Experiment 1: Recursive Depth First Search (DFS) from CSV

Theory

Depth First Search (DFS) is an algorithm for traversing or searching tree or graph data structures. The algorithm starts at the root node (selecting some arbitrary node as the root node in the case of a graph) and explores as far as possible along each branch before backtracking.

Key Concepts:

- **Graph Representation:** An undirected, unweighted graph can be represented using an adjacency list, where each node has a list of its adjacent nodes.
- **Recursion:** The recursive approach uses the function call stack to keep track of the visited nodes and the path taken.
- **Visited Set:** To avoid cycles and redundant processing, a set or boolean array is used to keep track of nodes that have already been visited.
- **CSV Input:** The graph structure (edges) is read from a Comma Separated Values (.csv) file. Each row typically represents an edge, listing the two nodes it connects.

How it works (Recursive):

1. Start DFS from a given starting node.
2. Mark the current node as visited.
3. Process the current node (e.g., print it).
4. For each neighbor of the current node:
    - If the neighbor has not been visited, recursively call DFS on the neighbor.
5. If the graph might be disconnected, iterate through all nodes and start DFS if a node hasn't been visited yet.

Pseudocode/Algorithm

```
DFS(graph, start_node, visited):
  Mark start_node as visited
  Print start_node

  For each neighbor in graph.get_neighbors(start_node):
    If neighbor is not in visited:
      DFS(graph, neighbor, visited)

Main Function:
  Read graph data from CSV file
  Create adjacency list representation of the graph
  Initialize an empty set 'visited'
  Choose a starting node 'start'

  // Handle disconnected graphs (optional, depends on requirement)
  // For each node 'v' in graph:
  //   If 'v' is not in visited:
  //     DFS(graph, v, visited)

  // Assuming a connected graph or starting from a specific node:
  If start node exists in graph:
      DFS(graph, start, visited)
  Else:
      Print "Start node not found"
```

Python Code

```
import csv
from collections import defaultdict

def build_graph_from_csv(filename):
    """Reads an undirected graph from a CSV file.

    Args:
        filename (str): The path to the CSV file.
                        Expected format: each row contains two nodes representing an edge.

    Returns:
        defaultdict: An adjacency list representation of the graph.
    """
    graph = defaultdict(list)
    try:
        with open(filename, 'r') as file:
            reader = csv.reader(file)
            # Skip header if exists (optional)
            # next(reader, None)
            for row in reader:
                if len(row) >= 2:
                    u, v = row[0].strip(), row[1].strip()
                    graph[u].append(v)
                    graph[v].append(u) # Because it's an undirected graph
                else:
                    print(f"Skipping invalid row: {row}")
    except FileNotFoundError:
        print(f"Error: File '{filename}' not found.")
        return None
    except Exception as e:
        print(f"An error occurred while reading the CSV: {e}")
        return None
    return graph

def dfs_recursive(graph, node, visited):
    """Performs Recursive Depth First Search.

    Args:
        graph (defaultdict): The adjacency list of the graph.
        node (str): The current node to visit.
        visited (set): A set of visited nodes.
    """
    visited.add(node)
    print(node, end=' ')

    for neighbor in graph[node]:
        if neighbor not in visited:
            dfs_recursive(graph, neighbor, visited)

# --- Main Execution ---
if __name__ == "__main__":
    csv_file = 'graph.csv' # Make sure this file exists in the same directory or provide the full path
    start_node = 'A'      # Choose your starting node

    # 1. Create a dummy graph.csv for testing
    try:
        with open(csv_file, 'w', newline='') as f:
            writer = csv.writer(f)
            writer.writerow(['Node1', 'Node2'])
            writer.writerow(['A', 'B'])
            writer.writerow(['A', 'C'])
            writer.writerow(['B', 'D'])
            writer.writerow(['B', 'E'])
            writer.writerow(['C', 'F'])
            writer.writerow(['E', 'F'])
```

```
            writer.writerow(['G', 'H']) # Example of a disconnected component
        print(f"Created dummy '{csv_file}' for demonstration.")
    except Exception as e:
        print(f"Could not create dummy CSV: {e}")

    # 2. Build the graph
    graph = build_graph_from_csv(csv_file)

    if graph is not None:
        print(f"\nGraph (Adjacency List):\n{dict(graph)}")

        # 3. Perform DFS
        visited_nodes = set()
        print(f"\nRecursive DFS starting from node '{start_node}':")
        if start_node in graph:
            dfs_recursive(graph, start_node, visited_nodes)
        else:
            print(f"Start node '{start_node}' not found in the graph.")

        # Optional: Handle disconnected components
        print("\n\nChecking for other components:")
        all_nodes = set(graph.keys())
        for node_list in graph.values():
            all_nodes.update(node_list)

        for node in all_nodes:
            if node not in visited_nodes:
                print(f"\nFound unvisited node '{node}', starting DFS for its component:")
                dfs_recursive(graph, node, visited_nodes)
        print("\nDFS complete.")
```

To Run the Code:

1. Save the Python code as a `.py` file (e.g., `dfs_recursive_csv.py`).
2. The code includes functionality to create a sample `graph.csv` file in the same directory if it doesn't exist. You can modify this CSV or create your own with the format:

```
Node1,Node2
A,B
A,C
B,D
...
```

3. Run the script from your terminal: `python dfs_recursive_csv.py`
4. The output will show the graph read from the CSV and the DFS traversal sequence.

Experiment 2: Non-Recursive Depth First Search (DFS) from User Input

Theory

Depth First Search (DFS) is a graph traversal algorithm that explores as far as possible along each branch before backtracking. Unlike the recursive version which uses the function call stack implicitly, the non-recursive (or iterative) version explicitly uses a stack data structure to keep track of the nodes to visit.

Key Concepts:

- **Graph Representation:** An undirected, unweighted graph is typically represented using an adjacency list (a dictionary where keys are nodes and values are lists of their neighbors).
- **Stack:** A stack (Last-In, First-Out) is used to manage the order of nodes to be visited. When a node is visited, its unvisited neighbors are pushed onto the stack.
- **Visited Set:** A set is used to keep track of nodes that have already been visited to prevent cycles and redundant processing.
- **User Input:** The graph structure (nodes and edges) is provided by the user during runtime.

How it works (Non-Recursive/Iterative):

1. Initialize an empty stack and an empty set `visited`.
2. Choose a starting node and push it onto the stack.
3. While the stack is not empty:
    a. Pop a node from the stack. Let's call it `current_node`.
    b. If `current_node` has not been visited:
        i. Mark `current_node` as visited.
        ii. Process `current_node` (e.g., print it).
        iii. For each neighbor of `current_node`:
        - If the neighbor has not been visited, push it onto the stack.
4. If the graph might be disconnected, repeat the process starting from any unvisited node until all nodes are visited.

Pseudocode/Algorithm

```
DFS_Iterative(graph, start_node):
  Initialize an empty stack S
  Initialize an empty set visited

  Push start_node onto S

  While S is not empty:
    current_node = S.pop()

    If current_node is not in visited:
      Mark current_node as visited
      Print current_node

      // Push neighbors in reverse order if specific traversal order matters,
      // otherwise, the order doesn't strictly matter for correctness.
      For each neighbor in graph.get_neighbors(current_node):
        If neighbor is not in visited:
          Push neighbor onto S

Main Function:
  Initialize an empty graph (e.g., adjacency list)
  Ask user for the number of edges
  Loop for the number of edges:
    Ask user for the two nodes forming an edge (u, v)
    Add edge (u, v) and (v, u) to the graph // Undirected

  Ask user for the starting node 'start'

  If start node exists in graph:
      DFS_Iterative(graph, start)
  Else:
      Print "Start node not found"

  // Optional: Handle disconnected graphs by iterating through all known nodes
  // all_nodes = get_all_nodes(graph)
  // for node in all_nodes:
  //   if node not in visited:
  //      DFS_Iterative(graph, node) // Need to manage visited set across calls
```

Python Code

```
from collections import defaultdict

def build_graph_from_user():
    """Builds an undirected graph based on user input."""
    graph = defaultdict(list)
    nodes = set()
    while True:
        try:
            num_edges = int(input("Enter the number of edges: "))
            if num_edges < 0:
                print("Number of edges cannot be negative. Please try again.")
                continue
            break
        except ValueError:
            print("Invalid input. Please enter an integer.")

    print(f"Enter {num_edges} edges (format: node1 node2, e.g., A B):")
    for i in range(num_edges):
        while True:
            try:
                edge = input(f"Edge {i+1}: ").strip().split()
                if len(edge) == 2:
                    u, v = edge[0], edge[1]
                    graph[u].append(v)
                    graph[v].append(u) # Undirected graph
                    nodes.add(u)
                    nodes.add(v)
                    break
                else:
                    print("Invalid format. Please enter two nodes separated by space.")
            except Exception as e:
                print(f"An error occurred: {e}. Please try again.")

    return graph, nodes

def dfs_iterative(graph, start_node, all_nodes):
    """Performs Non-Recursive (Iterative) Depth First Search.

    Args:
        graph (defaultdict): The adjacency list of the graph.
        start_node (str): The node to start the search from.
        all_nodes (set): A set of all nodes in the graph (for handling disconnected parts).
    """
    if start_node not in all_nodes:
        print(f"Error: Start node '{start_node}' is not in the graph.")
        # Optionally, pick an arbitrary node from all_nodes if start_node is invalid
        # if all_nodes:
        #     start_node = next(iter(all_nodes))
        #     print(f"Starting from arbitrary node '{start_node}' instead.")
        # else:
        #     print("Graph is empty.")
        #     return
        return

    visited = set()
    stack = [start_node]
    print(f"\nIterative DFS starting from node '{start_node}':")

    while stack:
        node = stack.pop()

        if node not in visited:
            visited.add(node)
            print(node, end=' ')
```

```
            # Add neighbors to the stack (in reverse order to mimic recursion)
            # For simple traversal, order doesn't strictly matter.
            # Neighbors are added only if they haven't been visited yet.
            # Adding all neighbors and checking visited status upon popping is also valid.
            for neighbor in reversed(graph.get(node, [])): # Use get for safety
                if neighbor not in visited:
                    stack.append(neighbor)

    # Handle disconnected components
    remaining_nodes = all_nodes - visited
    if remaining_nodes:
        print("\n\nChecking for other components:")
        while remaining_nodes:
            next_start_node = remaining_nodes.pop()
            if next_start_node not in visited:
                print(f"\nFound unvisited node '{next_start_node}', starting DFS for its component:")
                stack = [next_start_node]
                while stack:
                    node = stack.pop()
                    if node not in visited:
                        visited.add(node)
                        print(node, end=' ')
                        remaining_nodes.discard(node)
                        for neighbor in reversed(graph.get(node, [])):
                            if neighbor not in visited:
                                stack.append(neighbor)
    print("\nDFS complete.")


# --- Main Execution ---
if __name__ == "__main__":
    # 1. Build the graph from user input
    graph_adj_list, graph_nodes = build_graph_from_user()

    if not graph_nodes:
        print("\nGraph is empty. Exiting.")
    else:
        print(f"\nGraph (Adjacency List):\n{dict(graph_adj_list)}")
        print(f"All nodes: {graph_nodes}")

        # 2. Get the starting node from the user
        while True:
            start = input("Enter the starting node for DFS: ").strip()
            if start in graph_nodes:
                break
            else:
                print(f"Node '{start}' not found in the graph nodes {graph_nodes}. Please try again.")

        # 3. Perform DFS
        dfs_iterative(graph_adj_list, start, graph_nodes)
```

How to Run:

1. Save the code as a Python file (e.g., `dfs_iterative_user.py`).
2. Run it from the terminal: `python dfs_iterative_user.py`.
3. The program will prompt you to enter the number of edges.
4. Then, it will ask you to enter each edge (two node names separated by space).
5. Finally, it will ask for the starting node for the DFS traversal.
6. The output will show the graph and the sequence of nodes visited during the DFS.

Experiment 3: Breadth First Search (BFS) from User Input

Theory

Breadth First Search (BFS) is another fundamental algorithm for traversing or searching tree or graph data structures. It starts at a selected node (source or root) and explores the neighbor nodes at the present depth prior to moving on to the nodes at the next depth level. It uses a queue data structure to keep track of the nodes to visit.

Key Concepts:

- **Graph Representation:** An undirected, unweighted graph is typically represented using an adjacency list.
- **Queue:** A queue (First-In, First-Out) is used to manage the order of nodes to be visited. Nodes are added to the rear of the queue and removed from the front.
- **Visited Set:** A set is used to keep track of nodes that have already been visited to avoid cycles and redundant processing. A node is marked visited when it's added to the queue.
- **Level-by-Level Exploration:** BFS explores the graph layer by layer, finding the shortest path in terms of the number of edges from the source node to all other reachable nodes in an unweighted graph.
- **User Input:** The graph structure (nodes and edges) is provided by the user during runtime.

How it works (Iterative):

1. Initialize an empty queue `Q` and an empty set `visited`.
2. Choose a starting node `start_node`.
3. Add `start_node` to `Q` and mark `start_node` as visited.
4. While `Q` is not empty:
    a. Dequeue a node from `Q`. Let's call it `current_node`.
    b. Process `current_node` (e.g., print it).
    c. For each neighbor of `current_node`:
        i. If the neighbor has not been visited:
            - Mark the neighbor as visited.
            - Enqueue the neighbor.
5. If the graph might be disconnected, repeat the process starting from any unvisited node until all nodes are visited.

Pseudocode/Algorithm

```
BFS(graph, start_node):
  Initialize an empty queue Q
  Initialize an empty set visited

  If start_node is not in graph:
      Print "Start node not found"
      Return

  Mark start_node as visited
  Enqueue start_node into Q

  While Q is not empty:
    current_node = Q.dequeue()
    Print current_node

    For each neighbor in graph.get_neighbors(current_node):
      If neighbor is not in visited:
        Mark neighbor as visited
        Enqueue neighbor into Q

Main Function:
  Initialize an empty graph (e.g., adjacency list)
  Ask user for the number of edges
  Loop for the number of edges:
    Ask user for the two nodes forming an edge (u, v)
    Add edge (u, v) and (v, u) to the graph // Undirected

  Ask user for the starting node 'start'

  BFS(graph, start)

  // Optional: Handle disconnected graphs by iterating through all known nodes
  // all_nodes = get_all_nodes(graph)
  // visited_bfs = set() // Maintain a separate visited set if calling BFS multiple times
  // for node in all_nodes:
  //   if node not in visited_bfs:
  //      BFS_component(graph, node, visited_bfs) // Modified BFS to use passed visited set
```

Python Code

```
from collections import defaultdict, deque

def build_graph_from_user():
    """Builds an undirected graph based on user input."""
    graph = defaultdict(list)
    nodes = set()
    while True:
        try:
            num_edges = int(input("Enter the number of edges: "))
            if num_edges < 0:
                print("Number of edges cannot be negative. Please try again.")
                continue
            break
        except ValueError:
            print("Invalid input. Please enter an integer.")

    print(f"Enter {num_edges} edges (format: node1 node2, e.g., A B):")
    for i in range(num_edges):
        while True:
            try:
                edge = input(f"Edge {i+1}: ").strip().split()
                if len(edge) == 2:
                    u, v = edge[0], edge[1]
                    graph[u].append(v)
                    graph[v].append(u) # Undirected graph
                    nodes.add(u)
                    nodes.add(v)
                    break
                else:
                    print("Invalid format. Please enter two nodes separated by space.")
            except Exception as e:
                print(f"An error occurred: {e}. Please try again.")

    return graph, nodes

def bfs(graph, start_node, all_nodes):
    """Performs Breadth First Search.

    Args:
        graph (defaultdict): The adjacency list of the graph.
        start_node (str): The node to start the search from.
        all_nodes (set): A set of all nodes in the graph (for handling disconnected parts).
    """
    if start_node not in all_nodes:
        print(f"Error: Start node '{start_node}' is not in the graph.")
        return

    visited = set()
    queue = deque()

    print(f"\nBFS starting from node '{start_node}':")
    visited.add(start_node)
    queue.append(start_node)

    while queue:
        node = queue.popleft() # Dequeue from the front
        print(node, end=' ')

        for neighbor in graph.get(node, []):
            if neighbor not in visited:
                visited.add(neighbor)
                queue.append(neighbor)

    # Handle disconnected components
    remaining_nodes = all_nodes - visited
```

```
    if remaining_nodes:
        print("\n\nChecking for other components:")
        while remaining_nodes:
            next_start_node = remaining_nodes.pop()
            if next_start_node not in visited:
                print(f"\nFound unvisited node '{next_start_node}', starting BFS for its component:")
                visited.add(next_start_node)
                queue.append(next_start_node)
                while queue:
                    node = queue.popleft()
                    print(node, end=' ')
                    remaining_nodes.discard(node) # Remove from remaining as it's processed
                    for neighbor in graph.get(node, []):
                        if neighbor not in visited:
                            visited.add(neighbor)
                            queue.append(neighbor)

    print("\nBFS complete.")

# --- Main Execution ---
if __name__ == "__main__":
    # 1. Build the graph from user input
    graph_adj_list, graph_nodes = build_graph_from_user()

    if not graph_nodes:
        print("\nGraph is empty. Exiting.")
    else:
        print(f"\nGraph (Adjacency List):\n{dict(graph_adj_list)}")
        print(f"All nodes: {graph_nodes}")

        # 2. Get the starting node from the user
        while True:
            start = input("Enter the starting node for BFS: ").strip()
            if start in graph_nodes:
                break
            else:
                print(f"Node '{start}' not found in the graph nodes {graph_nodes}. Please try again.")

        # 3. Perform BFS
        bfs(graph_adj_list, start, graph_nodes)
```

How to Run:

1. Save the code as a Python file (e.g., `bfs_user.py`).
2. Run it from the terminal: `python bfs_user.py`.
3. The program will prompt you to enter the number of edges.
4. Then, it will ask you to enter each edge (two node names separated by space).
5. Finally, it will ask for the starting node for the BFS traversal.
6. The output will show the graph and the sequence of nodes visited during the BFS, exploring level by level.

Experiment 4: Best First Search (Directed, Unweighted Graph, User Input)

Theory

Best-First Search (BestFS) is an informed search algorithm that explores a graph by expanding the most promising node chosen according to a specified rule or heuristic function. It typically uses a priority queue to manage the nodes to visit, prioritizing nodes that appear closer to the goal based on the heuristic estimate.

This specific variant is often referred to as **Greedy Best-First Search** because it greedily chooses the node that *appears* to be closest to the goal according to the heuristic, without considering the cost incurred so far to reach that node (unlike A* search).

Key Concepts:

- **Graph Representation:** A directed, unweighted graph can be represented using an adjacency list where each key (node) maps to a list of nodes it points to.
- **Heuristic Function (h(n)):** An estimate of the cost from node `n` to the goal node. This value is provided by the user for each node.
- **Priority Queue:** Used to store nodes to be explored, ordered by their heuristic values (lowest heuristic value has higher priority).
- **Visited Set:** To prevent cycles and redundant exploration, a set keeps track of visited nodes.
- **User Input:** The graph structure (directed edges) and the heuristic value for each node are provided by the user.

How it works:

1. Initialize an empty priority queue `PQ` and an empty set `visited`.
2. Get the start node and the goal node from the user.
3. Get the heuristic value `h(start_node)` for the start node.
4. Add the start node to `PQ` with its heuristic value as priority: `PQ.add(start_node, h(start_node))`.
5. While `PQ` is not empty:
    a. Extract the node with the lowest heuristic value from `PQ`. Let's call it `current_node`.
    b. If `current_node` is the goal node, the path is found (though BestFS doesn't inherently store the path, it can be reconstructed if needed). Return success.
    c. If `current_node` has already been visited, continue to the next iteration (this handles cycles and ensures efficiency).
    d. Mark `current_node` as visited.
    e. Process `current_node` (e.g., print it as part of the explored sequence).
    f. For each neighbor of `current_node`:
        i. If the neighbor has not been visited:
            - Get the heuristic value `h(neighbor)` for the neighbor.
            - Add the neighbor to `PQ` with its heuristic value as priority: `PQ.add(neighbor, h(neighbor))`.
6. If the loop finishes and the goal was not reached, the goal is unreachable from the start node. Return failure.

