in range(n)] for _ in range(n)]
    #
    # # Fill naturally
    # k = 1
    # for i in range(n):
    #     for j in range(n):
    #         magic_square[i][j] = k
    #         k += 1
    #
    # # Swap elements based on pattern
    # for i in range(n):
    #     for j in range(n):
    #         if (i % 4 == 0 and j % 4 == 0) or \
    #            (i % 4 == 1 and j % 4 == 1) or \
    #            (i % 4 == 2 and j % 4 == 2) or \
    #            (i % 4 == 3 and j % 4 == 3) or \
    #            (i % 4 == 0 and j % 4 == 3) or \
    #            (i % 4 == 1 and j % 4 == 2) or \
    #            (i % 4 == 2 and j % 4 == 1) or \
    #            (i % 4 == 3 and j % 4 == 0):
    #             # These are the "unchanged" cells in some methods,
    #             # or swapped based on specific patterns.
    #             # A common method swaps (i,j) with (n-1-i, n-1-j) for
specific cells.
    #             pass # Complex logic goes here
    #
    # # A simpler doubly even method:
    # # 1. Fill 1 to N*N sequentially.
    # # 2. Create a swap_matrix (e.g., 1 for cells to swap, 0 for not).
    # #    For n=4, swap_matrix could be:
```

```
    # #    [[1,0,0,1],
    # #     [0,1,1,0],
    # #     [0,1,1,0],
    # #     [1,0,0,1]]
    # # 3. For cells where swap_matrix[i][j] is 1, replace magic_square[i][j]
with (n*n + 1 - magic_square[i][j])
    #
    # return magic_square

# Example usage:
# square = generate_doubly_even_magic_square(4)
# for row in square:
#     print(row)
```

Input:

```
Order of Magic Square (n): 4
```

Expected Output:

```
16  2  3 13
 5 11 10  8
 9  7  6 12
 4 14 15  1
```

Lab 10: Program to implement five House logic puzzle problem

Title: Five House Logic Puzzle Problem (Zebra Puzzle / Einstein's Riddle)

Aim: To implement a program that solves the Five House Logic Puzzle using logical deduction or constraint satisfaction techniques.

Procedure:

1. **Understand the Puzzle:** Familiarize yourself with the 15 clues of the Zebra Puzzle.
2. **Represent Entities:** Define variables and their possible domains for each attribute (color, nationality, drink, pet, cigarette) for each of the five houses.
   - Example: `house1_color`, `house2_nationality`, etc.
   - Domains: `colors = {red, green, blue, yellow, ivory}`, `nationalities = {Englishman, Spaniard, Norwegian, Ukrainian, Japanese}`, etc.
3. **Formulate Constraints:** Translate each of the 15 clues into logical constraints.
   - Example: "The Englishman lives in the Red house" -> `nationality[house_i] == Englishman` implies `color[house_i] == Red`.
   - "The Norwegian lives in the first house" -> `nationality[house1] == Norwegian`.
   - "The Green house is immediately to the right of the Ivory house" -> `color[house_i] == Ivory` implies `color[house_i+1] == Green`.
4. **Choose Solution Method:**
   - **Brute-Force with Pruning:** Generate all possible permutations and check if they satisfy all constraints (inefficient).
   - **Backtracking Search:** A more efficient method. Recursively try to assign values to variables, and if a constraint is violated, backtrack.
   - **Constraint Satisfaction Problem (CSP) Solver:** Use a specialized library or implement a basic CSP solver with techniques like Arc Consistency (AC-3) and Forward Checking.
5. **Print Solution:** Once a valid assignment is found, print the complete solution, specifically identifying who owns the zebra and who drinks water.

Source Code:

```
# Placeholder for Python code to solve the Five House Logic Puzzle
# This typically involves a backtracking search or a CSP library.

def solve_zebra_puzzle():
    # Define variables, domains, and constraints
    # Example structure (conceptual):
    # houses = [{}, {}, {}, {}, {}] # Each dict for a house's attributes
    #
    # nationalities = ['Englishman', 'Spaniard', 'Norwegian', 'Ukrainian',
'Japanese']
    # colors = ['Red', 'Green', 'Ivory', 'Yellow', 'Blue']
    # pets = ['Dog', 'Snails', 'Fox', 'Horse', 'Zebra']
    # drinks = ['Coffee', 'Tea', 'Milk', 'Orange Juice', 'Water']
    # cigarettes = ['Old Gold', 'Kools', 'Chesterfields', 'Lucky Strike',
'Parliaments']
    #
    # # Use a recursive backtracking function
    # def backtrack(house_index):
    #     if house_index == 5:
    #         # Check all 15 constraints for the complete assignment
    #         if check_all_constraints(houses):
    #             return True
    #         return False
```