Pseudocode/Algorithm

```
Best_First_Search(graph, heuristics, start_node, goal_node):
  Initialize an empty priority queue PQ // Stores (heuristic_value, node)
  Initialize an empty set visited

  If start_node is not in graph or goal_node is not in graph:
      Print "Start or Goal node not found in graph"
      Return Failure

  Add (heuristics[start_node], start_node) to PQ

  While PQ is not empty:
    (h_value, current_node) = PQ.extract_min() // Get node with lowest heuristic

    If current_node is goal_node:
      Print "Goal reached!"
      // Optional: Print path if tracked
      Return Success

    If current_node is in visited:
      Continue // Already processed this node via a potentially better (though not relevant in greedy BestFS) or same path

    Mark current_node as visited
    Print "Visiting node:", current_node

    For each neighbor in graph.get_neighbors(current_node):
      If neighbor is not in visited:
        If neighbor is in heuristics:
            Add (heuristics[neighbor], neighbor) to PQ
        Else:
            Print "Warning: Heuristic not found for neighbor", neighbor
            // Decide how to handle: skip, assume infinity, assume 0?
            // Assuming skip for now

  Print "Goal not reachable"
  Return Failure

Main Function:
  Initialize an empty graph (e.g., directed adjacency list)
  Initialize an empty dictionary 'heuristics'
  Ask user for the number of nodes and edges

  // Get Nodes and Heuristics
  Loop for the number of nodes:
      Ask user for node name
      Ask user for heuristic value for this node
      Store in 'heuristics' dictionary

  // Get Edges (Directed)
  Loop for the number of edges:
      Ask user for the source and destination nodes forming a directed edge (u -> v)
      Add edge u -> v to the graph

  Ask user for the starting node 'start'
  Ask user for the goal node 'goal'

  Best_First_Search(graph, heuristics, start, goal)
```

Python Code

```
import heapq
from collections import defaultdict

def build_graph_and_heuristics_from_user():
    """Builds a directed graph and collects heuristics based on user input."""
    graph = defaultdict(list)
    heuristics = {}
    nodes = set()

    # Get nodes and their heuristic values
    while True:
        try:
            num_nodes = int(input("Enter the number of nodes: "))
            if num_nodes <= 0:
                print("Number of nodes must be positive. Please try again.")
                continue
            break
        except ValueError:
            print("Invalid input. Please enter an integer.")

    print(f"Enter {num_nodes} node names and their heuristic values:")
    for i in range(num_nodes):
        while True:
            try:
                node_name = input(f"Node {i+1} name: ").strip()
                if not node_name:
                    print("Node name cannot be empty.")
                    continue
                heuristic_value = float(input(f"Heuristic value for node '{node_name}': ").strip())
                if node_name in heuristics:
                    print(f"Warning: Node '{node_name}' entered previously. Overwriting heuristic.")
                heuristics[node_name] = heuristic_value
                nodes.add(node_name)
                break
            except ValueError:
                print("Invalid heuristic value. Please enter a number.")
            except Exception as e:
                print(f"An error occurred: {e}. Please try again.")

    # Get edges
    while True:
        try:
            num_edges = int(input("Enter the number of directed edges: "))
            if num_edges < 0:
                print("Number of edges cannot be negative. Please try again.")
                continue
            break
        except ValueError:
            print("Invalid input. Please enter an integer.")

    print(f"Enter {num_edges} directed edges (format: source_node destination_node, e.g., A B for A -> B):")
    for i in range(num_edges):
        while True:
            try:
                edge = input(f"Edge {i+1}: ").strip().split()
                if len(edge) == 2:
                    u, v = edge[0], edge[1]
                    if u not in nodes or v not in nodes:
                        print(f"Error: One or both nodes ('{u}', '{v}') were not defined earlier. Please define all nodes first.")
                        continue # Ask for this edge again
                    graph[u].append(v) # Directed edge u -> v
                    break
                else:
                    print("Invalid format. Please enter two nodes separated by space.")
            except Exception as e:
```

```
                print(f"An error occurred: {e}. Please try again.")

    return graph, heuristics, nodes

def best_first_search(graph, heuristics, start_node, goal_node):
    """Performs Best-First Search (Greedy version).

    Args:
        graph (defaultdict): The directed adjacency list of the graph.
        heuristics (dict): A dictionary mapping nodes to their heuristic values.
        start_node (str): The node to start the search from.
        goal_node (str): The target node.

    Returns:
        list or None: A list representing the path found (sequence of visited nodes
                      leading to goal), or None if the goal is not reachable.
                      Note: This implementation primarily shows the visited order,
                      path reconstruction needs extra logic (e.g., storing parent pointers).
    """
    if start_node not in heuristics or goal_node not in heuristics:
        print(f"Error: Start node '{start_node}' or Goal node '{goal_node}' not found or has no heuristic.")
        return None

    priority_queue = []
    heapq.heappush(priority_queue, (heuristics[start_node], start_node))
    visited = set()
    # To reconstruct path, store parent pointers: parent = {start_node: None}
    visited_order = [] # Track the order nodes are processed

    print(f"\nBest-First Search from '{start_node}' to '{goal_node}':")

    while priority_queue:
        h_value, current_node = heapq.heappop(priority_queue)

        if current_node in visited:
            continue

        visited.add(current_node)
        visited_order.append(current_node)
        print(f"Visiting: {current_node} (h={h_value})")

        if current_node == goal_node:
            print("\nGoal reached!")
            # Path reconstruction would happen here using the 'parent' dictionary
            return visited_order # Return the sequence of visited nodes for now

        for neighbor in graph.get(current_node, []):
            if neighbor not in visited:
                if neighbor in heuristics:
                    heapq.heappush(priority_queue, (heuristics[neighbor], neighbor))
                    # parent[neighbor] = current_node # Store parent for path reconstruction
                else:
                    print(f"Warning: Heuristic value missing for neighbor '{neighbor}'. Skipping.")

    print("\nGoal not reachable.")
    return None

# --- Main Execution ---
if __name__ == "__main__":
    # 1. Build graph and heuristics
    graph_adj, node_heuristics, graph_nodes = build_graph_and_heuristics_from_user()

    if not graph_nodes:
        print("\nNo nodes defined. Exiting.")
    else:
        print(f"\nGraph (Directed Adjacency List):\n{dict(graph_adj)}")
```

```
print(f"Heuristics:\n{node_heuristics}")
print(f"All nodes: {graph_nodes}")

# 2. Get start and goal nodes
while True:
    start = input("Enter the starting node: ").strip()
    if start in graph_nodes:
        break
    else:
        print(f"Node '{start}' not found in the defined nodes {graph_nodes}. Please try again.")

while True:
    goal = input("Enter the goal node: ").strip()
    if goal in graph_nodes:
        break
    else:
        print(f"Node '{goal}' not found in the defined nodes {graph_nodes}. Please try again.")

# 3. Perform Best-First Search
path = best_first_search(graph_adj, node_heuristics, start, goal)

if path:
    print(f"\nSequence of visited nodes: {' -> '.join(path)}")
```

How to Run:

1. Save the code as a Python file (e.g., `bestfs_directed_user.py`).
2. Run it from the terminal: `python bestfs_directed_user.py`.
3. The program will first ask for the number of nodes, then the name and heuristic value for each node.
4. Next, it will ask for the number of directed edges and then each edge (source destination).
5. Finally, it will ask for the start and goal nodes.
6. The output will show the graph, heuristics, and the sequence of nodes visited by the Best-First Search algorithm as it tries to reach the goal.

Experiment 5: Best First Search (Undirected, Weighted Graph, User Input)

Theory

Best-First Search (BestFS) is an informed search algorithm that explores a graph by expanding the node deemed most promising according to a heuristic function. It uses a priority queue to prioritize nodes based on their heuristic estimate, aiming to find a path to the goal node efficiently.

This variant, often called **Greedy Best-First Search**, focuses solely on the heuristic value `h(n)` (estimated cost from node `n` to the goal) to decide which node to explore next. It does *not* consider the actual cost `g(n)` (cost from the start node to `n`) incurred so far. Even though the graph is weighted, these weights are not used by the standard Greedy Best-First Search algorithm for prioritization; only the heuristic values matter.

Key Concepts:

- **Graph Representation:** An undirected, weighted graph can be represented using an adjacency list where each entry stores not only the neighbor but also the weight of the edge connecting them (e.g., `graph[node] = [(neighbor1, weight1), (neighbor2, weight2), ...]`).
- **Heuristic Function (h(n)):** An estimate of the cost from node `n` to the goal node. Provided by the user for each node.
- **Priority Queue:** Stores nodes to be explored, ordered by their heuristic values (lowest `h(n)` has highest priority).
- **Visited Set:** Prevents cycles and redundant exploration.
- **User Input:** The graph structure (undirected weighted edges) and the heuristic value for each node are provided by the user.

How it works:

1. Initialize an empty priority queue `PQ` and an empty set `visited`.
2. Get the start node and the goal node from the user.
3. Get the heuristic value `h(start_node)` for the start node.
4. Add the start node to `PQ` with its heuristic value as priority: `PQ.add(start_node, h(start_node))`.
5. While `PQ` is not empty:
    a. Extract the node with the lowest heuristic value from `PQ`. Let's call it `current_node`.

b. If `current_node` is the goal node, the path is found. Return success.
c. If `current_node` has already been visited, continue.
d. Mark `current_node` as visited.
e. Process `current_node` (e.g., print it).
f. For each neighbor of `current_node` (obtained from the weighted adjacency list):
    i. If the neighbor has not been visited:
        - Get the heuristic value `h(neighbor)`.
        - Add the neighbor to `PQ` with its heuristic value as priority: `PQ.add(neighbor, h(neighbor))`.

6. If the loop finishes and the goal was not reached, return failure.

Pseudocode/Algorithm

```
Best_First_Search(graph, heuristics, start_node, goal_node):
  Initialize an empty priority queue PQ // Stores (heuristic_value, node)
  Initialize an empty set visited

  If start_node is not in graph or goal_node is not in graph:
      Print "Start or Goal node not found in graph"
      Return Failure

  Add (heuristics[start_node], start_node) to PQ

  While PQ is not empty:
    (h_value, current_node) = PQ.extract_min() // Get node with lowest heuristic

    If current_node is goal_node:
      Print "Goal reached!"
      Return Success

    If current_node is in visited:
      Continue

    Mark current_node as visited
    Print "Visiting node:", current_node

    // Iterate through neighbors (ignoring weights for priority)
    For each (neighbor, weight) in graph.get_neighbors(current_node):
      If neighbor is not in visited:
        If neighbor in heuristics:
            Add (heuristics[neighbor], neighbor) to PQ
        Else:
            Print "Warning: Heuristic not found for neighbor", neighbor
            // Skip or handle as needed

  Print "Goal not reachable"
  Return Failure

Main Function:
  Initialize an empty graph (e.g., weighted adjacency list)
  Initialize an empty dictionary 'heuristics'
  Ask user for the number of nodes and edges

  // Get Nodes and Heuristics
  Loop for the number of nodes:
      Ask user for node name
      Ask user for heuristic value for this node
      Store in 'heuristics' dictionary

  // Get Edges (Undirected, Weighted)
  Loop for the number of edges:
      Ask user for the two nodes and the weight (u, v, w)
      Add edge (u, v, w) and (v, u, w) to the graph

  Ask user for the starting node 'start'
  Ask user for the goal node 'goal'

  Best_First_Search(graph, heuristics, start, goal)
```

Python Code

```
import heapq
from collections import defaultdict

def build_weighted_graph_and_heuristics_from_user():
    """Builds an undirected weighted graph and collects heuristics from user input."""
    graph = defaultdict(list) # Stores neighbors as (neighbor, weight) tuples
    heuristics = {}
    nodes = set()

    # Get nodes and their heuristic values
    while True:
        try:
            num_nodes = int(input("Enter the number of nodes: "))
            if num_nodes <= 0:
                print("Number of nodes must be positive.")
                continue
            break
        except ValueError:
            print("Invalid input. Please enter an integer.")

    print(f"Enter {num_nodes} node names and their heuristic values:")
    for i in range(num_nodes):
        while True:
            try:
                node_name = input(f"Node {i+1} name: ").strip()
                if not node_name:
                    print("Node name cannot be empty.")
                    continue
                heuristic_value = float(input(f"Heuristic value for node '{node_name}': ").strip())
                if node_name in heuristics:
                    print(f"Warning: Node '{node_name}' entered previously. Overwriting heuristic.")
                heuristics[node_name] = heuristic_value
                nodes.add(node_name)
                break
            except ValueError:
                print("Invalid heuristic value. Please enter a number.")
            except Exception as e:
                print(f"An error occurred: {e}. Please try again.")

    # Get weighted edges
    while True:
        try:
            num_edges = int(input("Enter the number of undirected weighted edges: "))
            if num_edges < 0:
                print("Number of edges cannot be negative.")
                continue
            break
        except ValueError:
            print("Invalid input. Please enter an integer.")

    print(f"Enter {num_edges} edges (format: node1 node2 weight, e.g., A B 5):")
    for i in range(num_edges):
        while True:
            try:
                edge_input = input(f"Edge {i+1}: ").strip().split()
                if len(edge_input) == 3:
                    u, v, weight_str = edge_input
                    weight = float(weight_str) # Or int(weight_str) if weights are integers
                    if u not in nodes or v not in nodes:
                        print(f"Error: One or both nodes ('{u}', '{v}') were not defined. Define nodes first.")
                        continue
                    if weight < 0:
                         print("Warning: Edge weights are typically non-negative for pathfinding, but proceeding.")
                    # Add edge in both directions for undirected graph
                    graph[u].append((v, weight))
```

```
                    graph[v].append((u, weight))
                    break
                else:
                    print("Invalid format. Please enter node1 node2 weight.")
            except ValueError:
                print("Invalid weight. Please enter a numeric value for the weight.")
            except Exception as e:
                print(f"An error occurred: {e}. Please try again.")

    return graph, heuristics, nodes

def best_first_search_weighted(graph, heuristics, start_node, goal_node):
    """Performs Best-First Search (Greedy) on a weighted graph.
       Note: Weights are stored but NOT used for prioritization in Greedy BestFS.

    Args:
        graph (defaultdict): Adjacency list {node: [(neighbor, weight), ...]}.
        heuristics (dict): {node: heuristic_value}.
        start_node (str): The starting node.
        goal_node (str): The target node.

    Returns:
        list or None: Sequence of visited nodes leading to goal, or None if not reachable.
    """
    if start_node not in heuristics or goal_node not in heuristics:
        print(f"Error: Start '{start_node}' or Goal '{goal_node}' not found or lacks heuristic.")
        return None

    priority_queue = []
    heapq.heappush(priority_queue, (heuristics[start_node], start_node))
    visited = set()
    visited_order = []

    print(f"\nBest-First Search (Greedy) from '{start_node}' to '{goal_node}':")

    while priority_queue:
        h_value, current_node = heapq.heappop(priority_queue)

        if current_node in visited:
            continue

        visited.add(current_node)
        visited_order.append(current_node)
        print(f"Visiting: {current_node} (h={h_value})")

        if current_node == goal_node:
            print("\nGoal reached!")
            return visited_order

        # Explore neighbors, prioritize based on heuristic only
        for neighbor, weight in graph.get(current_node, []):
            if neighbor not in visited:
                if neighbor in heuristics:
                    heapq.heappush(priority_queue, (heuristics[neighbor], neighbor))
                else:
                    print(f"Warning: Heuristic missing for neighbor '{neighbor}'. Skipping.")

    print("\nGoal not reachable.")
    return None

# --- Main Execution ---
if __name__ == "__main__":
    # 1. Build graph and heuristics
    graph_adj, node_heuristics, graph_nodes = build_weighted_graph_and_heuristics_from_user()

    if not graph_nodes:
```

```
        print("\nNo nodes defined. Exiting.")
    else:
        print(f"\nGraph (Undirected, Weighted Adjacency List):")
        for node, neighbors in graph_adj.items():
            print(f"  {node}: {neighbors}")
        print(f"\nHeuristics:\n{node_heuristics}")
        print(f"All nodes: {graph_nodes}")

        # 2. Get start and goal nodes
        while True:
            start = input("Enter the starting node: ").strip()
            if start in graph_nodes:
                break
            else:
                print(f"Node '{start}' not found. Please try again.")

        while True:
            goal = input("Enter the goal node: ").strip()
            if goal in graph_nodes:
                break
            else:
                print(f"Node '{goal}' not found. Please try again.")

        # 3. Perform Best-First Search
        path = best_first_search_weighted(graph_adj, node_heuristics, start, goal)

        if path:
            print(f"\nSequence of visited nodes: {' -> '.join(path)}")
```

How to Run:

1. Save the code as a Python file (e.g., `bestfs_undirected_weighted_user.py`).
2. Run it from the terminal: `python bestfs_undirected_weighted_user.py`.
3. The program will ask for the number of nodes, then the name and heuristic value for each node.
4. Next, it will ask for the number of undirected weighted edges and then each edge (node1 node2 weight).
5. Finally, it will ask for the start and goal nodes.
6. The output shows the graph (with weights), heuristics, and the sequence of nodes visited by Greedy Best-First Search. Remember, the edge weights are ignored during the search prioritization.

Experiment 6: Best First Search (Undirected, Unweighted Graph, User Input)

Theory

Best-First Search (BestFS) is an informed search algorithm that uses a heuristic function to guide its exploration of a graph. It prioritizes expanding nodes that appear to be closest to the goal, based on the heuristic estimate. The specific variant used here is often called **Greedy Best-First Search**.

In this case, we are dealing with an undirected, unweighted graph. This means edges have no direction (if A is connected to B, B is connected to A) and all edges are considered to have a uniform cost (typically 1, though this cost isn't used by Greedy BestFS for prioritization).

Key Concepts:

- **Graph Representation:** An undirected, unweighted graph is usually represented using an adjacency list (e.g., a dictionary mapping each node to a list of its neighbors).
- **Heuristic Function (h(n)):** An estimate of the cost or distance from node `n` to the goal node. This value is provided by the user for each node.
- **Priority Queue:** A data structure (often implemented with a min-heap) used to store nodes to be explored. Nodes are ordered based on their heuristic values, with the lowest heuristic value having the highest priority.
- **Visited Set:** A set used to keep track of nodes that have already been visited to avoid redundant processing and infinite loops in graphs with cycles.
- **User Input:** The graph structure (nodes and undirected edges) and the heuristic value for each node are provided by the user at runtime.

How it works (Greedy Best-First Search):

1. Initialize an empty priority queue `PQ` and an empty set `visited`.

2. Get the start node and the goal node from the user.
3. Get the heuristic value `h(start_node)` for the start node.
4. Add the start node to `PQ` with its heuristic value as priority: `PQ.add(start_node, h(start_node))`.
5. While `PQ` is not empty:
    a. Extract the node with the lowest heuristic value from `PQ`. Call it `current_node`.
    b. If `current_node` is the goal node, the search is successful. Return success.
    c. If `current_node` is already in `visited`, skip it and continue to the next iteration.
    d. Mark `current_node` as visited.
    e. Process `current_node` (e.g., print it).
    f. For each neighbor of `current_node`:
        i. If the neighbor has not been visited:
            - Get the heuristic value `h(neighbor)` for the neighbor.
            - Add the neighbor to `PQ` with its heuristic value as priority: `PQ.add(neighbor, h(neighbor))`.
6. If the loop finishes (PQ becomes empty) and the goal was not reached, the goal is unreachable from the start node. Return failure.

Pseudocode/Algorithm

```
Best_First_Search(graph, heuristics, start_node, goal_node):
  Initialize an empty priority queue PQ // Stores (heuristic_value, node)
  Initialize an empty set visited

  If start_node is not in graph or goal_node is not in graph:
      Print "Start or Goal node not found in graph"
      Return Failure
  If start_node not in heuristics or goal_node not in heuristics:
      Print "Heuristic value missing for Start or Goal node"
      Return Failure

  Add (heuristics[start_node], start_node) to PQ

  While PQ is not empty:
    (h_value, current_node) = PQ.extract_min() // Get node with lowest heuristic

    If current_node is goal_node:
      Print "Goal reached!"
      // Optional: Reconstruct and print path if tracked
      Return Success

    If current_node is in visited:
      Continue

    Mark current_node as visited
    Print "Visiting node:", current_node, "(h=", h_value, ")"

    For each neighbor in graph.get_neighbors(current_node):
      If neighbor is not in visited:
        If neighbor in heuristics:
            Add (heuristics[neighbor], neighbor) to PQ
        Else:
            Print "Warning: Heuristic not found for neighbor", neighbor
            // Skip or handle as needed

  Print "Goal not reachable"
  Return Failure

Main Function:
  Initialize an empty graph (e.g., adjacency list for unweighted)
  Initialize an empty dictionary 'heuristics'
  Ask user for the number of nodes and edges

  // Get Nodes and Heuristics
  Loop for the number of nodes:
      Ask user for node name
      Ask user for heuristic value for this node
      Store in 'heuristics' dictionary
      Add node to graph keys (even if isolated initially)

  // Get Edges (Undirected, Unweighted)
  Loop for the number of edges:
      Ask user for the two nodes forming an edge (u, v)
      Add edge u -> v and v -> u to the graph

  Ask user for the starting node 'start'
  Ask user for the goal node 'goal'

  Best_First_Search(graph, heuristics, start, goal)
```

Python Code

```
import heapq
from collections import defaultdict

def build_unweighted_graph_and_heuristics_from_user():
    """Builds an undirected unweighted graph and collects heuristics from user input."""
    graph = defaultdict(list)
    heuristics = {}
    nodes = set()

    # Get nodes and their heuristic values
    while True:
        try:
            num_nodes = int(input("Enter the number of nodes: "))
            if num_nodes <= 0:
                print("Number of nodes must be positive.")
                continue
            break
        except ValueError:
            print("Invalid input. Please enter an integer.")

    print(f"Enter {num_nodes} node names and their heuristic values:")
    for i in range(num_nodes):
        while True:
            try:
                node_name = input(f"Node {i+1} name: ").strip()
                if not node_name:
                    print("Node name cannot be empty.")
                    continue
                heuristic_value = float(input(f"Heuristic value for node '{node_name}': ").strip())
                if node_name in heuristics:
                    print(f"Warning: Node '{node_name}' entered previously. Overwriting heuristic.")
                heuristics[node_name] = heuristic_value
                nodes.add(node_name)
                # Initialize node in graph even if it has no edges yet
                if node_name not in graph:
                    graph[node_name] = []
                break
            except ValueError:
                print("Invalid heuristic value. Please enter a number.")
            except Exception as e:
                print(f"An error occurred: {e}. Please try again.")

    # Get unweighted edges
    while True:
        try:
            num_edges = int(input("Enter the number of undirected unweighted edges: "))
            if num_edges < 0:
                print("Number of edges cannot be negative.")
                continue
            break
        except ValueError:
            print("Invalid input. Please enter an integer.")

    print(f"Enter {num_edges} edges (format: node1 node2, e.g., A B):")
    for i in range(num_edges):
        while True:
            try:
                edge_input = input(f"Edge {i+1}: ").strip().split()
                if len(edge_input) == 2:
                    u, v = edge_input
                    if u not in nodes or v not in nodes:
                        print(f"Error: One or both nodes ('{u}', '{v}') were not defined. Define nodes first.")
                        continue
                    # Add edge in both directions for undirected graph
                    graph[u].append(v)
```

```
                    graph[v].append(u)
                    break
                else:
                    print("Invalid format. Please enter node1 node2.")
            except Exception as e:
                print(f"An error occurred: {e}. Please try again.")

    return graph, heuristics, nodes

def best_first_search_unweighted(graph, heuristics, start_node, goal_node):
    """Performs Best-First Search (Greedy) on an unweighted graph.

    Args:
        graph (defaultdict): Adjacency list {node: [neighbor1, neighbor2, ...]}.
        heuristics (dict): {node: heuristic_value}.
        start_node (str): The starting node.
        goal_node (str): The target node.

    Returns:
        list or None: Sequence of visited nodes leading to goal, or None if not reachable.
    """
    if start_node not in heuristics or goal_node not in heuristics:
        print(f"Error: Start '{start_node}' or Goal '{goal_node}' not found or lacks heuristic.")
        return None

    priority_queue = []
    # Store tuples of (heuristic_value, node_name) in the heap
    heapq.heappush(priority_queue, (heuristics[start_node], start_node))
    visited = set()
    visited_order = [] # To track the sequence of visited nodes
    # parent = {start_node: None} # Optional: for path reconstruction

    print(f"\nBest-First Search (Greedy, Unweighted) from '{start_node}' to '{goal_node}':")

    while priority_queue:
        h_value, current_node = heapq.heappop(priority_queue)

        if current_node in visited:
            continue

        visited.add(current_node)
        visited_order.append(current_node)
        print(f"Visiting: {current_node} (h={h_value})")

        if current_node == goal_node:
            print("\nGoal reached!")
            # reconstruct_path(parent, start_node, goal_node) # Call path reconstruction if needed
            return visited_order

        # Explore neighbors, prioritize based on heuristic only
        for neighbor in graph.get(current_node, []):
            if neighbor not in visited:
                if neighbor in heuristics:
                    heapq.heappush(priority_queue, (heuristics[neighbor], neighbor))
                    # parent[neighbor] = current_node # Update parent for path reconstruction
                else:
                    print(f"Warning: Heuristic missing for neighbor '{neighbor}'. Skipping.")

    print("\nGoal not reachable.")
    return None

# --- Main Execution ---
if __name__ == "__main__":
    # 1. Build graph and heuristics
    graph_adj, node_heuristics, graph_nodes = build_unweighted_graph_and_heuristics_from_user()
```

```
    if not graph_nodes:
        print("\nNo nodes defined. Exiting.")
    else:
        print(f"\nGraph (Undirected, Unweighted Adjacency List):")
        for node, neighbors in graph_adj.items():
            print(f"  {node}: {neighbors}")
        print(f"\nHeuristics:\n{node_heuristics}")
        print(f"All nodes: {graph_nodes}")

        # 2. Get start and goal nodes
        while True:
            start = input("Enter the starting node: ").strip()
            if start in graph_nodes:
                break
            else:
                print(f"Node '{start}' not found. Please try again.")

        while True:
            goal = input("Enter the goal node: ").strip()
            if goal in graph_nodes:
                break
            else:
                print(f"Node '{goal}' not found. Please try again.")

        # 3. Perform Best-First Search
        path = best_first_search_unweighted(graph_adj, node_heuristics, start, goal)

        if path:
            print(f"\nSequence of visited nodes: {' -> '.join(path)}")
```

How to Run:

1. Save the code as a Python file (e.g., `bestfs_undirected_unweighted_user.py`).
2. Run it from the terminal: `python bestfs_undirected_unweighted_user.py`.
3. The program will ask for the number of nodes, then the name and heuristic value for each node.
4. Next, it will ask for the number of undirected unweighted edges and then each edge (node1 node2).
5. Finally, it will ask for the start and goal nodes.
6. The output shows the graph, heuristics, and the sequence of nodes visited by Greedy Best-First Search.

Experiment 7: Best First Search (Directed, Weighted Graph, User Input)

Theory

Best-First Search (BestFS) is an informed search algorithm that explores a graph by expanding the node deemed most promising based on a heuristic function `h(n)`. It uses a priority queue to manage nodes, prioritizing those with the lowest heuristic value (estimated cost to the goal).

This specific variant, often called **Greedy Best-First Search**, focuses *only* on the heuristic value `h(n)` for prioritization. Even though the graph is directed and weighted, the edge weights are *not* considered when deciding which node to explore next. The algorithm greedily chooses the path that looks best locally based on the heuristic, without regard to the actual cost accumulated so far.

Key Concepts:

- **Graph Representation:** A directed, weighted graph is typically represented using an adjacency list where each entry stores the neighbor node and the weight of the directed edge leading to it (e.g., `graph[node] = [(neighbor1, weight1), (neighbor2, weight2), ...]`).
- **Heuristic Function (h(n)):** An estimate of the cost from node `n` to the goal node. Provided by the user for each node.
- **Priority Queue:** Stores nodes to be explored, ordered by their heuristic values (lowest `h(n)` has highest priority).
- **Visited Set:** Prevents cycles and redundant exploration by keeping track of nodes already processed.
- **User Input:** The graph structure (directed weighted edges) and the heuristic value for each node are provided by the user.

How it works:

1. Initialize an empty priority queue `PQ` and an empty set `visited`.
2. Get the start node and the goal node from the user.
3. Get the heuristic value `h(start_node)` for the start node.
4. Add the start node to `PQ` with its heuristic value as priority: `PQ.add(start_node, h(start_node))`.
5. While `PQ` is not empty:
    a. Extract the node with the lowest heuristic value from `PQ`. Call it `current_node`.
    b. If `current_node` is the goal node, the search is successful. Return success.
    c. If `current_node` is already in `visited`, continue (already processed).
    d. Mark `current_node` as visited.
    e. Process `current_node` (e.g., print it).
    f. For each neighbor (and associated edge weight) of `current_node`:
        i. If the neighbor has not been visited:
            - Get the heuristic value `h(neighbor)`.
            - Add the neighbor to `PQ` with its heuristic value as priority: `PQ.add(neighbor, h(neighbor))`.
6. If the loop finishes and the goal was not reached, return failure.

Pseudocode/Algorithm

```
Best_First_Search(graph, heuristics, start_node, goal_node):
  Initialize an empty priority queue PQ // Stores (heuristic_value, node)
  Initialize an empty set visited

  If start_node not in graph or goal_node not in graph:
      Print "Start or Goal node not found in graph"
      Return Failure
  If start_node not in heuristics or goal_node not in heuristics:
      Print "Heuristic value missing for Start or Goal node"
      Return Failure

  Add (heuristics[start_node], start_node) to PQ

  While PQ is not empty:
    (h_value, current_node) = PQ.extract_min() // Get node with lowest heuristic

    If current_node is goal_node:
      Print "Goal reached!"
      Return Success

    If current_node is in visited:
      Continue

    Mark current_node as visited
    Print "Visiting node:", current_node, "(h=", h_value, ")"

    // Iterate through neighbors (ignoring weights for priority)
    For each (neighbor, weight) in graph.get_neighbors(current_node):
      If neighbor is not in visited:
        If neighbor in heuristics:
            Add (heuristics[neighbor], neighbor) to PQ
        Else:
            Print "Warning: Heuristic not found for neighbor", neighbor
            // Skip or handle as needed

  Print "Goal not reachable"
  Return Failure

Main Function:
  Initialize an empty graph (e.g., directed weighted adjacency list)
  Initialize an empty dictionary 'heuristics'
  Ask user for the number of nodes and edges

  // Get Nodes and Heuristics
  Loop for the number of nodes:
      Ask user for node name
      Ask user for heuristic value for this node
      Store in 'heuristics' dictionary
      Add node to graph keys

  // Get Edges (Directed, Weighted)
  Loop for the number of edges:
      Ask user for source, destination, and weight (u, v, w)
      Add edge u -> (v, w) to the graph

  Ask user for the starting node 'start'
  Ask user for the goal node 'goal'

  Best_First_Search(graph, heuristics, start, goal)
```

Python Code

```
import heapq
from collections import defaultdict

def build_directed_weighted_graph_and_heuristics_from_user():
    """Builds a directed weighted graph and collects heuristics from user input."""
    graph = defaultdict(list) # Stores neighbors as (neighbor, weight) tuples
    heuristics = {}
    nodes = set()

    # Get nodes and their heuristic values
    while True:
        try:
            num_nodes = int(input("Enter the number of nodes: "))
            if num_nodes <= 0:
                print("Number of nodes must be positive.")
                continue
            break
        except ValueError:
            print("Invalid input. Please enter an integer.")

    print(f"Enter {num_nodes} node names and their heuristic values:")
    for i in range(num_nodes):
        while True:
            try:
                node_name = input(f"Node {i+1} name: ").strip()
                if not node_name:
                    print("Node name cannot be empty.")
                    continue
                heuristic_value = float(input(f"Heuristic value for node '{node_name}': ").strip())
                if node_name in heuristics:
                    print(f"Warning: Node '{node_name}' entered previously. Overwriting heuristic.")
                heuristics[node_name] = heuristic_value
                nodes.add(node_name)
                # Initialize node in graph
                if node_name not in graph:
                    graph[node_name] = []
                break
            except ValueError:
                print("Invalid heuristic value. Please enter a number.")
            except Exception as e:
                print(f"An error occurred: {e}. Please try again.")

    # Get directed weighted edges
    while True:
        try:
            num_edges = int(input("Enter the number of directed weighted edges: "))
            if num_edges < 0:
                print("Number of edges cannot be negative.")
                continue
            break
        except ValueError:
            print("Invalid input. Please enter an integer.")

    print(f"Enter {num_edges} edges (format: source destination weight, e.g., A B 5 for A -> B with weight 5):")
    for i in range(num_edges):
        while True:
            try:
                edge_input = input(f"Edge {i+1}: ").strip().split()
                if len(edge_input) == 3:
                    u, v, weight_str = edge_input
                    weight = float(weight_str)
                    if u not in nodes or v not in nodes:
                        print(f"Error: Source '{u}' or Destination '{v}' not defined. Define nodes first.")
                        continue
                    if weight < 0:
```

```
                        print("Warning: Edge weights are typically non-negative, but proceeding.")
                    # Add directed edge u -> v with weight
                    graph[u].append((v, weight))
                    break
                else:
                    print("Invalid format. Please enter source destination weight.")
            except ValueError:
                print("Invalid weight. Please enter a numeric value.")
            except Exception as e:
                print(f"An error occurred: {e}. Please try again.")

    return graph, heuristics, nodes

def best_first_search_directed_weighted(graph, heuristics, start_node, goal_node):
    """Performs Best-First Search (Greedy) on a directed weighted graph.
       Note: Weights are stored but NOT used for prioritization.

    Args:
        graph (defaultdict): Adjacency list {node: [(neighbor, weight), ...]}.
        heuristics (dict): {node: heuristic_value}.
        start_node (str): The starting node.
        goal_node (str): The target node.

    Returns:
        list or None: Sequence of visited nodes leading to goal, or None if not reachable.
    """
    if start_node not in heuristics or goal_node not in heuristics:
        print(f"Error: Start '{start_node}' or Goal '{goal_node}' not found or lacks heuristic.")
        return None

    priority_queue = []
    heapq.heappush(priority_queue, (heuristics[start_node], start_node))
    visited = set()
    visited_order = []
    # parent = {start_node: None} # Optional for path reconstruction

    print(f"\nBest-First Search (Greedy, Directed, Weighted) from '{start_node}' to '{goal_node}':")

    while priority_queue:
        h_value, current_node = heapq.heappop(priority_queue)

        if current_node in visited:
            continue

        visited.add(current_node)
        visited_order.append(current_node)
        print(f"Visiting: {current_node} (h={h_value})")

        if current_node == goal_node:
            print("\nGoal reached!")
            # reconstruct_path(parent, start_node, goal_node)
            return visited_order

        # Explore neighbors, prioritize based on heuristic only
        for neighbor, weight in graph.get(current_node, []):
            if neighbor not in visited:
                if neighbor in heuristics:
                    heapq.heappush(priority_queue, (heuristics[neighbor], neighbor))
                    # parent[neighbor] = current_node
                else:
                    print(f"Warning: Heuristic missing for neighbor '{neighbor}'. Skipping.")

    print("\nGoal not reachable.")
    return None

# --- Main Execution ---
```

```
if __name__ == "__main__":
    # 1. Build graph and heuristics
    graph_adj, node_heuristics, graph_nodes = build_directed_weighted_graph_and_heuristics_from_user()

    if not graph_nodes:
        print("\nNo nodes defined. Exiting.")
    else:
        print(f"\nGraph (Directed, Weighted Adjacency List):")
        for node, neighbors in graph_adj.items():
            print(f"  {node}: {neighbors}")
        print(f"\nHeuristics:\n{node_heuristics}")
        print(f"All nodes: {graph_nodes}")

        # 2. Get start and goal nodes
        while True:
            start = input("Enter the starting node: ").strip()
            if start in graph_nodes:
                break
            else:
                print(f"Node '{start}' not found. Please try again.")

        while True:
            goal = input("Enter the goal node: ").strip()
            if goal in graph_nodes:
                break
            else:
                print(f"Node '{goal}' not found. Please try again.")

        # 3. Perform Best-First Search
        path = best_first_search_directed_weighted(graph_adj, node_heuristics, start, goal)

        if path:
            print(f"\nSequence of visited nodes: {' -> '.join(path)}")
```

How to Run:

1. Save the code as a Python file (e.g., `bestfs_directed_weighted_user.py`).
2. Run it from the terminal: `python bestfs_directed_weighted_user.py`.
3. The program will ask for the number of nodes, then the name and heuristic value for each node.
4. Next, it will ask for the number of directed weighted edges and then each edge (source destination weight).
5. Finally, it will ask for the start and goal nodes.
6. The output shows the graph (with weights), heuristics, and the sequence of nodes visited by Greedy Best-First Search. Remember, edge weights are ignored for prioritization.

Experiment 8: A* Algorithm (Directed, Weighted Graph, CSV Input)

Theory

A* (pronounced "A-star") is an informed search algorithm, widely used in pathfinding and graph traversal. It efficiently finds the least-cost path between a given initial node and one goal node (out of one or more possible goals).

A* maintains a priority queue of paths to be explored. It prioritizes paths based on a cost function `f(n) = g(n) + h(n)`:

- **g(n):** The actual cost of the path from the start node to node `n` found so far.
- **h(n):** The heuristic estimate of the cost from node `n` to the goal node. This heuristic must be **admissible** (never overestimates the actual cost) for A* to guarantee finding the optimal path. If it's also **consistent** (monotonic), A* runs more efficiently.
- **f(n):** The estimated total cost of the path through node `n`.

Key Concepts:

- **Graph Representation:** A directed, weighted graph is represented using an adjacency list where each entry stores the neighbor and the weight (cost) of the directed edge: `graph[node] = [(neighbor1, weight1), (neighbor2, weight2), ...]`).

- **Heuristic Function (h(n)):** An admissible estimate of the cost from `n` to the goal. Read from a CSV file along with the graph structure.
- **Priority Queue:** Stores nodes to visit, prioritized by their `f(n)` value (lowest `f(n)` first).
- **Cost Tracking (g(n)):** A dictionary or map stores the minimum cost found so far to reach each node from the start.
- **Visited/Closed Set:** Often implicitly handled by checking if a node's cost `g(n)` can be improved. A separate closed set can also be used to track fully processed nodes.
- **Parent Pointers:** To reconstruct the final path once the goal is reached.
- **CSV Input:** Both the graph structure (directed edges with weights) and the heuristic values for nodes are read from CSV files.

How it works:

1. Initialize: Create a priority queue `open_set`, dictionaries `g_cost` (initialized to infinity for all nodes except start), `f_cost` (calculated using `g_cost` and `h`), and `parent`.
2. Start: Set `g_cost[start_node] = 0`, calculate `f_cost[start_node] = h(start_node)`, and add `(f_cost[start_node], start_node)` to `open_set`.
3. Loop: While `open_set` is not empty:
    a. Extract the node `current_node` with the lowest `f_cost` from `open_set`.
    b. If `current_node` is the goal node, reconstruct and return the path using `parent` pointers.
    c. For each `neighbor` of `current_node` with edge weight `weight`:
        i. Calculate `tentative_g_cost = g_cost[current_node] + weight`.
        ii. If `tentative_g_cost < g_cost[neighbor]` (meaning a better path to `neighbor` is found):
            - Update `parent[neighbor] = current_node`.
            - Update `g_cost[neighbor] = tentative_g_cost`.
            - Calculate `f_cost[neighbor] = g_cost[neighbor] + h(neighbor)`.
            - If `neighbor` is not already in `open_set` (or if using a structure that allows updates), add `(f_cost[neighbor], neighbor)` to `open_set`.
4. If the loop finishes and the goal was not reached, return failure (goal unreachable).

Pseudocode/Algorithm

```
function reconstruct_path(parent, current):
    total_path = [current]
    while current in parent:
        current = parent[current]
        total_path.append(current)
    return total_path reversed

A_Star(graph, heuristics, start_node, goal_node):
    Initialize open_set as a priority queue // Stores (f_cost, node)
    Initialize parent = empty map // Stores parent pointers for path reconstruction
    Initialize g_cost = map with default value infinity
    Initialize f_cost = map with default value infinity

    g_cost[start_node] = 0
    f_cost[start_node] = heuristics[start_node]
    Add (f_cost[start_node], start_node) to open_set

    While open_set is not empty:
        (current_f, current_node) = open_set.extract_min()

        If current_node is goal_node:
            Return reconstruct_path(parent, current_node)

        // Optimization: If we extract a node already processed with a lower f_cost,
        // skip it. This depends on how duplicates are handled in the priority queue.

        For each (neighbor, weight) in graph.get_neighbors(current_node):
            tentative_g_cost = g_cost[current_node] + weight

            If tentative_g_cost < g_cost[neighbor]:
                // This path to neighbor is better than any previous one.
                parent[neighbor] = current_node
                g_cost[neighbor] = tentative_g_cost
                f_cost[neighbor] = g_cost[neighbor] + heuristics[neighbor]

                // Check if neighbor is in open_set; if not, add it.
                // If it is, update its priority if the priority queue supports it.
                // A simple approach is to just add it again; the check at extraction handles duplicates.
                Add (f_cost[neighbor], neighbor) to open_set

    Return Failure // Goal was not reached

Main Function:
    Read graph data (directed edges with weights) from graph_csv_file
    Read heuristic values from heuristics_csv_file
    Create graph representation (adjacency list)
    Store heuristics in a dictionary

    Define start_node and goal_node

    path = A_Star(graph, heuristics, start_node, goal_node)

    If path is not Failure:
        Print "Path found:", path
    Else:
        Print "Path not found"
```

Python Code

```
import csv
import heapq
from collections import defaultdict
import math # For math.inf

def read_graph_from_csv(filename):
    """Reads a directed weighted graph from a CSV file.

    Args:
        filename (str): Path to the CSV file.
                        Format: source,destination,weight

    Returns:
        tuple: (defaultdict, set) - Adjacency list and set of all nodes.
               Returns (None, None) on error.
    """
    graph = defaultdict(list)
    nodes = set()
    try:
        with open(filename, 'r', newline='') as file:
            reader = csv.reader(file)
            header = next(reader, None) # Skip header if exists
            print(f"Reading graph CSV with header: {header}")
            for i, row in enumerate(reader):
                if len(row) == 3:
                    u, v, weight_str = row[0].strip(), row[1].strip(), row[2].strip()
                    try:
                        weight = float(weight_str)
                        if weight < 0:
                            print(f"Warning: Negative weight found ({weight}) on edge {u}->{v} at row {i+2}. A* assumes non-negative
                        graph[u].append((v, weight))
                        nodes.add(u)
                        nodes.add(v)
                    except ValueError:
                        print(f"Skipping row {i+2}: Invalid weight '{weight_str}'")
                else:
                    print(f"Skipping invalid row {i+2}: {row} (Expected 3 columns)")
    except FileNotFoundError:
        print(f"Error: Graph file '{filename}' not found.")
        return None, None
    except Exception as e:
        print(f"An error occurred reading graph CSV '{filename}': {e}")
        return None, None
    # Ensure all nodes exist as keys in the graph, even if they have no outgoing edges
    for node in nodes:
        if node not in graph:
            graph[node] = []
    return graph, nodes

def read_heuristics_from_csv(filename):
    """Reads heuristic values from a CSV file.

    Args:
        filename (str): Path to the CSV file.
                        Format: node,heuristic_value

    Returns:
        dict or None: Dictionary mapping nodes to heuristic values, or None on error.
    """
    heuristics = {}
    try:
        with open(filename, 'r', newline='') as file:
            reader = csv.reader(file)
            header = next(reader, None) # Skip header
            print(f"Reading heuristics CSV with header: {header}")
```

```
            for i, row in enumerate(reader):
                if len(row) == 2:
                    node, h_val_str = row[0].strip(), row[1].strip()
                    try:
                        h_val = float(h_val_str)
                        if h_val < 0:
                             print(f"Warning: Negative heuristic found ({h_val}) for node '{node}' at row {i+2}. Heuristics must be r
                        heuristics[node] = h_val
                    except ValueError:
                        print(f"Skipping row {i+2}: Invalid heuristic value '{h_val_str}'")
                else:
                    print(f"Skipping invalid row {i+2}: {row} (Expected 2 columns)")
    except FileNotFoundError:
        print(f"Error: Heuristics file '{filename}' not found.")
        return None
    except Exception as e:
        print(f"An error occurred reading heuristics CSV '{filename}': {e}")
        return None
    return heuristics

def reconstruct_path(parent, current):
    """Reconstructs the path from start to goal using parent pointers."""
    path = [current]
    while current in parent:
        current = parent[current]
        path.append(current)
    return path[::-1] # Reverse to get start -> goal order

def a_star_search(graph, heuristics, start_node, goal_node, all_nodes):
    """Performs A* search.

    Args:
        graph (defaultdict): Adjacency list {node: [(neighbor, weight), ...]}.
        heuristics (dict): {node: heuristic_value}.
        start_node (str): The starting node.
        goal_node (str): The target node.
        all_nodes (set): Set of all nodes in the graph.

    Returns:
        list or None: The optimal path as a list of nodes, or None if no path exists.
    """
    if start_node not in all_nodes or goal_node not in all_nodes:
        print(f"Error: Start node '{start_node}' or Goal node '{goal_node}' not found in the graph nodes.")
        return None
    if start_node not in heuristics or goal_node not in heuristics:
        print(f"Error: Heuristic value missing for Start node '{start_node}' or Goal node '{goal_node}'.")
        return None

    open_set = [] # Priority queue (min-heap)
    parent = {}   # To reconstruct path {node: parent_node}

    # g_cost: cost from start to node
    g_cost = {node: math.inf for node in all_nodes}
    g_cost[start_node] = 0

    # f_cost: estimated total cost (g_cost + heuristic)
    f_cost = {node: math.inf for node in all_nodes}
    f_cost[start_node] = heuristics.get(start_node, math.inf) # Use get for safety

    heapq.heappush(open_set, (f_cost[start_node], start_node))

    print(f"\nA* Search from '{start_node}' to '{goal_node}':")

    while open_set:
        current_f, current_node = heapq.heappop(open_set)
```

```
        print(f"  Visiting: {current_node} (f={current_f:.2f}, g={g_cost[current_node]:.2f}, h={heuristics.get(current_node, 'N/A')})")

        # Optimization: If we found a shorter path already, skip
        if current_f > f_cost[current_node]:
             print(f"    (Skipping - already found shorter path to {current_node})")
             continue

        if current_node == goal_node:
            print("\nGoal reached!")
            path = reconstruct_path(parent, current_node)
            print(f"Optimal Path: {' -> '.join(path)}")
            print(f"Total Cost: {g_cost[goal_node]:.2f}")
            return path

        for neighbor, weight in graph.get(current_node, []):
            tentative_g_cost = g_cost[current_node] + weight

            if tentative_g_cost < g_cost[neighbor]:
                # Found a better path to the neighbor
                parent[neighbor] = current_node
                g_cost[neighbor] = tentative_g_cost
                h_neighbor = heuristics.get(neighbor, math.inf)
                if h_neighbor == math.inf:
                     print(f"Warning: Heuristic missing for neighbor '{neighbor}'. Assuming infinity.")
                f_cost[neighbor] = tentative_g_cost + h_neighbor
                heapq.heappush(open_set, (f_cost[neighbor], neighbor))
                print(f"    Updating neighbor {neighbor}: new g={tentative_g_cost:.2f}, f={f_cost[neighbor]:.2f}")

    print("\nGoal not reachable.")
    return None

# --- Main Execution ---
if __name__ == "__main__":
    graph_csv = 'graph_directed_weighted.csv'
    heuristics_csv = 'heuristics.csv'
    start_node = 'A' # Example start node
    goal_node = 'G'  # Example goal node

    # 1. Create dummy CSV files for demonstration
    try:
        with open(graph_csv, 'w', newline='') as f:
            writer = csv.writer(f)
            writer.writerow(['Source', 'Destination', 'Weight'])
            writer.writerow(['A', 'B', '1'])
            writer.writerow(['A', 'C', '4'])
            writer.writerow(['B', 'D', '2'])
            writer.writerow(['B', 'E', '5'])
            writer.writerow(['C', 'F', '1'])
            writer.writerow(['D', 'G', '3'])
            writer.writerow(['E', 'G', '2'])
            writer.writerow(['F', 'G', '3'])
            writer.writerow(['X', 'Y', '1']) # Disconnected part
        print(f"Created dummy graph file: '{graph_csv}'")

        with open(heuristics_csv, 'w', newline='') as f:
            writer = csv.writer(f)
            writer.writerow(['Node', 'Heuristic'])
            writer.writerow(['A', '7'])
            writer.writerow(['B', '6'])
            writer.writerow(['C', '8'])
            writer.writerow(['D', '4'])
            writer.writerow(['E', '3'])
            writer.writerow(['F', '5'])
            writer.writerow(['G', '0']) # Goal heuristic is 0
            writer.writerow(['X', '10'])
            writer.writerow(['Y', '11'])
```

```
        print(f"Created dummy heuristics file: '{heuristics_csv}'")
    except Exception as e:
        print(f"Could not create dummy CSV files: {e}")

    # 2. Read graph and heuristics
    graph, nodes = read_graph_from_csv(graph_csv)
    heuristics = read_heuristics_from_csv(heuristics_csv)

    if graph is not None and heuristics is not None and nodes:
        print(f"\nGraph Nodes: {nodes}")
        print(f"Graph Adjacency List:")
        for node, neighbors in graph.items():
            print(f"  {node}: {neighbors}")
        print(f"\nHeuristics: {heuristics}")

        # 3. Perform A* Search
        # Allow user to override start/goal if needed
        user_start = input(f"Enter start node (default '{start_node}'): ").strip() or start_node
        user_goal = input(f"Enter goal node (default '{goal_node}'): ").strip() or goal_node

        a_star_search(graph, heuristics, user_start, user_goal, nodes)
    else:
        print("\nExiting due to errors reading input files.")
```