```
    #
    #     # Try assigning values for current house_index
    #     # Iterate through permutations of remaining attributes
    #     # If a partial assignment is valid, recurse: backtrack(house_index
+ 1)
    #     # If not, backtrack (undo assignment)
    #
    # # Initial call: backtrack(0)
    # # If solution found, print it.
    pass

# solve_zebra_puzzle()
```

Input:

```
(No direct user input; the puzzle rules are hardcoded within the program.)
```

Expected Output:

```
The solution to the Zebra Puzzle:

House 1: Norwegian, Yellow, Water, Kools, Fox
House 2: Ukrainian, Blue, Tea, Horse, Chesterfields
House 3: Englishman, Red, Milk, Snails, Old Gold
House 4: Spaniard, Ivory, Orange Juice, Dog, Lucky Strike
House 5: Japanese, Green, Coffee, Zebra, Parliaments

The Japanese owns the Zebra.
The Norwegian drinks Water.
```

Lab 11: Program for solving A * shortest path algorithm.

Title: A* Shortest Path Algorithm

Aim: To implement the A* shortest path algorithm to find the shortest path between two nodes in a graph.

Procedure:

1. *Understand A:** A* is an informed search algorithm that uses a heuristic function to guide its search. It combines the cost to reach a node (`g(n)`) with an estimated cost from that node to the goal (`h(n)`). The total estimated cost is `f(n) = g(n) + h(n)`.
2. **Represent Graph:** Model the problem as a graph with nodes and weighted edges.
3. **Heuristic Function (**h(n)**):** Define an admissible heuristic function. An admissible heuristic never overestimates the actual cost to reach the goal (e.g., Euclidean distance or Manhattan distance for grid-based problems).
4. **Data Structures:**
   - `open_set` (priority queue/min-heap): Stores nodes to be evaluated, ordered by their `f(n)` value.
   - `closed_set` (set): Stores nodes that have already been evaluated.
   - `g_score` (dictionary): Stores the cost from the start node to each node.
   - `f_score` (dictionary): Stores the estimated total cost from the start node to the goal through each node.
   - `came_from` (dictionary): Stores the predecessor of each node to reconstruct the path.
5. **Algorithm Steps:**
   - Initialize `g_score` of start node to 0, others to infinity.
   - Initialize `f_score` of start node to `h(start_node)`, others to infinity.
   - Add start node to `open_set`.
   - While `open_set` is not empty:
     - Pop the node with the lowest `f_score` from `open_set`.
     - If it's the goal node, reconstruct and return the path.
     - Move current node to `closed_set`.
     - For each neighbor of the current node:
       - Calculate `tentative_g_score`.
       - If `tentative_g_score` is less than `g_score[neighbor]`, update `g_score`, `f_score`, and `came_from`, and add/update neighbor in `open_set`.

Source Code:

```
# Placeholder for Python code for A* shortest path algorithm

import heapq

def a_star_search(graph, start_node, goal_node, heuristic):
    # Your A* implementation here
    # Example:
    # open_set = [(0, start_node)] # (f_score, node)
    # came_from = {}
    # g_score = {node: float('inf') for node in graph}
    # g_score[start_node] = 0
    # f_score = {node: float('inf') for node in graph}
    # f_score[start_node] = heuristic(start_node, goal_node)
    #
    # while open_set:
```

```
    #     current_f, current_node = heapq.heappop(open_set)
    #
    #     if current_node == goal_node:
    #         path = []
    #         while current_node in came_from:
    #             path.insert(0, current_node)
    #             current_node = came_from[current_node]
    #         path.insert(0, start_node)
    #         return path, g_score[goal_node]
    #
    #     for neighbor, weight in graph[current_node].items():
    #         tentative_g_score = g_score[current_node] + weight
    #
    #         if tentative_g_score < g_score[neighbor]:
    #             came_from[neighbor] = current_node
    #             g_score[neighbor] = tentative_g_score
    #             f_score[neighbor] = tentative_g_score + heuristic(neighbor,
goal_node)
    #             heapq.heappush(open_set, (f_score[neighbor], neighbor))
    #
    # return None, float('inf') # No path found