To Run the Code:

1. Save the Python code as a `.py` file (e.g., `a_star_csv.py`).
2. The code includes functionality to create sample `graph_directed_weighted.csv` and `heuristics.csv` files in the same directory.
    - `graph_directed_weighted.csv` format: `Source,Destination,Weight` (one edge per row).
    - `heuristics.csv` format: `Node,Heuristic` (one node per row).
3. You can modify these CSV files or create your own with the specified formats.
4. Run the script from your terminal: `python a_star_csv.py`.
5. The script will prompt you to confirm or enter the start and goal nodes.
6. The output will show the graph, heuristics, the nodes visited during the A* search (with their f, g, h costs), and the final optimal path found (if any) along with its total cost.

Experiment 9: A* Algorithm (Directed, Weighted Graph, User Input)

Theory

A* (A-star) is an informed search algorithm used for finding the shortest path between nodes in a graph. It combines the strengths of Dijkstra's algorithm (which finds the shortest path based on actual cost from the start) and Greedy Best-First Search (which uses a heuristic to estimate the cost to the goal).

A* prioritizes nodes based on the evaluation function `f(n) = g(n) + h(n)`:

- **g(n):** The actual cost of the path found so far from the start node to node `n`.
- **h(n):** The heuristic estimate of the cost from node `n` to the goal node. For A* to guarantee optimality (finding the true shortest path), the heuristic must be **admissible** (it never overestimates the actual cost to the goal).
- **f(n):** The estimated total cost of the path from start to goal going through node `n`.

Key Concepts:

- **Graph Representation:** A directed, weighted graph is represented using an adjacency list: `graph[node] = [(neighbor1, weight1), (neighbor2, weight2), ...]`, where `weight` is the cost of the directed edge.
- **Heuristic Function (h(n)):** An admissible estimate of the cost from `n` to the goal. Provided by the user.
- **Priority Queue (Open Set):** Stores nodes to be explored, prioritized by their `f(n)` value (lowest `f(n)` first).
- **Cost Tracking (g(n)):** A dictionary stores the minimum cost found so far to reach each node from the start.
- **Parent Pointers:** A dictionary stores the predecessor of each node on the best path found so far, used to reconstruct the final path.
- **Closed Set (Implicit/Explicit):** Keeps track of nodes already processed to avoid redundant work. Often handled implicitly by checking if a newly found path to a node is better than a previous one.
- **User Input:** The graph structure (nodes, directed weighted edges) and heuristic values are provided by the user during runtime.

How it works:

1. Initialize: Create a priority queue `open_set`, dictionaries `g_cost` (infinity except start), `f_cost` (infinity except start), and `parent`.
2. Start: Set `g_cost[start_node] = 0`, `f_cost[start_node] = h(start_node)`, add `(f_cost[start_node], start_node)` to `open_set`.
3. Loop: While `open_set` is not empty:
    a. Extract `current_node` with the lowest `f_cost` from `open_set`.
    b. If `current_node` is the goal, reconstruct and return the path.
    c. For each `neighbor` of `current_node` with edge weight `weight`:
        i. Calculate `tentative_g_cost = g_cost[current_node] + weight`.
        ii. If `tentative_g_cost < g_cost[neighbor]`:
            - Update `parent[neighbor] = current_node`.
            - Update `g_cost[neighbor] = tentative_g_cost`.
            - Calculate `f_cost[neighbor] = g_cost[neighbor] + h(neighbor)`.
            - Add `(f_cost[neighbor], neighbor)` to `open_set` (or update if already present and priority queue supports it).
4. If loop finishes, return failure (goal unreachable).

Pseudocode/Algorithm

```
function reconstruct_path(parent, current):
    total_path = [current]
    while current in parent:
        current = parent[current]
        total_path.append(current)
    return total_path reversed

A_Star(graph, heuristics, start_node, goal_node):
    Initialize open_set as a priority queue // Stores (f_cost, node)
    Initialize parent = empty map
    Initialize g_cost = map with default value infinity
    Initialize f_cost = map with default value infinity

    g_cost[start_node] = 0
    f_cost[start_node] = heuristics[start_node]
    Add (f_cost[start_node], start_node) to open_set

    While open_set is not empty:
        (current_f, current_node) = open_set.extract_min()

        If current_node is goal_node:
            Return reconstruct_path(parent, current_node)

        // Optimization: If current_f > f_cost[current_node], skip (already found better path)
        if current_f > f_cost[current_node]:
            continue

        For each (neighbor, weight) in graph.get_neighbors(current_node):
            tentative_g_cost = g_cost[current_node] + weight

            If tentative_g_cost < g_cost[neighbor]:
                parent[neighbor] = current_node
                g_cost[neighbor] = tentative_g_cost
                f_cost[neighbor] = g_cost[neighbor] + heuristics[neighbor]
                Add (f_cost[neighbor], neighbor) to open_set

    Return Failure // Goal not reached

Main Function:
    Initialize graph, heuristics, nodes set
    Ask user for number of nodes
    Loop for number of nodes:
        Get node name and heuristic value
        Store node and heuristic

    Ask user for number of directed weighted edges
    Loop for number of edges:
        Get source, destination, weight
        Add edge to graph

    Ask user for start node and goal node

    path = A_Star(graph, heuristics, start_node, goal_node)

    If path is not Failure:
        Print "Path found:", path, "Cost:", g_cost[goal_node]
    Else:
        Print "Path not found"
```

Python Code

```
import heapq
from collections import defaultdict
import math

def build_graph_heuristics_from_user():
    """Builds directed weighted graph and heuristics from user input."""
    graph = defaultdict(list)
    heuristics = {}
    nodes = set()

    # Get nodes and heuristics
    while True:
        try:
            num_nodes = int(input("Enter the number of nodes: "))
            if num_nodes <= 0: print("Must be positive."); continue
            break
        except ValueError: print("Invalid input.")

    print(f"Enter {num_nodes} node names and their heuristic values:")
    for i in range(num_nodes):
        while True:
            try:
                node_name = input(f"Node {i+1} name: ").strip()
                if not node_name: print("Name cannot be empty."); continue
                h_val = float(input(f"Heuristic for '{node_name}': ").strip())
                if h_val < 0: print("Warning: Heuristic should be non-negative.")
                if node_name in heuristics: print(f"Warning: Overwriting heuristic for '{node_name}'.")
                heuristics[node_name] = h_val
                nodes.add(node_name)
                if node_name not in graph: graph[node_name] = [] # Ensure node exists
                break
            except ValueError: print("Invalid heuristic value.")
            except Exception as e: print(f"Error: {e}")

    # Get directed weighted edges
    while True:
        try:
            num_edges = int(input("Enter the number of directed weighted edges: "))
            if num_edges < 0: print("Cannot be negative."); continue
            break
        except ValueError: print("Invalid input.")

    print(f"Enter {num_edges} edges (format: source destination weight, e.g., A B 5):")
    for i in range(num_edges):
        while True:
            try:
                u, v, w_str = input(f"Edge {i+1}: ").strip().split()
                weight = float(w_str)
                if u not in nodes or v not in nodes:
                    print(f"Error: Node '{u}' or '{v}' not defined. Define nodes first.")
                    continue
                if weight < 0: print(f"Warning: A* assumes non-negative weights for optimality ({u}->{v}, w={weight}).")
                graph[u].append((v, weight))
                break
            except ValueError: print("Invalid format or weight. Use: node1 node2 weight")
            except Exception as e: print(f"Error: {e}")

    return graph, heuristics, nodes

def reconstruct_path(parent, current):
    path = [current]
    while current in parent:
        current = parent[current]
        path.append(current)
    return path[::-1]
```

```
def a_star_search_user(graph, heuristics, start_node, goal_node, all_nodes):
    """Performs A* search with user-provided graph/heuristics."""
    if start_node not in all_nodes or goal_node not in all_nodes:
        print(f"Error: Start '{start_node}' or Goal '{goal_node}' not in defined nodes.")
        return None
    if start_node not in heuristics or goal_node not in heuristics:
        print(f"Error: Heuristic missing for Start '{start_node}' or Goal '{goal_node}'.")
        return None

    open_set = []
    parent = {}
    g_cost = {node: math.inf for node in all_nodes}
    f_cost = {node: math.inf for node in all_nodes}

    g_cost[start_node] = 0
    h_start = heuristics.get(start_node, math.inf)
    f_cost[start_node] = h_start
    heapq.heappush(open_set, (f_cost[start_node], start_node))

    print(f"\nA* Search from '{start_node}' to '{goal_node}':")

    while open_set:
        current_f, current_node = heapq.heappop(open_set)

        print(f"  Visiting: {current_node} (f={current_f:.2f}, g={g_cost[current_node]:.2f}, h={heuristics.get(current_node, 'N/A')})")

        if current_f > f_cost[current_node]:
            print(f"    (Skipping - already found shorter path to {current_node})")
            continue

        if current_node == goal_node:
            print("\nGoal reached!")
            path = reconstruct_path(parent, current_node)
            print(f"Optimal Path: {' -> '.join(path)}")
            print(f"Total Cost: {g_cost[goal_node]:.2f}")
            return path

        for neighbor, weight in graph.get(current_node, []):
            tentative_g_cost = g_cost[current_node] + weight

            if tentative_g_cost < g_cost[neighbor]:
                parent[neighbor] = current_node
                g_cost[neighbor] = tentative_g_cost
                h_neighbor = heuristics.get(neighbor, math.inf)
                if h_neighbor == math.inf: print(f"Warning: Heuristic missing for '{neighbor}'. Assuming infinity.")
                f_cost[neighbor] = tentative_g_cost + h_neighbor
                heapq.heappush(open_set, (f_cost[neighbor], neighbor))
                print(f"    Updating neighbor {neighbor}: new g={tentative_g_cost:.2f}, f={f_cost[neighbor]:.2f}")

    print("\nGoal not reachable.")
    return None

# --- Main Execution ---
if __name__ == "__main__":
    # 1. Build graph and heuristics from user
    graph_adj, node_heuristics, graph_nodes = build_graph_heuristics_from_user()

    if not graph_nodes:
        print("\nNo nodes defined. Exiting.")
    else:
        print(f"\nGraph (Directed, Weighted Adjacency List):")
        for node, neighbors in graph_adj.items(): print(f"  {node}: {neighbors}")
        print(f"\nHeuristics: {node_heuristics}")
        print(f"All nodes: {graph_nodes}")
```

```
# 2. Get start and goal nodes
while True:
    start = input("Enter the starting node: ").strip()
    if start in graph_nodes: break
    else: print(f"Node '{start}' not found. Try again.")

while True:
    goal = input("Enter the goal node: ").strip()
    if goal in graph_nodes: break
    else: print(f"Node '{goal}' not found. Try again.")

# 3. Perform A* Search
a_star_search_user(graph_adj, node_heuristics, start, goal, graph_nodes)
```

How to Run:

1. Save the code as a Python file (e.g., `a_star_user.py`).
2. Run it from the terminal: `python a_star_user.py`.
3. The program will prompt you for:
    - Number of nodes.
    - Name and heuristic value for each node.
    - Number of directed weighted edges.
    - Source, destination, and weight for each edge.
    - Start node.
    - Goal node.
4. The output will show the graph details, the steps of the A* search (visited nodes and costs), and the final optimal path and its cost if found.

Experiment 10: A* Algorithm (Undirected, Weighted Graph, CSV Input)

Theory

A* (A-star) is an informed search algorithm renowned for finding the least-cost path in a graph. It balances the actual cost incurred from the start node (`g(n)`) with an estimated heuristic cost to the goal (`h(n)`).

For an **undirected, weighted graph**, A* works similarly to the directed case, but edges are considered traversable in both directions. The cost function remains `f(n) = g(n) + h(n)`:

- **g(n):** The actual cost (sum of edge weights) of the shortest path found so far from the start node to node `n`.
- **h(n):** The heuristic estimate of the cost from node `n` to the goal. Must be **admissible** (never overestimates the true cost) for optimality.
- **f(n):** The estimated total cost of the path from start to goal through node `n`.

Key Concepts:

- **Graph Representation:** An undirected, weighted graph is represented using an adjacency list where `graph[node]` contains tuples `(neighbor, weight)`. Since it's undirected, if `(B, w)` is in `graph[A]`, then `(A, w)` must be in `graph[B]`.
- **Heuristic Function (h(n)):** Admissible estimate of cost from `n` to goal. Read from CSV.
- **Priority Queue (Open Set):** Stores `(f_cost, node)` tuples, prioritized by `f_cost`.
- **Cost Tracking (g(n)):** Stores the minimum cost found to reach each node.
- **Parent Pointers:** Used to reconstruct the final path.
- **CSV Input:** Graph structure (undirected edges with weights) and heuristic values are read from separate CSV files.

How it works:

1. Initialize `open_set` (priority queue), `g_cost` (infinity except start=0), `f_cost` (infinity except start=h(start)), `parent` map.
2. Add `(f_cost[start_node], start_node)` to `open_set`.
3. Loop while `open_set` is not empty:
    a. Extract `current_node` with the lowest `f_cost`.
    b. If `current_node` is the goal, reconstruct and return the path.
    c. For each `neighbor` of `current_node` with edge weight `weight`:
        i. Calculate `tentative_g_cost = g_cost[current_node] + weight`.
        ii. If `tentative_g_cost < g_cost[neighbor]`:

- Update `parent[neighbor] = current_node`.
- Update `g_cost[neighbor] = tentative_g_cost`.
- Calculate `f_cost[neighbor] = g_cost[neighbor] + h(neighbor)`.
- Add `(f_cost[neighbor], neighbor)` to `open_set`.

4. If loop finishes, return failure.

Pseudocode/Algorithm

```
function reconstruct_path(parent, current):
    total_path = [current]
    while current in parent:
        current = parent[current]
        total_path.append(current)
    return total_path reversed

A_Star(graph, heuristics, start_node, goal_node):
    Initialize open_set as a priority queue // Stores (f_cost, node)
    Initialize parent = empty map
    Initialize g_cost = map with default value infinity
    Initialize f_cost = map with default value infinity

    g_cost[start_node] = 0
    f_cost[start_node] = heuristics[start_node]
    Add (f_cost[start_node], start_node) to open_set

    While open_set is not empty:
        (current_f, current_node) = open_set.extract_min()

        If current_node is goal_node:
            Return reconstruct_path(parent, current_node)

        // Optimization: Skip if already found a better path
        if current_f > f_cost[current_node]:
            continue

        For each (neighbor, weight) in graph.get_neighbors(current_node):
            tentative_g_cost = g_cost[current_node] + weight

            If tentative_g_cost < g_cost[neighbor]:
                parent[neighbor] = current_node
                g_cost[neighbor] = tentative_g_cost
                f_cost[neighbor] = g_cost[neighbor] + heuristics[neighbor]
                Add (f_cost[neighbor], neighbor) to open_set

    Return Failure // Goal not reached

Main Function:
    Read graph data (undirected edges, weights) from graph_csv
    Read heuristics from heuristics_csv
    Create graph representation (adjacency list, ensuring bidirectionality)
    Store heuristics

    Define start_node and goal_node

    path = A_Star(graph, heuristics, start_node, goal_node)

    If path is not Failure:
        Print "Path found:", path, "Cost:", g_cost[goal_node]
    Else:
        Print "Path not found"
```

Python Code

```
import csv
import heapq
from collections import defaultdict
import math

def read_undirected_weighted_graph_csv(filename):
    """Reads an undirected weighted graph from CSV.

    Args:
        filename (str): Path to CSV. Format: node1,node2,weight

    Returns:
        tuple: (defaultdict, set) - Adjacency list and nodes set, or (None, None).
    """
    graph = defaultdict(list)
    nodes = set()
    try:
        with open(filename, 'r', newline='') as file:
            reader = csv.reader(file)
            header = next(reader, None)
            print(f"Reading graph CSV with header: {header}")
            for i, row in enumerate(reader):
                if len(row) == 3:
                    u, v, w_str = row[0].strip(), row[1].strip(), row[2].strip()
                    try:
                        weight = float(w_str)
                        if weight < 0:
                            print(f"Warning: Negative weight ({weight}) on edge {u}-{v} at row {i+2}. A* assumes non-negative weights
                        graph[u].append((v, weight))
                        graph[v].append((u, weight)) # Add edge in both directions
                        nodes.add(u)
                        nodes.add(v)
                    except ValueError:
                        print(f"Skipping row {i+2}: Invalid weight '{w_str}'")
                else:
                    print(f"Skipping invalid row {i+2}: {row}")
    except FileNotFoundError:
        print(f"Error: Graph file '{filename}' not found.")
        return None, None
    except Exception as e:
        print(f"Error reading graph CSV '{filename}': {e}")
        return None, None
    for node in nodes:
        if node not in graph: graph[node] = []
    return graph, nodes

def read_heuristics_from_csv(filename):
    """Reads heuristic values from a CSV file.

    Args:
        filename (str): Path to the CSV file. Format: node,heuristic_value

    Returns:
        dict or None: Dictionary {node: heuristic}, or None on error.
    """
    heuristics = {}
    try:
        with open(filename, 'r', newline='') as file:
            reader = csv.reader(file)
            header = next(reader, None)
            print(f"Reading heuristics CSV with header: {header}")
            for i, row in enumerate(reader):
                if len(row) == 2:
                    node, h_str = row[0].strip(), row[1].strip()
                    try:
```

```
                        h_val = float(h_str)
                        if h_val < 0: print(f"Warning: Negative heuristic ({h_val}) for '{node}' at row {i+2}.")
                        heuristics[node] = h_val
                    except ValueError:
                        print(f"Skipping row {i+2}: Invalid heuristic '{h_str}'")
                else:
                    print(f"Skipping invalid row {i+2}: {row}")
    except FileNotFoundError:
        print(f"Error: Heuristics file '{filename}' not found.")
        return None
    except Exception as e:
        print(f"Error reading heuristics CSV '{filename}': {e}")
        return None
    return heuristics

def reconstruct_path(parent, current):
    path = [current]
    while current in parent:
        current = parent[current]
        path.append(current)
    return path[::-1]

def a_star_search_undirected(graph, heuristics, start_node, goal_node, all_nodes):
    """Performs A* search on an undirected weighted graph."""
    if start_node not in all_nodes or goal_node not in all_nodes:
        print(f"Error: Start '{start_node}' or Goal '{goal_node}' not in graph nodes.")
        return None, math.inf
    if start_node not in heuristics or goal_node not in heuristics:
        print(f"Error: Heuristic missing for Start '{start_node}' or Goal '{goal_node}'.")
        return None, math.inf

    open_set = []
    parent = {}
    g_cost = {node: math.inf for node in all_nodes}
    f_cost = {node: math.inf for node in all_nodes}

    g_cost[start_node] = 0
    h_start = heuristics.get(start_node, math.inf)
    f_cost[start_node] = h_start
    heapq.heappush(open_set, (f_cost[start_node], start_node))

    print(f"\nA* Search (Undirected) from '{start_node}' to '{goal_node}':")

    while open_set:
        current_f, current_node = heapq.heappop(open_set)

        print(f"  Visiting: {current_node} (f={current_f:.2f}, g={g_cost[current_node]:.2f}, h={heuristics.get(current_node, 'N/A')})")

        if current_f > f_cost[current_node]:
            print(f"    (Skipping - already found shorter path to {current_node})")
            continue

        if current_node == goal_node:
            print("\nGoal reached!")
            path = reconstruct_path(parent, current_node)
            print(f"Optimal Path: {' -> '.join(path)}")
            print(f"Total Cost: {g_cost[goal_node]:.2f}")
            return path, g_cost[goal_node]

        for neighbor, weight in graph.get(current_node, []):
            tentative_g_cost = g_cost[current_node] + weight

            if tentative_g_cost < g_cost[neighbor]:
                parent[neighbor] = current_node
                g_cost[neighbor] = tentative_g_cost
                h_neighbor = heuristics.get(neighbor, math.inf)
```

```
                if h_neighbor == math.inf: print(f"Warning: Heuristic missing for '{neighbor}'. Assuming infinity.")
                f_cost[neighbor] = tentative_g_cost + h_neighbor
                heapq.heappush(open_set, (f_cost[neighbor], neighbor))
                print(f"    Updating neighbor {neighbor}: new g={tentative_g_cost:.2f}, f={f_cost[neighbor]:.2f}")

    print("\nGoal not reachable.")
    return None, math.inf

# --- Main Execution ---
if __name__ == "__main__":
    graph_csv = 'graph_undirected_weighted.csv'
    heuristics_csv = 'heuristics_undir.csv' # Can be same as directed if applicable
    start_node = 'A'
    goal_node = 'G'

    # 1. Create dummy CSV files
    try:
        with open(graph_csv, 'w', newline='') as f:
            writer = csv.writer(f)
            writer.writerow(['Node1', 'Node2', 'Weight'])
            writer.writerow(['A', 'B', '1'])
            writer.writerow(['A', 'C', '4'])
            writer.writerow(['B', 'D', '2'])
            writer.writerow(['B', 'E', '5'])
            writer.writerow(['C', 'F', '1'])
            writer.writerow(['D', 'G', '3'])
            writer.writerow(['E', 'G', '2'])
            writer.writerow(['F', 'G', '3'])
            writer.writerow(['B', 'C', '2']) # Extra edge for undirected example
            writer.writerow(['X', 'Y', '1'])
        print(f"Created dummy graph file: '{graph_csv}'")

        with open(heuristics_csv, 'w', newline='') as f:
            writer = csv.writer(f)
            writer.writerow(['Node', 'Heuristic'])
            writer.writerow(['A', '7'])
            writer.writerow(['B', '6'])
            writer.writerow(['C', '5']) # Heuristics might differ from directed case
            writer.writerow(['D', '4'])
            writer.writerow(['E', '3'])
            writer.writerow(['F', '4'])
            writer.writerow(['G', '0'])
            writer.writerow(['X', '10'])
            writer.writerow(['Y', '11'])
        print(f"Created dummy heuristics file: '{heuristics_csv}'")
    except Exception as e:
        print(f"Could not create dummy CSV files: {e}")

    # 2. Read graph and heuristics
    graph, nodes = read_undirected_weighted_graph_csv(graph_csv)
    heuristics = read_heuristics_from_csv(heuristics_csv)

    if graph is not None and heuristics is not None and nodes:
        print(f"\nGraph Nodes: {nodes}")
        print(f"Graph Adjacency List (Undirected, Weighted):")
        for node, neighbors in graph.items(): print(f"  {node}: {neighbors}")
        print(f"\nHeuristics: {heuristics}")

        # 3. Perform A* Search
        user_start = input(f"Enter start node (default '{start_node}'): ").strip() or start_node
        user_goal = input(f"Enter goal node (default '{goal_node}'): ").strip() or goal_node

        path, cost = a_star_search_undirected(graph, heuristics, user_start, user_goal, nodes)
        # Output handled within the function
    else:
```

```
        print("\nExiting due to errors reading input files.")
```

To Run the Code:

1. Save the Python code (e.g., `a_star_undir_csv.py`).
2. The script creates sample `graph_undirected_weighted.csv` and `heuristics_undir.csv` files.
    - `graph_undirected_weighted.csv` format: `Node1,Node2,Weight` (one edge per row, represents connection in both directions).
    - `heuristics_undir.csv` format: `Node,Heuristic`.
3. Modify these CSVs or create your own.
4. Run from terminal: `python a_star_undir_csv.py`.
5. Enter start and goal nodes when prompted.
6. The output shows the graph, heuristics, A* search steps, and the optimal path/cost if found.

Experiment 11: A* Algorithm (Undirected, Weighted Graph, User Input)

Theory

A* (A-star) is a pathfinding algorithm known for its efficiency and optimality (finding the least-cost path). It works on a graph by combining the actual cost from the start node (`g(n)`) with a heuristic estimate to the goal node (`h(n)`).

For an **undirected, weighted graph**, edges connect nodes in both directions with a specific cost (weight). A* uses the evaluation function `f(n) = g(n) + h(n)`:

- **g(n):** The actual cost (sum of weights) of the best path found so far from the start node to `n`.
- **h(n):** The heuristic estimate of the cost from `n` to the goal. Must be **admissible** (never overestimates the true cost) for A* to guarantee finding the optimal path.
- **f(n):** The estimated total cost of the path from start to goal passing through `n`.

Key Concepts:

- **Graph Representation:** Adjacency list for an undirected, weighted graph: `graph[node] = [(neighbor1, weight1), (neighbor2, weight2), ...]`. If `(B, w)` is in `graph[A]`, then `(A, w)` is in `graph[B]`.
- **Heuristic Function (h(n)):** Admissible estimate provided by the user.
- **Priority Queue (Open Set):** Stores `(f_cost, node)` tuples, ordered by `f_cost`.
- **Cost Tracking (g(n)):** Stores the minimum cost found to reach each node.
- **Parent Pointers:** Used to reconstruct the final path.
- **User Input:** Graph structure (nodes, undirected weighted edges) and heuristic values are provided by the user.

How it works:

1. Initialize `open_set` (priority queue), `g_cost` (infinity except start=0), `f_cost` (infinity except start=h(start)), `parent` map.
2. Add `(f_cost[start_node], start_node)` to `open_set`.
3. Loop while `open_set` is not empty:
    a. Extract `current_node` with the lowest `f_cost`.
    b. If `current_node` is the goal, reconstruct and return the path.
    c. For each `neighbor` of `current_node` with edge weight `weight`:
        i. Calculate `tentative_g_cost = g_cost[current_node] + weight`.
        ii. If `tentative_g_cost < g_cost[neighbor]`:
            - Update `parent[neighbor] = current_node`.
            - Update `g_cost[neighbor] = tentative_g_cost`.
            - Calculate `f_cost[neighbor] = g_cost[neighbor] + h(neighbor)`.
            - Add `(f_cost[neighbor], neighbor)` to `open_set`.
4. If loop finishes, return failure.

Pseudocode/Algorithm

```
function reconstruct_path(parent, current):
    total_path = [current]
    while current in parent:
        current = parent[current]
        total_path.append(current)
    return total_path reversed

A_Star(graph, heuristics, start_node, goal_node):
    Initialize open_set as a priority queue // Stores (f_cost, node)
    Initialize parent = empty map
    Initialize g_cost = map with default value infinity
    Initialize f_cost = map with default value infinity

    g_cost[start_node] = 0
    f_cost[start_node] = heuristics[start_node]
    Add (f_cost[start_node], start_node) to open_set

    While open_set is not empty:
        (current_f, current_node) = open_set.extract_min()

        If current_node is goal_node:
            Return reconstruct_path(parent, current_node)

        // Optimization: Skip if already found a better path
        if current_f > f_cost[current_node]:
            continue

        For each (neighbor, weight) in graph.get_neighbors(current_node):
            tentative_g_cost = g_cost[current_node] + weight

            If tentative_g_cost < g_cost[neighbor]:
                parent[neighbor] = current_node
                g_cost[neighbor] = tentative_g_cost
                f_cost[neighbor] = g_cost[neighbor] + heuristics[neighbor]
                Add (f_cost[neighbor], neighbor) to open_set

    Return Failure // Goal not reached

Main Function:
    Initialize graph, heuristics, nodes set
    Ask user for number of nodes
    Loop for number of nodes:
        Get node name and heuristic value
        Store node and heuristic

    Ask user for number of undirected weighted edges
    Loop for number of edges:
        Get node1, node2, weight
        Add edge (node1, node2, weight) and (node2, node1, weight) to graph

    Ask user for start node and goal node

    path = A_Star(graph, heuristics, start_node, goal_node)

    If path is not Failure:
        Print "Path found:", path, "Cost:", g_cost[goal_node]
    Else:
        Print "Path not found"
```

Python Code

```
import heapq
from collections import defaultdict
import math

def build_undirected_graph_heuristics_from_user():
    """Builds undirected weighted graph and heuristics from user input."""
    graph = defaultdict(list)
    heuristics = {}
    nodes = set()

    # Get nodes and heuristics
    while True:
        try:
            num_nodes = int(input("Enter the number of nodes: "))
            if num_nodes <= 0: print("Must be positive."); continue
            break
        except ValueError: print("Invalid input.")

    print(f"Enter {num_nodes} node names and their heuristic values:")
    for i in range(num_nodes):
        while True:
            try:
                node_name = input(f"Node {i+1} name: ").strip()
                if not node_name: print("Name cannot be empty."); continue
                h_val = float(input(f"Heuristic for '{node_name}': ").strip())
                if h_val < 0: print("Warning: Heuristic should be non-negative.")
                if node_name in heuristics: print(f"Warning: Overwriting heuristic for '{node_name}'.")
                heuristics[node_name] = h_val
                nodes.add(node_name)
                if node_name not in graph: graph[node_name] = []
                break
            except ValueError: print("Invalid heuristic value.")
            except Exception as e: print(f"Error: {e}")

    # Get undirected weighted edges
    while True:
        try:
            num_edges = int(input("Enter the number of undirected weighted edges: "))
            if num_edges < 0: print("Cannot be negative."); continue
            break
        except ValueError: print("Invalid input.")

    print(f"Enter {num_edges} edges (format: node1 node2 weight, e.g., A B 5):")
    for i in range(num_edges):
        while True:
            try:
                u, v, w_str = input(f"Edge {i+1}: ").strip().split()
                weight = float(w_str)
                if u not in nodes or v not in nodes:
                    print(f"Error: Node '{u}' or '{v}' not defined. Define nodes first.")
                    continue
                if weight < 0: print(f"Warning: A* assumes non-negative weights ({u}-{v}, w={weight}).")
                graph[u].append((v, weight))
                graph[v].append((u, weight)) # Add edge in both directions
                break
            except ValueError: print("Invalid format or weight. Use: node1 node2 weight")
            except Exception as e: print(f"Error: {e}")

    return graph, heuristics, nodes

def reconstruct_path(parent, current):
    path = [current]
    while current in parent:
        current = parent[current]
        path.append(current)
```

```
    return path[::-1]

def a_star_search_undirected_user(graph, heuristics, start_node, goal_node, all_nodes):
    """Performs A* search on user-provided undirected weighted graph."""
    if start_node not in all_nodes or goal_node not in all_nodes:
        print(f"Error: Start '{start_node}' or Goal '{goal_node}' not in defined nodes.")
        return None, math.inf
    if start_node not in heuristics or goal_node not in heuristics:
        print(f"Error: Heuristic missing for Start '{start_node}' or Goal '{goal_node}'.")
        return None, math.inf

    open_set = []
    parent = {}
    g_cost = {node: math.inf for node in all_nodes}
    f_cost = {node: math.inf for node in all_nodes}

    g_cost[start_node] = 0
    h_start = heuristics.get(start_node, math.inf)
    f_cost[start_node] = h_start
    heapq.heappush(open_set, (f_cost[start_node], start_node))

    print(f"\nA* Search (Undirected) from '{start_node}' to '{goal_node}':")

    while open_set:
        current_f, current_node = heapq.heappop(open_set)

        print(f"  Visiting: {current_node} (f={current_f:.2f}, g={g_cost[current_node]:.2f}, h={heuristics.get(current_node, 'N/A')})")

        if current_f > f_cost[current_node]:
            print(f"    (Skipping - already found shorter path to {current_node})")
            continue

        if current_node == goal_node:
            print("\nGoal reached!")
            path = reconstruct_path(parent, current_node)
            print(f"Optimal Path: {' -> '.join(path)}")
            print(f"Total Cost: {g_cost[goal_node]:.2f}")
            return path, g_cost[goal_node]

        for neighbor, weight in graph.get(current_node, []):
            tentative_g_cost = g_cost[current_node] + weight

            if tentative_g_cost < g_cost[neighbor]:
                parent[neighbor] = current_node
                g_cost[neighbor] = tentative_g_cost
                h_neighbor = heuristics.get(neighbor, math.inf)
                if h_neighbor == math.inf: print(f"Warning: Heuristic missing for '{neighbor}'. Assuming infinity.")
                f_cost[neighbor] = tentative_g_cost + h_neighbor
                heapq.heappush(open_set, (f_cost[neighbor], neighbor))
                print(f"    Updating neighbor {neighbor}: new g={tentative_g_cost:.2f}, f={f_cost[neighbor]:.2f}")

    print("\nGoal not reachable.")
    return None, math.inf

# --- Main Execution ---
if __name__ == "__main__":
    # 1. Build graph and heuristics from user
    graph_adj, node_heuristics, graph_nodes = build_undirected_graph_heuristics_from_user()

    if not graph_nodes:
        print("\nNo nodes defined. Exiting.")
    else:
        print(f"\nGraph (Undirected, Weighted Adjacency List):")
        for node, neighbors in graph_adj.items(): print(f"  {node}: {neighbors}")
        print(f"\nHeuristics: {node_heuristics}")
        print(f"All nodes: {graph_nodes}")
```

```
# 2. Get start and goal nodes
while True:
    start = input("Enter the starting node: ").strip()
    if start in graph_nodes: break
    else: print(f"Node '{start}' not found. Try again.")

while True:
    goal = input("Enter the goal node: ").strip()
    if goal in graph_nodes: break
    else: print(f"Node '{goal}' not found. Try again.")

# 3. Perform A* Search
path, cost = a_star_search_undirected_user(graph_adj, node_heuristics, start, goal, graph_nodes)
# Output handled within the function
```

How to Run:

1. Save the code as a Python file (e.g., `a_star_undir_user.py`).
2. Run it from the terminal: `python a_star_undir_user.py`.
3. The program will prompt you for:
    - Number of nodes.
    - Name and heuristic value for each node.
    - Number of undirected weighted edges.
    - Node1, Node2, and weight for each edge.
    - Start node.
    - Goal node.
4. The output will show the graph details, the steps of the A* search, and the final optimal path and its cost if found.

Experiment 12: Fuzzy Set Operations (Union, Intersection, Complement)

Theory

Fuzzy set theory, introduced by Lotfi Zadeh in 1965, deals with sets whose elements have degrees of membership. Unlike classical (crisp) sets where an element either belongs or does not belong to a set, in fuzzy sets, elements can belong to a set to a certain degree, typically represented by a membership value between 0 and 1.

- **Fuzzy Set:** A fuzzy set A in a universe of discourse U is characterized by a membership function Î¼_A(x) which maps each element x in U to a real number in the interval [0, 1]. The value Î¼_A(x) represents the "degree of membership" of x in A.
    - Î¼_A(x) = 1 means x fully belongs to A.
    - Î¼_A(x) = 0 means x does not belong to A at all.
    - 0 < Î¼_A(x) < 1 means x partially belongs to A.
- **Representation:** Fuzzy sets are often represented as a set of ordered pairs: A = {(x, Î¼_A(x)) | x âˆˆ U}.

Standard Fuzzy Set Operations:

1. **Complement (Â¬A or A'):** The complement of a fuzzy set A represents the degree to which elements *do not* belong to A.
    - Membership function: Î¼_Â¬A(x) = 1 - Î¼_A(x) for all x âˆˆ U.
2. **Intersection (A â©ˆ B):** The intersection of two fuzzy sets A and B represents the degree to which elements belong to *both* A and B. The standard intersection is defined using the minimum (min) operator.
    - Membership function: Î¼_(A â©ˆ B)(x) = min(Î¼_A(x), Î¼_B(x)) for all x âˆˆ U.
    - Other t-norms can also be used (e.g., algebraic product: Î¼_A(x) * Î¼_B(x)).
3. **Union (A âˆª B):** The union of two fuzzy sets A and B represents the degree to which elements belong to *either* A or B (or both). The standard union is defined using the maximum (max) operator.
    - Membership function: Î¼_(A âˆª B)(x) = max(Î¼_A(x), Î¼_B(x)) for all x âˆˆ U.
    - Other t-conorms (s-norms) can also be used (e.g., algebraic sum: Î¼_A(x) + Î¼_B(x) - Î¼_A(x) * Î¼_B(x)).

Universe of Discourse (U): The set of all possible elements relevant to a particular problem.

Pseudocode/Algorithm

Let A, B, C be fuzzy sets defined over a universe U. Assume A, B, C are represented as dictionaries mapping elements x âˆˆ U to membership values Î¼(x).

```
// Ensure all sets are defined over the same explicit or implicit universe U
// Get all unique elements from the domains of A, B, C to form the effective universe
Universe = keys(A) â^ª keys(B) â^ª keys(C)

Function Fuzzy_Complement(Set A, Universe U):
  Complement_Set = empty dictionary
  For each element x in U:
    membership_A = Get_Membership(A, x) // Returns 0 if x not in A's keys
    Complement_Set[x] = 1.0 - membership_A
  Return Complement_Set

Function Fuzzy_Intersection(Set A, Set B, Universe U):
  Intersection_Set = empty dictionary
  For each element x in U:
    membership_A = Get_Membership(A, x)
    membership_B = Get_Membership(B, x)
    Intersection_Set[x] = min(membership_A, membership_B)
  Return Intersection_Set

Function Fuzzy_Union(Set A, Set B, Universe U):
  Union_Set = empty dictionary
  For each element x in U:
    membership_A = Get_Membership(A, x)
    membership_B = Get_Membership(B, x)
    Union_Set[x] = max(membership_A, membership_B)
  Return Union_Set

// Helper function to handle elements potentially missing from a set's definition
Function Get_Membership(Set S, element x):
  If x is in keys(S):
    Return S[x]
  Else:
    Return 0.0 // Assume 0 membership if not explicitly defined

Main:
  Define Fuzzy Set A (e.g., A = {'a': 0.8, 'b': 0.5})
  Define Fuzzy Set B (e.g., B = {'a': 0.4, 'b': 0.9, 'c': 0.6})
  Define Fuzzy Set C (e.g., C = {'b': 0.2, 'c': 0.7, 'd': 1.0})

  // Determine the universe from all keys
  U = keys(A) â^ª keys(B) â^ª keys(C)

  // Calculate and display complements
  Complement_A = Fuzzy_Complement(A, U)
  Complement_B = Fuzzy_Complement(B, U)
  Complement_C = Fuzzy_Complement(C, U)
  Print "Complement A:", Complement_A
  Print "Complement B:", Complement_B
  Print "Complement C:", Complement_C

  // Calculate and display intersections
  Intersection_AB = Fuzzy_Intersection(A, B, U)
  Intersection_AC = Fuzzy_Intersection(A, C, U)
  Intersection_BC = Fuzzy_Intersection(B, C, U)
  Intersection_ABC = Fuzzy_Intersection(Intersection_AB, C, U)
  Print "Intersection A â^© B:", Intersection_AB
  Print "Intersection A â^© C:", Intersection_AC
  Print "Intersection B â^© C:", Intersection_BC
  Print "Intersection A â^© B â^© C:", Intersection_ABC

  // Calculate and display unions
  Union_AB = Fuzzy_Union(A, B, U)
  Union_AC = Fuzzy_Union(A, C, U)
  Union_BC = Fuzzy_Union(B, C, U)
  Union_ABC = Fuzzy_Union(Union_AB, C, U)
```

```
Print "Union A â^ª B:", Union_AB
Print "Union A â^ª C:", Union_AC
Print "Union B â^ª C:", Union_BC
Print "Union A â^ª B â^ª C:", Union_ABC
```