# Example usage:
# graph = {
#     'A': {'B': 1, 'C': 4},
#     'B': {'D': 5, 'E': 2},
#     'C': {'F': 1},
#     'D': {'G': 3},
#     'E': {'G': 2},
#     'F': {'G': 1},
#     'G': {}
# }
#
# # Simple heuristic (e.g., straight-line distance if coordinates are known)
# # For demonstration, assume a dummy heuristic
# def dummy_heuristic(node, goal):
#     return 0 # A* becomes Dijkstra if heuristic is 0
#
# path, cost = a_star_search(graph, 'A', 'G', dummy_heuristic)
# print(f"Path: {path}, Cost: {cost}")
```

Input:

```
Graph Definition:
Nodes: A, B, C, D, E, F, G
Edges (and weights):
A-B: 10, A-C: 15
B-D: 20, B-E: 5
C-F: 10
D-G: 10
E-G: 5
F-G: 5

Heuristic Values (h(n) to G):
h(A)=20, h(B)=15, h(C)=10, h(D)=10, h(E)=5, h(F)=5, h(G)=0

Start Node: A
Goal Node: G
```

Expected Output:

```
Shortest path from A to G: A -> B -> E -> G
Total Cost: 20
```

Lab 12: Program which demonstrates Best First Search.

Title: Best-First Search (Greedy Best-First Search)

Aim: To implement a program that demonstrates the Best-First Search (also known as Greedy Best-First Search) algorithm for finding a path in a graph.

Procedure:

1. **Understand Best-First Search:** Best-First Search expands the node that appears to be closest to the goal, as estimated by a heuristic function `h(n)`. Unlike A*, it does not consider the cost to reach the current node (`g(n)`).
2. **Represent Graph:** Model the problem as a graph with nodes and edges.
3. **Heuristic Function (h(n)):** Define a heuristic function that estimates the cost from node `n` to the goal.
4. **Data Structures:**
    - `open_list` (priority queue/min-heap): Stores nodes to be evaluated, ordered by their `h(n)` value.
    - `closed_list` (set): Stores nodes that have already been evaluated.
    - `came_from` (dictionary): Stores the predecessor of each node to reconstruct the path.
5. **Algorithm Steps:**
    - Add the start node to `open_list` with its heuristic value.
    - While `open_list` is not empty:
        - Pop the node with the lowest `h(n)` from `open_list`.
        - If it's the goal node, reconstruct and return the path.
        - Move current node to `closed_list`.
        - For each neighbor of the current node:
            - If the neighbor is not in `closed_list` and not in `open_list`:
                - Calculate its heuristic value `h(neighbor)`.
                - Set `came_from[neighbor] = current_node`.
                - Add `(h(neighbor), neighbor)` to `open_list`.

Source Code:

```
# Placeholder for Python code for Best-First Search

import heapq

def best_first_search(graph, start_node, goal_node, heuristic):
    # Your Best-First Search implementation here
    # Example:
    # open_list = [(heuristic(start_node, goal_node), start_node)] #
(h_score, node)
    # came_from = {}
    # closed_list = set()
    #
    # while open_list:
    #     current_h, current_node = heapq.heappop(open_list)
    #
    #     if current_node == goal_node:
    #         path = []
    #         while current_node in came_from:
    #             path.insert(0, current_node)
    #             current_node = came_from[current_node]
    #         path.insert(0, start_node)
```

```
    #         return path
    #
    #     closed_list.add(current_node)
    #
    #     for neighbor in graph[current_node]: # Assuming unweighted graph
for simplicity, or just use nodes
    #         if neighbor not in closed_list:
    #             # Check if already in open_list to update if better h_score
(though not typical for pure BFS)
    #             # For simple BFS, just add if not visited.
    #             if neighbor not in [node for h, node in open_list]:
    #                 came_from[neighbor] = current_node
    #                 heapq.heappush(open_list, (heuristic(neighbor,
goal_node), neighbor))
    #
    # return None # No path found