Python Code

```
def get_membership(fuzzy_set, element):
    """Helper to get membership value, returns 0 if element not in set."""
    return fuzzy_set.get(element, 0.0)

def fuzzy_complement(fuzzy_set, universe):
    """Calculates the complement of a fuzzy set."""
    complement_set = {}
    for element in universe:
        complement_set[element] = 1.0 - get_membership(fuzzy_set, element)
    return complement_set

def fuzzy_intersection(set_a, set_b, universe):
    """Calculates the intersection of two fuzzy sets using min operator."""
    intersection_set = {}
    for element in universe:
        membership_a = get_membership(set_a, element)
        membership_b = get_membership(set_b, element)
        intersection_set[element] = min(membership_a, membership_b)
    return intersection_set

def fuzzy_union(set_a, set_b, universe):
    """Calculates the union of two fuzzy sets using max operator."""
    union_set = {}
    for element in universe:
        membership_a = get_membership(set_a, element)
        membership_b = get_membership(set_b, element)
        union_set[element] = max(membership_a, membership_b)
    return union_set

def print_fuzzy_set(name, fuzzy_set):
    """Prints a fuzzy set in a readable format."""
    print(f"{name}: {{", end='')
    items = [f"'{k}': {v:.2f}" for k, v in sorted(fuzzy_set.items())]
    print(', '.join(items), end='')
    print("}")

# --- Main Execution ---
if __name__ == "__main__":
    # Define 3 fuzzy sets (represented as dictionaries)
    # Example: Fuzzy sets representing 'Young', 'MiddleAged', 'Old' over an age universe
    # Let's use a simpler abstract example:
    set_A = {'a': 0.2, 'b': 0.7, 'c': 0.5, 'd': 0.0}
    set_B = {'a': 0.8, 'b': 0.3, 'c': 0.9, 'e': 0.4}
    set_C = {'b': 0.5, 'c': 0.1, 'd': 1.0, 'e': 0.6, 'f': 0.7}

    print("--- Original Fuzzy Sets ---")
    print_fuzzy_set("Set A", set_A)
    print_fuzzy_set("Set B", set_B)
    print_fuzzy_set("Set C", set_C)

    # Determine the universe of discourse from all elements in the sets
    universe = set(set_A.keys()) | set(set_B.keys()) | set(set_C.keys())
    print(f"\nUniverse of Discourse (U): {sorted(list(universe))}")

    # --- Complement Operations ---
    print("\n--- Complement Operations ---")
    comp_A = fuzzy_complement(set_A, universe)
    comp_B = fuzzy_complement(set_B, universe)
    comp_C = fuzzy_complement(set_C, universe)
    print_fuzzy_set("Complement A (Â¬A)", comp_A)
    print_fuzzy_set("Complement B (Â¬B)", comp_B)
    print_fuzzy_set("Complement C (Â¬C)", comp_C)

    # --- Intersection Operations ---
    print("\n--- Intersection Operations (min) ---")
```

```
inter_AB = fuzzy_intersection(set_A, set_B, universe)
inter_AC = fuzzy_intersection(set_A, set_C, universe)
inter_BC = fuzzy_intersection(set_B, set_C, universe)
# Intersection of all three: (A â^© B) â^© C
inter_ABC = fuzzy_intersection(inter_AB, set_C, universe)

print_fuzzy_set("Intersection A â^© B", inter_AB)
print_fuzzy_set("Intersection A â^© C", inter_AC)
print_fuzzy_set("Intersection B â^© C", inter_BC)
print_fuzzy_set("Intersection A â^© B â^© C", inter_ABC)

# --- Union Operations ---
print("\n--- Union Operations (max) ---")
union_AB = fuzzy_union(set_A, set_B, universe)
union_AC = fuzzy_union(set_A, set_C, universe)
union_BC = fuzzy_union(set_B, set_C, universe)
# Union of all three: (A â^ª B) â^ª C
union_ABC = fuzzy_union(union_AB, set_C, universe)

print_fuzzy_set("Union A â^ª B", union_AB)
print_fuzzy_set("Union A â^ª C", union_AC)
print_fuzzy_set("Union B â^ª C", union_BC)
print_fuzzy_set("Union A â^ª B â^ª C", union_ABC)
```

Explanation:

1. **Representation:** Fuzzy sets `set_A`, `set_B`, and `set_C` are defined as Python dictionaries where keys are elements of the universe and values are their membership degrees.
2. **Universe:** The code first determines the complete universe of discourse by combining all unique keys from the defined sets.
3. **Helper `get_membership`:** This function safely retrieves the membership value for an element, returning 0.0 if the element isn't explicitly in the set's dictionary.
4. **`fuzzy_complement`:** Implements Î¼_Â¬A(x) = 1 - Î¼_A(x) for each element in the universe.
5. **`fuzzy_intersection`:** Implements Î¼_(A â^© B)(x) = min(Î¼_A(x), Î¼_B(x)) for each element.
6. **`fuzzy_union`:** Implements Î¼_(A â^ª B)(x) = max(Î¼_A(x), Î¼_B(x)) for each element.
7. **Demonstration:** The code calculates and prints the complements of all three sets, the pairwise intersections (Aâ^©B, Aâ^©C, Bâ^©C), the intersection of all three (Aâ^©Bâ^©C), the pairwise unions (Aâ^ªB, Aâ^ªC, Bâ^ªC), and the union of all three (Aâ^ªBâ^ªC).
8. **Output:** The results clearly show the membership degrees for each element in the resulting fuzzy sets for each operation.

Experiment 13: Fuzzy Set Operations - De Morganâ€™s Law (Complement of Union)

Theory

Fuzzy set theory extends classical set theory to handle degrees of membership. Elements belong to fuzzy sets with a membership value between 0 and 1.

Basic Operations (Standard Definitions):

- **Complement (Â¬A):** Î¼_Â¬A(x) = 1 - Î¼_A(x)
- **Intersection (A â^© B):** Î¼_(A â^© B)(x) = min(Î¼_A(x), Î¼_B(x))
- **Union (A â^ª B):** Î¼_(A â^ª B)(x) = max(Î¼_A(x), Î¼_B(x))

De Morgan's Laws in Fuzzy Logic:

De Morgan's laws, fundamental principles in classical set theory and logic, also hold true for standard fuzzy set operations (complement, min-intersection, max-union).

This experiment focuses on the first De Morgan's law for fuzzy sets:

Â¬(A â^ª B) = Â¬A â^© Â¬B

This law states that the complement of the union of two fuzzy sets A and B is equal to the intersection of their complements.

- **Left-Hand Side (LHS): Â¬(A â^ª B)**
    1. First, find the union of A and B: Î¼_(A â^ª B)(x) = max(Î¼_A(x), Î¼_B(x)).

2. Then, find the complement of the result: Î¼_Â¬(A âˆª B)(x) = 1 - Î¼_(A âˆª B)(x) = 1 - max(Î¼_A(x), Î¼_B(x)).

- **Right-Hand Side (RHS): Â¬A âˆ© Â¬B**

1. First, find the complement of A: Î¼_Â¬A(x) = 1 - Î¼_A(x).
2. First, find the complement of B: Î¼_Â¬B(x) = 1 - Î¼_B(x).
3. Then, find the intersection of the complements: Î¼_(Â¬A âˆ© Â¬B)(x) = min(Î¼_Â¬A(x), Î¼_Â¬B(x)) = min(1 - Î¼_A(x), 1 - Î¼_B(x)).

De Morgan's law confirms that these two calculations yield the same membership function for all elements x in the universe.

Pseudocode/Algorithm

Let A, B be fuzzy sets defined over a universe U. Assume A, B are represented as dictionaries mapping elements x â€ U to membership values Î¼(x).

```
// Helper functions (from Experiment 12)
Function Get_Membership(Set S, element x)
Function Fuzzy_Complement(Set S, Universe U)
Function Fuzzy_Intersection(Set A, Set B, Universe U)
Function Fuzzy_Union(Set A, Set B, Universe U)

Main:
  Define Fuzzy Set A
  Define Fuzzy Set B

  // Determine the universe from keys of A and B
  U = keys(A) âˆª keys(B)

  // --- Calculate LHS: Â¬(A âˆª B) ---
  // 1. Calculate Union A âˆª B
  Union_AB = Fuzzy_Union(A, B, U)
  // 2. Calculate Complement of the Union
  LHS = Fuzzy_Complement(Union_AB, U)

  // --- Calculate RHS: Â¬A âˆ© Â¬B ---
  // 1. Calculate Complement Â¬A
  Complement_A = Fuzzy_Complement(A, U)
  // 2. Calculate Complement Â¬B
  Complement_B = Fuzzy_Complement(B, U)
  // 3. Calculate Intersection of Complements
  RHS = Fuzzy_Intersection(Complement_A, Complement_B, U)

  // --- Verification ---
  Print "Set A:", A
  Print "Set B:", B
  Print "Universe U:", U

  Print "\n--- De Morgan's Law: Â¬(A âˆª B) = Â¬A âˆ© Â¬B ---"
  Print "LHS: Â¬(A âˆª B)", LHS
  Print "RHS: Â¬A âˆ© Â¬B", RHS

  // Check if LHS and RHS are equal for all elements
  is_equal = True
  For each element x in U:
    If LHS[x] is not approximately equal to RHS[x]: // Use tolerance for float comparison
      is_equal = False
      Print "Mismatch found at element:", x, "LHS:", LHS[x], "RHS:", RHS[x]
      Break

  If is_equal:
    Print "\nVerification Successful: De Morgan's Law Â¬(A âˆª B) = Â¬A âˆ© Â¬B holds."
  Else:
    Print "\nVerification Failed: De Morgan's Law does not hold (check implementation or float precision)."
```

Python Code

```
import numpy as np # Using numpy for potential float precision handling

def get_membership(fuzzy_set, element):
    """Helper to get membership value, returns 0 if element not in set."""
    return fuzzy_set.get(element, 0.0)

def fuzzy_complement(fuzzy_set, universe):
    """Calculates the complement of a fuzzy set."""
    complement_set = {}
    for element in universe:
        complement_set[element] = 1.0 - get_membership(fuzzy_set, element)
    return complement_set

def fuzzy_intersection(set_a, set_b, universe):
    """Calculates the intersection of two fuzzy sets using min operator."""
    intersection_set = {}
    for element in universe:
        membership_a = get_membership(set_a, element)
        membership_b = get_membership(set_b, element)
        intersection_set[element] = min(membership_a, membership_b)
    return intersection_set

def fuzzy_union(set_a, set_b, universe):
    """Calculates the union of two fuzzy sets using max operator."""
    union_set = {}
    for element in universe:
        membership_a = get_membership(set_a, element)
        membership_b = get_membership(set_b, element)
        union_set[element] = max(membership_a, membership_b)
    return union_set

def print_fuzzy_set(name, fuzzy_set):
    """Prints a fuzzy set in a readable format."""
    print(f"{name}: {{", end='')
    # Sort items for consistent output
    items = [f"'{k}': {v:.3f}" for k, v in sorted(fuzzy_set.items())]
    print(', '.join(items), end='')
    print("}")

# --- Main Execution: Demonstrate De Morgan's Law Â¬(A âˆª B) = Â¬A âˆ© Â¬B ---
if __name__ == "__main__":
    # Define 2 fuzzy sets
    set_A = {'x1': 0.7, 'x2': 0.4, 'x3': 0.9, 'x4': 0.1}
    set_B = {'x1': 0.2, 'x2': 0.8, 'x3': 0.5, 'x5': 0.6}

    print("--- Original Fuzzy Sets ---")
    print_fuzzy_set("Set A", set_A)
    print_fuzzy_set("Set B", set_B)

    # Determine the universe of discourse
    universe = set(set_A.keys()) | set(set_B.keys())
    print(f"\nUniverse of Discourse (U): {sorted(list(universe))}")

    # --- Calculate LHS: Â¬(A âˆª B) ---
    print("\n--- Calculating LHS: Â¬(A âˆª B) ---")
    union_AB = fuzzy_union(set_A, set_B, universe)
    print_fuzzy_set("  1. Union (A âˆª B)", union_AB)
    lhs_result = fuzzy_complement(union_AB, universe)
    print_fuzzy_set("  2. LHS = Â¬(A âˆª B)", lhs_result)

    # --- Calculate RHS: Â¬A âˆ© Â¬B ---
    print("\n--- Calculating RHS: Â¬A âˆ© Â¬B ---")
    comp_A = fuzzy_complement(set_A, universe)
    print_fuzzy_set("  1. Complement Â¬A", comp_A)
    comp_B = fuzzy_complement(set_B, universe)
```

```
print_fuzzy_set("  2. Complement Â¬B", comp_B)
rhs_result = fuzzy_intersection(comp_A, comp_B, universe)
print_fuzzy_set("  3. RHS = Â¬A â^© Â¬B", rhs_result)

# --- Verification ---
print("\n--- Verification ---")
verification_passed = True
for element in universe:
    # Use numpy.isclose for robust floating-point comparison
    if not np.isclose(lhs_result.get(element, 0.0), rhs_result.get(element, 0.0)):
        verification_passed = False
        print(f"Mismatch found for element '{element}':")
        print(f"  LHS [Â¬(A â^ª B)]({element}) = {lhs_result.get(element, 0.0):.3f}")
        print(f"  RHS [Â¬A â^© Â¬B]({element}) = {rhs_result.get(element, 0.0):.3f}")
        break # Stop at first mismatch

if verification_passed:
    print("\nSUCCESS: De Morgan's Law Â¬(A â^ª B) = Â¬A â^© Â¬B holds for the given sets.")
else:
    print("\nFAILURE: De Morgan's Law Â¬(A â^ª B) = Â¬A â^© Â¬B does NOT hold (check implementation or precision).")
```

Explanation:

1. **Setup:** Defines two fuzzy sets, `set_A` and `set_B`, and determines their combined universe.
2. **LHS Calculation:**
   - Calculates `A â^ª B` using the `fuzzy_union` function (max operator).
   - Calculates the complement of the result using `fuzzy_complement` (1 - membership).
3. **RHS Calculation:**
   - Calculates `Â¬A` using `fuzzy_complement`.
   - Calculates `Â¬B` using `fuzzy_complement`.
   - Calculates the intersection of `Â¬A` and `Â¬B` using `fuzzy_intersection` (min operator).
4. **Verification:** Compares the membership values of the LHS result (`lhs_result`) and RHS result (`rhs_result`) for every element in the universe. It uses `numpy.isclose` to handle potential floating-point inaccuracies.
5. **Output:** Prints the original sets, the intermediate steps for both LHS and RHS calculations, and a final message indicating whether the verification was successful, confirming that De Morgan's law holds for standard fuzzy operations.

Experiment 14: Fuzzy Set Operations - De Morganâ€™s Law (Complement of Intersection)

Theory

Fuzzy set theory allows elements to have partial membership (between 0 and 1) in a set. Standard operations like complement, intersection, and union are defined based on these membership degrees.

Basic Operations (Standard Definitions):

- **Complement (Â¬A):** Î¼_Â¬A(x) = 1 - Î¼_A(x)
- **Intersection (A â^© B):** Î¼_(A â^© B)(x) = min(Î¼_A(x), Î¼_B(x))
- **Union (A â^ª B):** Î¼_(A â^ª B)(x) = max(Î¼_A(x), Î¼_B(x))

De Morgan's Laws in Fuzzy Logic:

Similar to classical set theory, De Morgan's laws apply to standard fuzzy set operations.

This experiment focuses on the second De Morgan's law for fuzzy sets:

Â¬(A â^© B) = Â¬A â^ª Â¬B

This law states that the complement of the intersection of two fuzzy sets A and B is equal to the union of their complements.

- **Left-Hand Side (LHS): Â¬(A â© B)**
    1. First, find the intersection of A and B: Î¼_(A â© B)(x) = min(Î¼_A(x), Î¼_B(x)).
    2. Then, find the complement of the result: Î¼_Â¬(A â© B)(x) = 1 - Î¼_(A â© B)(x) = 1 - min(Î¼_A(x), Î¼_B(x)).
- **Right-Hand Side (RHS): Â¬A âª Â¬B**
    1. First, find the complement of A: Î¼_Â¬A(x) = 1 - Î¼_A(x).
    2. First, find the complement of B: Î¼_Â¬B(x) = 1 - Î¼_B(x).
    3. Then, find the union of the complements: Î¼_(Â¬A âª Â¬B)(x) = max(Î¼_Â¬A(x), Î¼_Â¬B(x)) = max(1 - Î¼_A(x), 1 - Î¼_B(x)).

De Morgan's law confirms that `1 - min(a, b)` is equivalent to `max(1 - a, 1 - b)` for membership values `a` and `b`.

Pseudocode/Algorithm

Let A, B be fuzzy sets defined over a universe U. Assume A, B are represented as dictionaries mapping elements x â^ U to membership values Î¼(x).

```
// Helper functions (from Experiment 12)
Function Get_Membership(Set S, element x)
Function Fuzzy_Complement(Set S, Universe U)
Function Fuzzy_Intersection(Set A, Set B, Universe U)
Function Fuzzy_Union(Set A, Set B, Universe U)

Main:
  Define Fuzzy Set A
  Define Fuzzy Set B

  // Determine the universe from keys of A and B
  U = keys(A) â^ª keys(B)

  // --- Calculate LHS: Â¬(A â© B) ---
  // 1. Calculate Intersection A â© B
  Intersection_AB = Fuzzy_Intersection(A, B, U)
  // 2. Calculate Complement of the Intersection
  LHS = Fuzzy_Complement(Intersection_AB, U)

  // --- Calculate RHS: Â¬A âª Â¬B ---
  // 1. Calculate Complement Â¬A
  Complement_A = Fuzzy_Complement(A, U)
  // 2. Calculate Complement Â¬B
  Complement_B = Fuzzy_Complement(B, U)
  // 3. Calculate Union of Complements
  RHS = Fuzzy_Union(Complement_A, Complement_B, U)

  // --- Verification ---
  Print "Set A:", A
  Print "Set B:", B
  Print "Universe U:", U

  Print "\n--- De Morgan's Law: Â¬(A â© B) = Â¬A âª Â¬B ---"
  Print "LHS: Â¬(A â© B)", LHS
  Print "RHS: Â¬A âª Â¬B", RHS

  // Check if LHS and RHS are equal for all elements
  is_equal = True
  For each element x in U:
    If LHS[x] is not approximately equal to RHS[x]: // Use tolerance for float comparison
      is_equal = False
      Print "Mismatch found at element:", x, "LHS:", LHS[x], "RHS:", RHS[x]
      Break

  If is_equal:
    Print "\nVerification Successful: De Morgan's Law Â¬(A â© B) = Â¬A âª Â¬B holds."
  Else:
    Print "\nVerification Failed: De Morgan's Law does not hold (check implementation or float precision)."
```

Python Code

```
import numpy as np # Using numpy for potential float precision handling

def get_membership(fuzzy_set, element):
    """Helper to get membership value, returns 0 if element not in set."""
    return fuzzy_set.get(element, 0.0)

def fuzzy_complement(fuzzy_set, universe):
    """Calculates the complement of a fuzzy set."""
    complement_set = {}
    for element in universe:
        complement_set[element] = 1.0 - get_membership(fuzzy_set, element)
    return complement_set

def fuzzy_intersection(set_a, set_b, universe):
    """Calculates the intersection of two fuzzy sets using min operator."""
    intersection_set = {}
    for element in universe:
        membership_a = get_membership(set_a, element)
        membership_b = get_membership(set_b, element)
        intersection_set[element] = min(membership_a, membership_b)
    return intersection_set

def fuzzy_union(set_a, set_b, universe):
    """Calculates the union of two fuzzy sets using max operator."""
    union_set = {}
    for element in universe:
        membership_a = get_membership(set_a, element)
        membership_b = get_membership(set_b, element)
        union_set[element] = max(membership_a, membership_b)
    return union_set

def print_fuzzy_set(name, fuzzy_set):
    """Prints a fuzzy set in a readable format."""
    print(f"{name}: {{", end='')
    # Sort items for consistent output
    items = [f"'{k}': {v:.3f}" for k, v in sorted(fuzzy_set.items())]
    print(', '.join(items), end='')
    print("}")

# --- Main Execution: Demonstrate De Morgan's Law Â¬(A â©Â B) = Â¬A âªÂ Â¬B ---
if __name__ == "__main__":
    # Define 2 fuzzy sets
    set_A = {'apple': 0.8, 'banana': 0.3, 'cherry': 0.6, 'date': 0.1}
    set_B = {'apple': 0.4, 'banana': 0.9, 'date': 0.5, 'fig': 0.7}

    print("--- Original Fuzzy Sets ---")
    print_fuzzy_set("Set A", set_A)
    print_fuzzy_set("Set B", set_B)

    # Determine the universe of discourse
    universe = set(set_A.keys()) | set(set_B.keys())
    print(f"\nUniverse of Discourse (U): {sorted(list(universe))}")

    # --- Calculate LHS: Â¬(A â©Â B) ---
    print("\n--- Calculating LHS: Â¬(A â©Â B) ---")
    intersection_AB = fuzzy_intersection(set_A, set_B, universe)
    print_fuzzy_set("  1. Intersection (A â©Â B)", intersection_AB)
    lhs_result = fuzzy_complement(intersection_AB, universe)
    print_fuzzy_set("  2. LHS = Â¬(A â©Â B)", lhs_result)

    # --- Calculate RHS: Â¬A âªÂ Â¬B ---
    print("\n--- Calculating RHS: Â¬A âªÂ Â¬B ---")
    comp_A = fuzzy_complement(set_A, universe)
    print_fuzzy_set("  1. Complement Â¬A", comp_A)
    comp_B = fuzzy_complement(set_B, universe)
```

```
print_fuzzy_set("  2. Complement Â¬B", comp_B)
rhs_result = fuzzy_union(comp_A, comp_B, universe)
print_fuzzy_set("  3. RHS = Â¬A âˆª Â¬B", rhs_result)

# --- Verification ---
print("\n--- Verification ---")
verification_passed = True
for element in universe:
    # Use numpy.isclose for robust floating-point comparison
    if not np.isclose(lhs_result.get(element, 0.0), rhs_result.get(element, 0.0)):
        verification_passed = False
        print(f"Mismatch found for element '{element}':")
        print(f"  LHS [Â¬(A âˆ© B)]({element}) = {lhs_result.get(element, 0.0):.3f}")
        print(f"  RHS [Â¬A âˆª Â¬B]({element}) = {rhs_result.get(element, 0.0):.3f}")
        break # Stop at first mismatch

if verification_passed:
    print("\nSUCCESS: De Morgan's Law Â¬(A âˆ© B) = Â¬A âˆª Â¬B holds for the given sets.")
else:
    print("\nFAILURE: De Morgan's Law Â¬(A âˆ© B) = Â¬A âˆª Â¬B does NOT hold (check implementation or precision).")
```

Explanation:

1. **Setup:** Defines two fuzzy sets, `set_A` and `set_B`, and determines their combined universe.
2. **LHS Calculation:**
   - Calculates `A âˆ© B` using the `fuzzy_intersection` function (min operator).
   - Calculates the complement of the result using `fuzzy_complement` (1 - membership).
3. **RHS Calculation:**
   - Calculates `Â¬A` using `fuzzy_complement`.
   - Calculates `Â¬B` using `fuzzy_complement`.
   - Calculates the union of `Â¬A` and `Â¬B` using `fuzzy_union` (max operator).
4. **Verification:** Compares the membership values of the LHS result (`lhs_result`) and RHS result (`rhs_result`) for every element in the universe using `numpy.isclose` for floating-point accuracy.
5. **Output:** Prints the original sets, the intermediate steps for both LHS and RHS calculations, and a final message confirming whether the second De Morgan's law holds for the standard fuzzy operations.

Experiment 15: Min-Max Algorithm (Computer Wins or Draws)

Theory

Min-Max Algorithm:

Min-Max is a decision-making algorithm used in two-player, zero-sum games (like Tic-Tac-Toe, Chess, Nim) where players have perfect information. The goal is to find the optimal move for a player, assuming the opponent also plays optimally.

- **Zero-Sum Game:** One player's gain is the other player's loss.
- **Players:** Typically referred to as MAX (the player trying to maximize the score, often the AI) and MIN (the player trying to minimize the score, often the human opponent).
- **Game Tree:** The game states are represented as nodes in a tree. The root is the current state, and children represent states reachable after one move.
- **Utility Function:** Assigns a numerical score to terminal states (end-game states). For Tic-Tac-Toe:
  - +1 if MAX (Computer) wins.
  - -1 if MIN (Human) wins.
  - 0 for a draw.
- **Recursive Exploration:** The algorithm explores the game tree recursively.
  - **MAX Player's Turn:** Chooses the move that leads to the state with the *maximum* utility value (obtained from the MIN player's subsequent moves).
  - **MIN Player's Turn:** Chooses the move that leads to the state with the *minimum* utility value (obtained from the MAX player's subsequent moves).

Goal: Computer Wins or Draws

To ensure the computer (MAX) always wins or draws, the Min-Max algorithm explores all possible game outcomes from the current state. By always choosing the move that maximizes its potential score (assuming the opponent minimizes it), the computer guarantees the best possible outcome for itself, which, in a perfectly played game like Tic-Tac-Toe from the start, is at least a draw.

Tic-Tac-Toe Specifics:

- **State:** The 3x3 board configuration.
- **Moves:** Placing the player's mark (X or O) in an empty cell.
- **Terminal States:** Board is full (draw), or one player has three marks in a row/column/diagonal (win/loss).

Pseudocode/Algorithm

```
// Represents the board (e.g., 3x3 list of lists)
// Player symbols (e.g., 'X' for computer/MAX, 'O' for human/MIN)

Function evaluate(board):
  // Check rows, columns, diagonals for a win
  If Computer ('X') has won:
    Return +1
  Else if Human ('O') has won:
    Return -1
  Else if Board is full (no empty cells):
    Return 0 // Draw
  Else:
    Return null // Game not over

Function minimax(board, depth, is_maximizing_player):
  score = evaluate(board)

  // Terminal state reached
  If score is not null:
    Return score

  If is_maximizing_player: // Computer's turn (MAX)
    best_score = -infinity
    For each possible move on the board:
      Make the move for Computer ('X')
      // Recursively call minimax for the opponent's turn
      current_score = minimax(board, depth + 1, False)
      Undo the move
      best_score = max(best_score, current_score)
    Return best_score

  Else: // Human's turn (MIN)
    best_score = +infinity
    For each possible move on the board:
      Make the move for Human ('O')
      // Recursively call minimax for the computer's turn
      current_score = minimax(board, depth + 1, True)
      Undo the move
      best_score = min(best_score, current_score)
    Return best_score

Function find_best_move(board):
  best_score = -infinity
  best_move = null // (row, col)

  For each possible move (row, col) on the board:
    If cell (row, col) is empty:
      Make the move for Computer ('X')
      move_score = minimax(board, 0, False) // Start recursion for MIN player
      Undo the move

      If move_score > best_score:
        best_score = move_score
        best_move = (row, col)

  Return best_move

Main Game Loop:
  Initialize empty board
  Decide who goes first (e.g., Computer)

  While game is not over:
    If it's Computer's turn:
      (row, col) = find_best_move(board)
      Make move at (row, col) for Computer ('X')
```

```
  Else (Human's turn):
    Get valid move (row, col) from Human input
    Make move at (row, col) for Human ('O')

  Print the board
  Check if game has ended (evaluate(board) is not null)

Declare winner or draw based on evaluate(board)
```

Python Code

```
import math
import random

# Player symbols
COMPUTER = 'X'
HUMAN = 'O'
EMPTY = '-'

def print_board(board):
    """Prints the Tic-Tac-Toe board."""
    print("\nBoard:")
    for row in board:
        print(" | ".join(row))
        print("---------")
    print()

def evaluate(board):
    """Checks for a win, loss, or draw.
    Returns +1 for Computer win, -1 for Human win, 0 for Draw, None otherwise.
    """
    # Check rows, columns, and diagonals
    lines = []
    lines.extend(board) # Rows
    lines.extend([list(col) for col in zip(*board)]) # Columns
    lines.append([board[i][i] for i in range(3)]) # Main diagonal
    lines.append([board[i][2 - i] for i in range(3)]) # Anti-diagonal

    for line in lines:
        if all(cell == COMPUTER for cell in line):
            return 1
        if all(cell == HUMAN for cell in line):
            return -1

    # Check for draw (no empty cells left)
    if all(cell != EMPTY for row in board for cell in row):
        return 0

    # Game is not over yet
    return None

def get_empty_cells(board):
    """Returns a list of (row, col) tuples for empty cells."""
    cells = []
    for r in range(3):
        for c in range(3):
            if board[r][c] == EMPTY:
                cells.append((r, c))
    return cells

def minimax(board, depth, is_maximizing):
    """Minimax algorithm implementation."""
    score = evaluate(board)

    # Terminal state
    if score is not None:
        return score

    empty_cells = get_empty_cells(board)

    if is_maximizing: # Computer's turn (Maximize)
        best_val = -math.inf
        for r, c in empty_cells:
            board[r][c] = COMPUTER
            value = minimax(board, depth + 1, False)
            board[r][c] = EMPTY # Undo move
```

```
            best_val = max(best_val, value)
        return best_val
    else: # Human's turn (Minimize)
        best_val = math.inf
        for r, c in empty_cells:
            board[r][c] = HUMAN
            value = minimax(board, depth + 1, True)
            board[r][c] = EMPTY # Undo move
            best_val = min(best_val, value)
        return best_val

def find_best_move(board):
    """Finds the best move for the Computer using Minimax."""
    best_val = -math.inf
    best_move = (-1, -1)
    empty_cells = get_empty_cells(board)

    # If it's the first move, choose randomly for variety (optional)
    # if len(empty_cells) == 9:
    #     return random.choice(empty_cells)

    for r, c in empty_cells:
        board[r][c] = COMPUTER
        move_val = minimax(board, 0, False) # Evaluate opponent's potential moves
        board[r][c] = EMPTY # Undo move

        # print(f"Move ({r},{c}) evaluated score: {move_val}") # Debugging

        if move_val > best_val:
            best_move = (r, c)
            best_val = move_val

    # print(f"Chosen best move: {best_move} with score {best_val}") # Debugging
    return best_move

# --- Main Game Loop ---
if __name__ == "__main__":
    board = [[EMPTY, EMPTY, EMPTY],
             [EMPTY, EMPTY, EMPTY],
             [EMPTY, EMPTY, EMPTY]]

    current_player = COMPUTER # Or HUMAN if human starts
    print("Tic-Tac-Toe: Computer (X) vs Human (O)")
    print(f"{current_player} goes first.")

    while True:
        print_board(board)
        game_over_score = evaluate(board)

        if game_over_score is not None:
            if game_over_score == 1:
                print("Computer (X) wins!")
            elif game_over_score == -1:
                print("Human (O) wins!")
            else:
                print("It's a Draw!")
            break

        if current_player == COMPUTER:
            print("Computer's turn...")
            row, col = find_best_move(board)
            if board[row][col] == EMPTY:
                board[row][col] = COMPUTER
                print(f"Computer places X at ({row}, {col})")
                current_player = HUMAN
            else:
```

```
            print("Error: Computer tried invalid move?") # Should not happen
            break
    else: # Human's turn
        print("Human's turn (O).")
        while True:
            try:
                move = input("Enter your move (row col, e.g., 0 1): ").split()
                r, c = int(move[0]), int(move[1])
                if 0 <= r <= 2 and 0 <= c <= 2 and board[r][c] == EMPTY:
                    board[r][c] = HUMAN
                    current_player = COMPUTER
                    break
                else:
                    print("Invalid move. Cell occupied or out of bounds. Try again.")
            except (ValueError, IndexError):
                print("Invalid input format. Enter row and column numbers (0-2) separated by space.")
```

Explanation:

1. **Board Representation:** A 3x3 list of lists stores the game state (`EMPTY`, `COMPUTER`, `HUMAN`).
2. **`evaluate(board)`:** Checks if the current board state is a terminal state (win, loss, draw) and returns the corresponding score (+1, -1, 0) or `None` if the game continues.
3. **`minimax(board, depth, is_maximizing)`:** The core recursive function. It explores possible moves:
    - If `is_maximizing` (Computer's turn), it tries all empty cells, makes the move, recursively calls `minimax` for the minimizing player, undoes the move, and returns the maximum score found.
    - If not `is_maximizing` (Human's turn), it does the same but returns the minimum score found.
4. **`find_best_move(board)`:** Iterates through all empty cells, simulates placing the computer's mark, calls `minimax` to evaluate the outcome assuming the opponent plays optimally (minimizing), undoes the move, and chooses the move that resulted in the highest evaluated score.
5. **Game Loop:** Alternates turns between the computer and the human. The computer uses `find_best_move` to select its move. The human provides input. The loop continues until `evaluate` returns a non-`None` score.

   This implementation ensures the computer plays optimally according to the Min-Max algorithm, guaranteeing it will either win or draw if it plays perfectly from any state.

Experiment 16: Min-Max Algorithm (Computer Loses or Draws)

Theory

Min-Max Algorithm:

Min-Max is a recursive algorithm used in two-player, zero-sum games with perfect information to determine the optimal move. It explores the game tree, assuming both players play optimally.

- **Players:** MAX (tries to maximize score) and MIN (tries to minimize score).
- **Utility Function:** Assigns scores to terminal states. For Tic-Tac-Toe:
    - +1 if the standard MAX player wins.
    - -1 if the standard MIN player wins.
    - 0 for a draw.
- **Goal: Computer Loses or Draws:** In this variation, the computer is programmed to play sub-optimally, specifically to *lose* or *draw*, but never win. This means the computer (let's still call it 'X' for consistency, but it's playing to *minimize* its own score according to the standard evaluation) should aim for the worst possible outcome for itself.

  To achieve this, we can flip the logic: The computer, when it's its turn, will choose the move that leads to the *minimum* possible score (according to the standard +1 for X win, -1 for O win evaluation), assuming the opponent (Human, 'O') plays to *maximize* that score.

  Essentially, the computer plays the role of the MIN player from the standard Min-Max perspective, while the human plays the role of the MAX player.

Modified Logic:

- **Computer's Turn ('X'):** Choose the move that leads to the state with the *minimum* utility value (obtained from the Human's subsequent maximizing moves).
- **Human's Turn ('O'):** Assume the human plays optimally to *maximize* their score (minimize the computer's score). Choose the move that leads to the state with the *maximum* utility value (obtained from the Computer's subsequent minimizing moves).

Pseudocode/Algorithm

```
// Represents the board (e.g., 3x3 list of lists)
// Player symbols (e.g., 'X' for computer/MIN-acting, 'O' for human/MAX-acting)

Function evaluate(board):
  // Standard evaluation: +1 for 'X' win, -1 for 'O' win, 0 for draw
  If Computer ('X') has won:
    Return +1
  Else if Human ('O') has won:
    Return -1
  Else if Board is full:
    Return 0 // Draw
  Else:
    Return null // Game not over

// Note: is_computers_turn determines if we minimize or maximize
Function minimax_lose_or_draw(board, depth, is_computers_turn):
  score = evaluate(board)

  // Terminal state reached
  If score is not null:
    Return score

  If is_computers_turn: // Computer's turn ('X') - Aims to MINIMIZE the score
    worst_score = +infinity
    For each possible move on the board:
      Make the move for Computer ('X')
      // Recursively call for the opponent's turn (opponent maximizes)
      current_score = minimax_lose_or_draw(board, depth + 1, False)
      Undo the move
      worst_score = min(worst_score, current_score)
    Return worst_score

  Else: // Human's turn ('O') - Assumed to MAXIMIZE the score
    best_score = -infinity
    For each possible move on the board:
      Make the move for Human ('O')
      // Recursively call for the computer's turn (computer minimizes)
      current_score = minimax_lose_or_draw(board, depth + 1, True)
      Undo the move
      best_score = max(best_score, current_score)
    Return best_score

Function find_worst_move_for_computer(board):
  worst_score = +infinity
  worst_move = null // (row, col)

  For each possible move (row, col) on the board:
    If cell (row, col) is empty:
      Make the move for Computer ('X')
      // Evaluate based on the opponent (Human) trying to maximize
      move_score = minimax_lose_or_draw(board, 0, False)
      Undo the move

      // Computer chooses the move leading to the minimum score
      If move_score < worst_score:
        worst_score = move_score
        worst_move = (row, col)
      // Optional: If scores are equal, maybe add randomness or pick first found

  // Handle case where no move found (shouldn't happen if board not full)
  If worst_move is null and board is not full:
      Find any valid empty cell as fallback

  Return worst_move
```

```
Main Game Loop:
  Initialize empty board
  Decide who goes first (e.g., Human)

  While game is not over:
    If it's Computer's turn ('X'):
      (row, col) = find_worst_move_for_computer(board)
      Make move at (row, col) for Computer ('X')
    Else (Human's turn 'O'):
      Get valid move (row, col) from Human input
      Make move at (row, col) for Human ('O')

    Print the board
    Check if game has ended (evaluate(board) is not null)

  Declare winner or draw based on evaluate(board)
```

Python Code

```
import math
import random

# Player symbols
COMPUTER = 'X' # Plays to lose or draw (acts as MIN)
HUMAN = 'O'    # Plays to win (acts as MAX)
EMPTY = '-'

def print_board(board):
    """Prints the Tic-Tac-Toe board."""
    print("\nBoard:")
    for row in board:
        print(" | ".join(row))
        print("---------")
    print()

def evaluate(board):
    """Checks for a win, loss, or draw (Standard Evaluation).
    Returns +1 for X win, -1 for O win, 0 for Draw, None otherwise.
    """
    lines = []
    lines.extend(board)
    lines.extend([list(col) for col in zip(*board)])
    lines.append([board[i][i] for i in range(3)])
    lines.append([board[i][2 - i] for i in range(3)])

    for line in lines:
        if all(cell == COMPUTER for cell in line): return 1 # X wins
        if all(cell == HUMAN for cell in line): return -1 # O wins

    if all(cell != EMPTY for row in board for cell in row):
        return 0 # Draw

    return None # Game not over

def get_empty_cells(board):
    """Returns a list of (row, col) tuples for empty cells."""
    cells = []
    for r in range(3):
        for c in range(3):
            if board[r][c] == EMPTY:
                cells.append((r, c))
    return cells

def minimax_lose_or_draw(board, depth, is_computers_turn):
    """Minimax where Computer ('X') minimizes, Human ('O') maximizes."""
    score = evaluate(board)
    if score is not None: return score

    empty_cells = get_empty_cells(board)

    if is_computers_turn: # Computer's turn ('X') -> Minimize score
        worst_val = math.inf
        for r, c in empty_cells:
            board[r][c] = COMPUTER
            value = minimax_lose_or_draw(board, depth + 1, False) # Opponent's turn (maximizes)
            board[r][c] = EMPTY
            worst_val = min(worst_val, value)
        return worst_val
    else: # Human's turn ('O') -> Maximize score
        best_val = -math.inf
        for r, c in empty_cells:
            board[r][c] = HUMAN
            value = minimax_lose_or_draw(board, depth + 1, True) # Computer's turn (minimizes)
            board[r][c] = EMPTY
```

```
            best_val = max(best_val, value)
        return best_val

def find_worst_move_for_computer(board):
    """Finds the move for the Computer that leads to the minimum score."""
    worst_val = math.inf
    worst_move = (-1, -1)
    empty_cells = get_empty_cells(board)

    # Shuffle empty cells to add randomness among equally bad moves (optional)
    random.shuffle(empty_cells)

    for r, c in empty_cells:
        board[r][c] = COMPUTER
        # Evaluate based on Human (opponent) trying to maximize
        move_val = minimax_lose_or_draw(board, 0, False)
        board[r][c] = EMPTY

        # print(f"Move ({r},{c}) evaluated score: {move_val}") # Debugging

        if move_val < worst_val:
            worst_move = (r, c)
            worst_val = move_val
        # If worst_move is still not set (e.g., first iteration), set it
        elif worst_move == (-1, -1):
             worst_move = (r, c)

    # print(f"Chosen worst move: {worst_move} with score {worst_val}") # Debugging
    # Fallback if something went wrong (shouldn't be needed in TicTacToe)
    if worst_move == (-1, -1) and empty_cells:
        print("Warning: Fallback move selection!")
        worst_move = empty_cells[0]

    return worst_move

# --- Main Game Loop ---
if __name__ == "__main__":
    board = [[EMPTY, EMPTY, EMPTY],
             [EMPTY, EMPTY, EMPTY],
             [EMPTY, EMPTY, EMPTY]]

    # Let Human start to give Computer a chance to lose
    current_player = HUMAN
    print("Tic-Tac-Toe: Computer (X) vs Human (O)")
    print("Goal: Computer tries to lose or draw.")
    print(f"{current_player} goes first.")

    while True:
        print_board(board)
        game_over_score = evaluate(board)

        if game_over_score is not None:
            if game_over_score == 1: print("Computer (X) wins! (Unexpectedly?)")
            elif game_over_score == -1: print("Human (O) wins!")
            else: print("It's a Draw!")
            break

        if current_player == COMPUTER:
            print("Computer's turn (trying to lose/draw)...")
            row, col = find_worst_move_for_computer(board)
            if board[row][col] == EMPTY:
                board[row][col] = COMPUTER
                print(f"Computer places X at ({row}, {col})")
                current_player = HUMAN
            else:
                print("Error: Computer chose invalid move?")
```

```
            break # Should not happen
    else: # Human's turn
        print("Human's turn (O).")
        while True:
            try:
                move = input("Enter your move (row col, e.g., 0 1): ").split()
                r, c = int(move[0]), int(move[1])
                if 0 <= r <= 2 and 0 <= c <= 2 and board[r][c] == EMPTY:
                    board[r][c] = HUMAN
                    current_player = COMPUTER
                    break
                else:
                    print("Invalid move. Cell occupied or out of bounds. Try again.")
            except (ValueError, IndexError):
                print("Invalid input format. Enter row and column numbers (0-2) separated by space.")
```

Explanation:

1. **Evaluation:** The `evaluate` function remains the same, providing a standard score (+1 for X win, -1 for O win, 0 for draw).
2. **`minimax_lose_or_draw`:** This is the core change. When `is_computers_turn` is `True`, the function seeks the *minimum* score (`worst_val = min(...)`). When it's the human's turn (`False`), it assumes the human plays optimally to *maximize* the score (`best_val = max(...)`).
3. **`find_worst_move_for_computer`:** This function iterates through possible moves for the computer ('X'). For each move, it calls `minimax_lose_or_draw` starting from the opponent's perspective (`is_computers_turn=False`) to see the outcome if the opponent plays to maximize. The computer then chooses the move (`worst_move`) that resulted in the *minimum* evaluated score (`worst_val`).
4. **Game Loop:** The structure is similar to Experiment 15, but the computer now calls `find_worst_move_for_computer` to make its decision. The human is assumed to play normally (trying to win).

This setup forces the computer to select moves that, assuming the human plays optimally to win, will lead to the worst possible outcome for the computer (a loss or a draw).

Experiment 17: Simple Multi-Layer Perceptron (N Inputs, 2 Hidden, 1 Output)

Theory

Multi-Layer Perceptron (MLP):

An MLP is a type of feedforward artificial neural network (ANN). It consists of at least three layers of nodes: an input layer, one or more hidden layers, and an output layer. Except for the input nodes, each node is a neuron that uses a non-linear activation function.

- **Input Layer:** Receives the initial data (features). In this case, N binary inputs (0 or 1).
- **Hidden Layers:** Intermediate layers between the input and output layers. They allow the network to learn complex patterns by transforming the input data. This MLP has two hidden layers.
- **Output Layer:** Produces the final result. Here, it has one neuron producing a single binary output (typically interpreted as 0 or 1, often via a threshold on an activation function like Sigmoid).
- **Neurons (Nodes):** Each neuron (except input) receives weighted inputs from the previous layer, adds a bias, applies an activation function, and passes the result to the next layer.
- **Weights:** Parameters associated with the connections between neurons. They determine the strength of the connection. Initialized randomly.
- **Biases:** Parameters associated with each neuron (except input). They allow shifting the activation function. Initialized randomly.
- **Activation Function:** Introduces non-linearity, allowing the network to learn complex relationships. Common choices include Sigmoid, ReLU, Tanh. For a binary output, Sigmoid is often used in the output layer.
- **Feedforward:** Information flows in one direction, from the input layer, through the hidden layers, to the output layer, without cycles.

Task Specifics:

This experiment requires creating the structure of an MLP with:

- `N` binary inputs.
- A first hidden layer (size can be chosen, e.g., `h1_size`).
- A second hidden layer (size can be chosen, e.g., `h2_size`).
- An output layer with 1 neuron for binary output.

The task focuses on initializing the weights and biases randomly and displaying them. It does *not* involve training (like backpropagation) or making predictions, only setting up the initial random state. The "number of steps" mentioned is ambiguous in this context without training; it might refer to the number of layers (input + hidden + output = 4 steps/layers) or simply be a placeholder. We will display the structure and initial values.