# Example usage:
# graph = {
#     'A': ['B', 'C'],
#     'B': ['D', 'E'],
#     'C': ['F'],
#     'D': ['G'],
#     'E': ['G'],
#     'F': ['G'],
#     'G': []
# }
#
# # Simple heuristic (e.g., straight-line distance if coordinates are known)
# # For demonstration, assume a dummy heuristic
# def dummy_heuristic(node, goal):
#     # This heuristic would need to be defined based on the problem
#     # For example, if nodes were (x,y) coordinates, it could be Euclidean
distance.
#     # For this generic graph, let's assign arbitrary values for
demonstration:
#     h_values = {'A': 6, 'B': 4, 'C': 5, 'D': 2, 'E': 1, 'F': 1, 'G': 0}
#     return h_values.get(node, float('inf'))
#
# path = best_first_search(graph, 'A', 'G', dummy_heuristic)
# print(f"Path: {path}")
```

Input:

```
Graph Definition:
Nodes: S, A, B, C, D, E, G
Edges:
S-A, S-B
A-C, A-D
B-E
C-G
D-G
E-G

Heuristic Values (h(n) to G):
h(S)=10, h(A)=8, h(B)=7, h(C)=3, h(D)=4, h(E)=2, h(G)=0

Start Node: S
Goal Node: G
```

Expected Output:

```
Path found by Best-First Search: S -> B -> E -> G
```

Lab 13: Program which demonstrate the precedence properties of operators in C language.

Title: Operator Precedence and Associativity in C Language

Aim: To write a C program that demonstrates the precedence and associativity rules of various operators in the C programming language.

Procedure:

1. **Select Operators:** Choose a variety of C operators with different precedence levels (e.g., arithmetic `+`, `-`, `*`, `/`, `%`; relational `==`, `!=`, `<`, `>`; logical `&&`, `||`, `!`; assignment `=`).
2. **Construct Expressions:** Create C expressions that combine these operators in ways that highlight their precedence.
   - Example: `a + b * c` (multiplication before addition).
   - Example: `a && b || c` (logical AND before logical OR).
3. **Use Parentheses:** Include expressions with explicit parentheses to show how they override default precedence.
   - Example: `(a + b) * c`.
4. **Demonstrate Associativity:** For operators with the same precedence (e.g., `+`, `-` are left-to-right associative; assignment `=` is right-to-left associative), create expressions to show their grouping.
   - Example: `a = b = c;` (evaluates `b = c` first).
5. **Print Results:** Use `printf` statements to display the expressions and their computed results, along with comments explaining the evaluation order.

Source Code:

```
// Placeholder for C code to demonstrate operator precedence and
associativity

#include <stdio.h>

int main() {
    int a = 10, b = 5, c = 2, d = 0;

    printf("Demonstrating Operator Precedence and Associativity in C:\n\n");

    // Arithmetic Operators: *, / have higher precedence than +, -
    printf("1. Arithmetic Precedence:\n");
    d = a + b * c; // Equivalent to a + (b * c)
    printf("   a + b * c (10 + 5 * 2) = %d\n", d); // Expected: 20
    d = (a + b) * c; // Parentheses override precedence
    printf("   (a + b) * c ((10 + 5) * 2) = %d\n", d); // Expected: 30
    printf("\n");

    // Relational and Logical Operators: && has higher precedence than ||
    printf("2. Logical Precedence:\n");
    int x = 1, y = 0, z = 1;
    d = x && y || z; // Equivalent to (x && y) || z
    printf("   x && y || z (1 && 0 || 1) = %d\n", d); // Expected: 1 (True)
    d = x && (y || z); // Parentheses override precedence
    printf("   x && (y || z) (1 && (0 || 1)) = %d\n", d); // Expected: 1
(True)
    printf("\n");

    // Assignment Operator: Right-to-left associativity
```

```
    printf("3. Assignment Associativity:\n");
    int p, q, r;
    p = q = r = 100; // Evaluates r = 100, then q = r, then p = q
    printf("   p = q = r = 100; p=%d, q=%d, r=%d\n", p, q, r); // Expected:
100 100 100
    printf("\n");

    // Increment/Decrement and other operators
    printf("4. Mixed Operators:\n");
    int val = 5;
    int result = ++val * 2; // ++val (pre-increment) happens before *
    printf("   ++val * 2 (val=5 initially) = %d (val becomes %d)\n", result,
val); // Expected: 12 (val is 6)

    val = 5;
    result = val++ * 2; // val++ (post-increment) happens after *
    printf("   val++ * 2 (val=5 initially) = %d (val becomes %d)\n", result,
val); // Expected: 10 (val is 6)
    printf("\n");

    return 0;
}
```

Input:

```
(No direct user input; values are hardcoded in the program.)
```

Expected Output:

```
Demonstrating Operator Precedence and Associativity in C:

1. Arithmetic Precedence:
   a + b * c (10 + 5 * 2) = 20
   (a + b) * c ((10 + 5) * 2) = 30

2. Logical Precedence:
   x && y || z (1 && 0 || 1) = 1
   x && (y || z) (1 && (0 || 1)) = 1

3. Assignment Associativity:
   p = q = r = 100; p=100, q=100, r=100

4. Mixed Operators:
   ++val * 2 (val=5 initially) = 12 (val becomes 6)
   val++ * 2 (val=5 initially) = 10 (val becomes 6)
```

Lab 14: Program to calculate factorial of a number using recursion.

Title: Factorial Calculation using Recursion

Aim: To implement a program that calculates the factorial of a given non-negative integer using a recursive function.

Procedure:

1. **Understand Factorial:** The factorial of a non-negative integer `n`, denoted by n!, is the product of all positive integers less than or equal to `n`.
   - $n!=n\times(n-1)\times(n-2)\times\cdots\times1$**Error! Filename not specified.**
   - Special case: $0!=1$**Error! Filename not specified.**
2. **Define Recursive Function:** Create a function `factorial(n)` that calls itself.
   - **Base Case:** If `n` is 0, return 1 (this stops the recursion).
   - **Recursive Step:** If `n` is greater than 0, return `n * factorial(n-1)`.
3. **User Input:** Prompt the user to enter a non-negative integer.
4. **Call Function and Print:** Call the `factorial` function with the user's input and print the result.
5. **Error Handling:** Include basic error handling for negative input.

Source Code:

```
# Placeholder for Python code to calculate factorial using recursion

def factorial(n):
    # Your recursive implementation here
    # Example:
    # if n < 0:
    #     return "Factorial is not defined for negative numbers."
    # elif n == 0:
    #     return 1
    # else:
    #     return n * factorial(n - 1)

# Example usage:
# num = int(input("Enter a non-negative integer: "))
# result = factorial(num)
# print(f"The factorial of {num} is {result}")
```

Input:

```
Enter a non-negative integer: 5
```

Expected Output:

```
The factorial of 5 is 120
```

Lab 15: Program to implement five House logic puzzle problem

Title: Five House Logic Puzzle Problem (Zebra Puzzle / Einstein's Riddle) - Reiteration

Aim: To re-implement or further explore the Five House Logic Puzzle (Zebra Puzzle) using logical deduction or constraint satisfaction techniques. (This lab appears to be a duplicate of Lab 10, suggesting a deeper dive or alternative implementation.)

Procedure:

1. **Review Puzzle:** Re-familiarize with the 15 clues of the Zebra Puzzle.
2. **Refine Representation:** Consider alternative ways to represent the puzzle's entities and relationships (e.g., using classes, more advanced data structures).
3. **Advanced Solution Method (Optional):** If Lab 10 used a basic backtracking, consider implementing a more efficient Constraint Satisfaction Problem (CSP) solver with techniques like:
   - **Forward Checking:** When a variable is assigned a value, remove inconsistent values from the domains of its unassigned neighbors.
   - **Arc Consistency (AC-3):** Ensure that for every arc (constraint) between two variables, every value in the domain of one variable has a consistent value in the domain of the other.
   - **Min-Conflicts Heuristic:** For local search, choose the variable that is involved in the most conflicts.
4. **Print Solution:** Once a valid assignment is found, print the complete solution, specifically identifying who owns the zebra and who drinks water.

Source Code:

```
# Placeholder for Python code to solve the Five House Logic Puzzle
(reiteration)
# This could be an enhanced version of Lab 10's solution.

def solve_zebra_puzzle_v2():
    # Your (potentially improved) implementation here
    # This might involve more sophisticated CSP techniques or a different
approach
    # compared to Lab 10.
    pass

# solve_zebra_puzzle_v2()
```

Input:

```
(No direct user input; the puzzle rules are hardcoded within the program.)
```

Expected Output:

```
The solution to the Zebra Puzzle:

House 1: Norwegian, Yellow, Water, Kools, Fox
House 2: Ukrainian, Blue, Tea, Horse, Chesterfields
House 3: Englishman, Red, Milk, Snails, Old Gold
House 4: Spaniard, Ivory, Orange Juice, Dog, Lucky Strike
House 5: Japanese, Green, Coffee, Zebra, Parliaments

The Japanese owns the Zebra.
```

The Norwegian drinks Water.