Weight Matrix Dimensions:

- Input to Hidden Layer 1 (W1): `(N, h1_size)`
- Bias for Hidden Layer 1 (b1): `(1, h1_size)`
- Hidden Layer 1 to Hidden Layer 2 (W2): `(h1_size, h2_size)`
- Bias for Hidden Layer 2 (b2): `(1, h2_size)`
- Hidden Layer 2 to Output Layer (W3): `(h2_size, 1)`
- Bias for Output Layer (b3): `(1, 1)`

Pseudocode/Algorithm

```
// Define Network Structure
Input: N (number of binary inputs)
Define h1_size (number of neurons in hidden layer 1)
Define h2_size (number of neurons in hidden layer 2)
Output_size = 1

// Initialize Weights and Biases Randomly
Function initialize_mlp(N, h1_size, h2_size, output_size):
  // Weights
  W1 = Generate random matrix of size (N, h1_size)
  W2 = Generate random matrix of size (h1_size, h2_size)
  W3 = Generate random matrix of size (h2_size, output_size)

  // Biases
  b1 = Generate random vector of size (1, h1_size)
  b2 = Generate random vector of size (1, h2_size)
  b3 = Generate random vector of size (1, output_size)

  Return W1, b1, W2, b2, W3, b3

// Display Function
Function display_parameters(W1, b1, W2, b2, W3, b3):
  Print "--- MLP Structure ---"
  Print "Input Layer Size:", N
  Print "Hidden Layer 1 Size:", h1_size
  Print "Hidden Layer 2 Size:", h2_size
  Print "Output Layer Size:", output_size

  Print "\n--- Initial Random Weights ---"
  Print "W1 (Input -> Hidden1) Matrix Shape:", shape(W1)
  Print W1
  Print "W2 (Hidden1 -> Hidden2) Matrix Shape:", shape(W2)
  Print W2
  Print "W3 (Hidden2 -> Output) Matrix Shape:", shape(W3)
  Print W3

  Print "\n--- Initial Random Biases ---"
  Print "b1 (Hidden1) Vector Shape:", shape(b1)
  Print b1
  Print "b2 (Hidden2) Vector Shape:", shape(b2)
  Print b2
  Print "b3 (Output) Vector Shape:", shape(b3)
  Print b3

  // Interpret "number of steps" as number of layers
  Print "\nNumber of Layers (Steps):", 4 // Input, Hidden1, Hidden2, Output

Main:
  Ask user for N (number of inputs)
  Set h1_size (e.g., 5)
  Set h2_size (e.g., 3)
  output_size = 1

  (W1, b1, W2, b2, W3, b3) = initialize_mlp(N, h1_size, h2_size, output_size)

  display_parameters(W1, b1, W2, b2, W3, b3)
```

Python Code

```
import numpy as np

def initialize_mlp(n_inputs, n_hidden1, n_hidden2, n_outputs):
    """Initializes weights and biases for a 2-hidden layer MLP randomly.

    Args:
        n_inputs (int): Number of input features (N).
        n_hidden1 (int): Number of neurons in the first hidden layer.
        n_hidden2 (int): Number of neurons in the second hidden layer.
        n_outputs (int): Number of output neurons (1 for this case).

    Returns:
        tuple: Contains weight matrices (W1, W2, W3) and bias vectors (b1, b2, b3).
    """
    # Initialize weights with small random values (e.g., from standard normal distribution)
    # Using random.randn which gives values from a standard normal distribution.
    # You could also use random.rand for uniform [0, 1) or scale them.
    W1 = np.random.randn(n_inputs, n_hidden1)
    W2 = np.random.randn(n_hidden1, n_hidden2)
    W3 = np.random.randn(n_hidden2, n_outputs)

    # Initialize biases, often starting with zeros or small random values
    b1 = np.random.randn(1, n_hidden1)
    b2 = np.random.randn(1, n_hidden2)
    b3 = np.random.randn(1, n_outputs)

    return W1, b1, W2, b2, W3, b3

def display_parameters(N, h1_size, h2_size, output_size, W1, b1, W2, b2, W3, b3):
    """Displays the network structure and initial parameters."""
    print("--- MLP Structure ---")
    print(f"Input Layer Size (N): {N}")
    print(f"Hidden Layer 1 Size: {h1_size}")
    print(f"Hidden Layer 2 Size: {h2_size}")
    print(f"Output Layer Size: {output_size}")

    print("\n--- Initial Random Weights ---")
    print(f"W1 (Input -> Hidden1) Matrix Shape: {W1.shape}")
    print(W1)
    print(f"\nW2 (Hidden1 -> Hidden2) Matrix Shape: {W2.shape}")
    print(W2)
    print(f"\nW3 (Hidden2 -> Output) Matrix Shape: {W3.shape}")
    print(W3)

    print("\n--- Initial Random Biases ---")
    print(f"b1 (Hidden1) Vector Shape: {b1.shape}")
    print(b1)
    print(f"\nb2 (Hidden2) Vector Shape: {b2.shape}")
    print(b2)
    print(f"\nb3 (Output) Vector Shape: {b3.shape}")
    print(b3)

    # Interpret "number of steps" as the number of layers involved in computation
    # Input layer doesn't compute, so 3 computational layers (2 hidden + 1 output)
    # Or simply the total number of layers = 4
    print(f"\nNumber of Layers (including input): 4")
    print(f"Number of Computational Layers (Steps): 3")

# --- Main Execution ---
if __name__ == "__main__":
    while True:
        try:
            n_inputs = int(input("Enter the number of binary inputs (N): "))
            if n_inputs > 0:
                break
```

```
            else:
                print("Number of inputs must be positive.")
        except ValueError:
            print("Invalid input. Please enter an integer.")

    # Define the size of the hidden layers (can be adjusted)
    n_hidden1 = 5
    n_hidden2 = 3
    n_outputs = 1 # Single binary output

    print(f"\nInitializing MLP with N={n_inputs}, Hidden1={n_hidden1}, Hidden2={n_hidden2}, Output={n_outputs}")

    # Initialize the MLP parameters
    W1, b1, W2, b2, W3, b3 = initialize_mlp(n_inputs, n_hidden1, n_hidden2, n_outputs)

    # Display the structure and initial parameters
    display_parameters(n_inputs, n_hidden1, n_hidden2, n_outputs, W1, b1, W2, b2, W3, b3)
```

Explanation:

1. **`initialize_mlp` Function:** Takes the number of neurons in each layer as input. It uses `numpy.random.randn` to create weight matrices and bias vectors with the correct dimensions, filled with random values drawn from a standard normal distribution.
2. **`display_parameters` Function:** Takes the layer sizes and the initialized weights/biases. It prints the shape and values of each weight matrix (W1, W2, W3) and bias vector (b1, b2, b3). It also clarifies the interpretation of "number of steps" as the number of layers.
3. **Main Block:**
    - Prompts the user to enter the number of inputs `N`.
    - Sets predefined sizes for the two hidden layers (`n_hidden1`, `n_hidden2`). These could also be taken as user input if desired.
    - Calls `initialize_mlp` to get the random weights and biases.
    - Calls `display_parameters` to show the results as required by the experiment.

This code fulfills the requirement of setting up the MLP structure with random weights/biases and displaying them, without performing any training or prediction.

Experiment 18: Simple Multi-Layer Perceptron (4 Inputs, 1 Hidden, 2 Outputs)

Theory

Multi-Layer Perceptron (MLP):

An MLP is a fundamental type of feedforward artificial neural network (ANN). It consists of an input layer, one or more hidden layers, and an output layer. Neurons in the hidden and output layers typically use non-linear activation functions to learn complex data patterns.

- **Input Layer:** Receives the raw input features. In this case, 4 binary inputs (0 or 1).
- **Hidden Layer:** An intermediate layer that transforms the input data. This MLP has one hidden layer.
- **Output Layer:** Produces the final prediction or classification. Here, it has two neurons, each producing a binary output (or a value interpretable as binary, like probabilities from Sigmoid).
- **Neurons:** Process weighted inputs plus a bias, apply an activation function, and pass the output forward.
- **Weights & Biases:** Learnable parameters that define the network's behavior. Initialized randomly.
- **Activation Function:** Introduces non-linearity (e.g., Sigmoid, ReLU, Tanh).
- **Feedforward:** Data flows strictly from input to output.

Task Specifics:

This experiment requires defining an MLP with:

- 4 binary inputs.
- One hidden layer (size can be chosen, e.g., `h_size`).
- An output layer with 2 neurons for two binary outputs.

Similar to Experiment 17, the focus is on initializing the weights and biases randomly and displaying them. No training (backpropagation) is involved.

Weight Matrix Dimensions:

- Input to Hidden Layer (W1): `(4, h_size)`
- Bias for Hidden Layer (b1): `(1, h_size)`
- Hidden Layer to Output Layer (W2): `(h_size, 2)`
- Bias for Output Layer (b2): `(1, 2)`

Pseudocode/Algorithm

```
// Define Network Structure
Input_size = 4
Define h_size (number of neurons in the hidden layer)
Output_size = 2

// Initialize Weights and Biases Randomly
Function initialize_mlp(input_size, h_size, output_size):
  // Weights
  W1 = Generate random matrix of size (input_size, h_size)
  W2 = Generate random matrix of size (h_size, output_size)

  // Biases
  b1 = Generate random vector of size (1, h_size)
  b2 = Generate random vector of size (1, output_size)

  Return W1, b1, W2, b2

// Display Function
Function display_parameters(W1, b1, W2, b2):
  Print "--- MLP Structure ---"
  Print "Input Layer Size:", 4
  Print "Hidden Layer Size:", h_size
  Print "Output Layer Size:", 2

  Print "\n--- Initial Random Weights ---"
  Print "W1 (Input -> Hidden) Matrix Shape:", shape(W1)
  Print W1
  Print "W2 (Hidden -> Output) Matrix Shape:", shape(W2)
  Print W2

  Print "\n--- Initial Random Biases ---"
  Print "b1 (Hidden) Vector Shape:", shape(b1)
  Print b1
  Print "b2 (Output) Vector Shape:", shape(b2)
  Print b2

  // Interpret "number of steps" as number of layers
  Print "\nNumber of Layers (Steps):", 3 // Input, Hidden, Output

Main:
  input_size = 4
  Set h_size (e.g., 5)
  output_size = 2

  (W1, b1, W2, b2) = initialize_mlp(input_size, h_size, output_size)

  display_parameters(W1, b1, W2, b2)
```

Python Code

```
import numpy as np

def initialize_mlp(n_inputs, n_hidden, n_outputs):
    """Initializes weights and biases for a 1-hidden layer MLP randomly.

    Args:
        n_inputs (int): Number of input features (4 for this case).
        n_hidden (int): Number of neurons in the hidden layer.
        n_outputs (int): Number of output neurons (2 for this case).

    Returns:
        tuple: Contains weight matrices (W1, W2) and bias vectors (b1, b2).
    """
    # Initialize weights with small random values
    W1 = np.random.randn(n_inputs, n_hidden)
    W2 = np.random.randn(n_hidden, n_outputs)

    # Initialize biases
    b1 = np.random.randn(1, n_hidden)
    b2 = np.random.randn(1, n_outputs)

    return W1, b1, W2, b2

def display_parameters(n_inputs, n_hidden, n_outputs, W1, b1, W2, b2):
    """Displays the network structure and initial parameters."""
    print("--- MLP Structure ---")
    print(f"Input Layer Size: {n_inputs}")
    print(f"Hidden Layer Size: {n_hidden}")
    print(f"Output Layer Size: {n_outputs}")

    print("\n--- Initial Random Weights ---")
    print(f"W1 (Input -> Hidden) Matrix Shape: {W1.shape}")
    print(W1)
    print(f"\nW2 (Hidden -> Output) Matrix Shape: {W2.shape}")
    print(W2)

    print("\n--- Initial Random Biases ---")
    print(f"b1 (Hidden) Vector Shape: {b1.shape}")
    print(b1)
    print(f"\nb2 (Output) Vector Shape: {b2.shape}")
    print(b2)

    # Interpret "number of steps" as the number of layers
    print(f"\nNumber of Layers (including input): 3")
    print(f"Number of Computational Layers (Steps): 2")

# --- Main Execution ---
if __name__ == "__main__":
    n_inputs = 4 # Fixed number of binary inputs

    # Define the size of the hidden layer (can be adjusted)
    n_hidden = 5 # Example size
    n_outputs = 2 # Two binary outputs

    print(f"Initializing MLP with Inputs={n_inputs}, Hidden={n_hidden}, Outputs={n_outputs}")

    # Initialize the MLP parameters
    W1, b1, W2, b2 = initialize_mlp(n_inputs, n_hidden, n_outputs)

    # Display the structure and initial parameters
    display_parameters(n_inputs, n_hidden, n_outputs, W1, b1, W2, b2)
```

Explanation:

1. **`initialize_mlp` Function:** Creates weight matrices `W1` (Input to Hidden) and `W2` (Hidden to Output), and bias vectors `b1` (Hidden) and `b2` (Output) using `numpy.random.randn` for random initialization.
2. **`display_parameters` Function:** Prints the network architecture (layer sizes) and the shapes and values of the initialized weights and biases.
3. **Main Block:**
    - Sets `n_inputs` to 4 and `n_outputs` to 2 as specified.
    - Sets a size for the hidden layer (`n_hidden`).
    - Calls `initialize_mlp` to get the random parameters.
    - Calls `display_parameters` to show the results.

This code sets up the required MLP structure with 4 inputs, 1 hidden layer, and 2 outputs, initializing its parameters randomly and displaying them.

Experiment 19: MLP with Backpropagation (Sigmoid Activation)

Theory

Multi-Layer Perceptron (MLP):

An MLP is a feedforward neural network with one or more hidden layers. It learns complex, non-linear relationships between inputs and outputs.

Backpropagation Algorithm:

Backpropagation is the standard algorithm for training MLPs. It's a supervised learning algorithm that adjusts the network's weights and biases to minimize the difference (error) between the network's output and the desired target output.

Steps:

1. **Initialization:** Initialize weights (W1, W2, W3) and biases (b1, b2, b3) randomly (often small values).
2. **Forward Pass:**
    - Present an input vector `X` to the network.
    - Calculate the weighted sum and apply the activation function for each neuron layer by layer:
        - Hidden Layer 1: `Z1 = X.dot(W1) + b1`, `A1 = sigmoid(Z1)`
        - Hidden Layer 2: `Z2 = A1.dot(W2) + b2`, `A2 = sigmoid(Z2)`
        - Output Layer: `Z3 = A2.dot(W3) + b3`, `Output (Y_hat) = sigmoid(Z3)`
3. **Calculate Error:** Compute the difference between the predicted output `Y_hat` and the target output `Y`. A common loss function for binary classification is Mean Squared Error (MSE) or Binary Cross-Entropy.
    - MSE: `Loss = 1/m * sum((Y - Y_hat)^2)` (where m is the number of samples)
4. **Backward Pass (Gradient Calculation):**
    - Calculate the gradient of the loss function with respect to the output layer's activation (`dLoss/dY_hat`).
    - Propagate this error backward through the network, layer by layer, calculating the gradients of the loss with respect to weights and biases.
    - **Output Layer Gradients:**
        - `dZ3 = dLoss/dY_hat * sigmoid_derivative(Z3)`
        - `dW3 = A2.T.dot(dZ3)`
        - `db3 = sum(dZ3, axis=0)`
    - **Hidden Layer 2 Gradients:**
        - `dA2 = dZ3.dot(W3.T)`
        - `dZ2 = dA2 * sigmoid_derivative(Z2)`
        - `dW2 = A1.T.dot(dZ2)`
        - `db2 = sum(dZ2, axis=0)`
    - **Hidden Layer 1 Gradients:**
        - `dA1 = dZ2.dot(W2.T)`
        - `dZ1 = dA1 * sigmoid_derivative(Z1)`
        - `dW1 = X.T.dot(dZ1)`
        - `db1 = sum(dZ1, axis=0)`
5. **Update Weights and Biases:** Adjust the weights and biases using gradient descent (or a variant like Adam, RMSprop).
    - `W = W - learning_rate * dW`
    - `b = b - learning_rate * db`
6. **Repeat:** Repeat steps 2-5 for a fixed number of epochs (iterations over the entire dataset) or until the error converges to a satisfactory level.

Sigmoid Activation Function:

- Formula: `sigmoid(x) = 1 / (1 + exp(-x))`
- Output Range: (0, 1)

- Derivative: `sigmoid_derivative(x) = sigmoid(x) * (1 - sigmoid(x))`
- Use: Often used in hidden layers (historically) and especially in the output layer for binary classification problems, as its output can be interpreted as a probability.

Task Specifics:

- N binary inputs.
- Two hidden layers.
- One output neuron.
- Use Sigmoid activation in all layers (hidden and output).
- Implement the backpropagation algorithm for training.

Pseudocode/Algorithm

```
// Define Network Structure
Input: N (number of inputs)
Define h1_size, h2_size
Output_size = 1
Learning_rate
Epochs

// Activation Function
Function sigmoid(x):
  Return 1 / (1 + exp(-x))

Function sigmoid_derivative(x): // x is the output of sigmoid(z)
  Return x * (1 - x)

// Initialize Weights and Biases
Function initialize_mlp(N, h1_size, h2_size, output_size):
  W1 = Random matrix (N, h1_size)
  b1 = Random vector (1, h1_size)
  W2 = Random matrix (h1_size, h2_size)
  b2 = Random vector (1, h2_size)
  W3 = Random matrix (h2_size, output_size)
  b3 = Random vector (1, output_size)
  Return W1, b1, W2, b2, W3, b3

// Training Function
Function train_mlp(X_train, Y_train, N, h1_size, h2_size, output_size, epochs, learning_rate):
  W1, b1, W2, b2, W3, b3 = initialize_mlp(N, h1_size, h2_size, output_size)

  For epoch from 1 to epochs:
    // --- Forward Pass ---
    Z1 = X_train.dot(W1) + b1
    A1 = sigmoid(Z1)
    Z2 = A1.dot(W2) + b2
    A2 = sigmoid(Z2)
    Z3 = A2.dot(W3) + b3
    Y_hat = sigmoid(Z3)

    // --- Calculate Error (e.g., MSE) ---
    error = Y_train - Y_hat
    loss = mean(error^2)

    // --- Backward Pass ---
    // Output Layer Gradients
    dLoss_dYhat = -2 * error / number_of_samples // Derivative of MSE
    dYhat_dZ3 = sigmoid_derivative(Y_hat) // Derivative of sigmoid(Z3) w.r.t Z3
    dZ3 = dLoss_dYhat * dYhat_dZ3

    dW3 = A2.T.dot(dZ3)
    db3 = sum(dZ3, axis=0, keepdims=True)
    dA2 = dZ3.dot(W3.T)

    // Hidden Layer 2 Gradients
    dZ2 = dA2 * sigmoid_derivative(A2) // A2 = sigmoid(Z2)
    dW2 = A1.T.dot(dZ2)
    db2 = sum(dZ2, axis=0, keepdims=True)
    dA1 = dZ2.dot(W2.T)

    // Hidden Layer 1 Gradients
    dZ1 = dA1 * sigmoid_derivative(A1) // A1 = sigmoid(Z1)
    dW1 = X_train.T.dot(dZ1)
    db1 = sum(dZ1, axis=0, keepdims=True)

    // --- Update Weights and Biases ---
    W1 = W1 - learning_rate * dW1
    b1 = b1 - learning_rate * db1
```

```
    W2 = W2 - learning_rate * dW2
    b2 = b2 - learning_rate * db2
    W3 = W3 - learning_rate * dW3
    b3 = b3 - learning_rate * db3

    If epoch % 100 == 0: // Print progress periodically
      Print "Epoch:", epoch, "Loss:", loss

  Return W1, b1, W2, b2, W3, b3

Main:
  Define N
  Generate or load training data X_train (binary inputs), Y_train (binary outputs)
  Set h1_size, h2_size
  Set epochs, learning_rate

  W1, b1, W2, b2, W3, b3 = train_mlp(X_train, Y_train, N, h1_size, h2_size, 1, epochs, learning_rate)

  Print "Training Complete."
  Print "Final Weights and Biases:"
  Print W1, b1, W2, b2, W3, b3
```

Python Code

```
import numpy as np

# Sigmoid activation function and its derivative
def sigmoid(x):
    return 1 / (1 + np.exp(-x))

def sigmoid_derivative(x): # x is the output of sigmoid
    return x * (1 - x)

class MLP_Sigmoid:
    def __init__(self, n_inputs, n_hidden1, n_hidden2, n_outputs):
        self.n_inputs = n_inputs
        self.n_hidden1 = n_hidden1
        self.n_hidden2 = n_hidden2
        self.n_outputs = n_outputs

        # Initialize weights and biases
        self.W1 = np.random.randn(self.n_inputs, self.n_hidden1) * 0.1
        self.b1 = np.zeros((1, self.n_hidden1))
        self.W2 = np.random.randn(self.n_hidden1, self.n_hidden2) * 0.1
        self.b2 = np.zeros((1, self.n_hidden2))
        self.W3 = np.random.randn(self.n_hidden2, self.n_outputs) * 0.1
        self.b3 = np.zeros((1, self.n_outputs))

        # Placeholders for activations and weighted sums during backprop
        self.Z1, self.A1 = None, None
        self.Z2, self.A2 = None, None
        self.Z3, self.Y_hat = None, None

    def forward_pass(self, X):
        self.Z1 = X.dot(self.W1) + self.b1
        self.A1 = sigmoid(self.Z1)
        self.Z2 = self.A1.dot(self.W2) + self.b2
        self.A2 = sigmoid(self.Z2)
        self.Z3 = self.A2.dot(self.W3) + self.b3
        self.Y_hat = sigmoid(self.Z3)
        return self.Y_hat

    def backward_pass(self, X, Y, learning_rate):
        m = Y.shape[0] # Number of samples

        # Calculate error and loss derivative (using MSE for simplicity)
        error = self.Y_hat - Y
        # Gradient of MSE loss w.r.t Y_hat is 2 * error / m, but the 2/m can be absorbed into learning rate
        # Or simply use 'error' if using Binary Cross Entropy derivative (Y_hat - Y)
        # Let's stick to MSE derivative for consistency with pseudocode
        dLoss_dYhat = 2 * error / m

        # Output Layer Gradients
        dYhat_dZ3 = sigmoid_derivative(self.Y_hat)
        dZ3 = dLoss_dYhat * dYhat_dZ3
        dW3 = self.A2.T.dot(dZ3)
        db3 = np.sum(dZ3, axis=0, keepdims=True)
        dA2 = dZ3.dot(self.W3.T)

        # Hidden Layer 2 Gradients
        dZ2 = dA2 * sigmoid_derivative(self.A2)
        dW2 = self.A1.T.dot(dZ2)
        db2 = np.sum(dZ2, axis=0, keepdims=True)
        dA1 = dZ2.dot(self.W2.T)

        # Hidden Layer 1 Gradients
        dZ1 = dA1 * sigmoid_derivative(self.A1)
        dW1 = X.T.dot(dZ1)
        db1 = np.sum(dZ1, axis=0, keepdims=True)
```

```
        # Update Weights and Biases
        self.W1 -= learning_rate * dW1
        self.b1 -= learning_rate * db1
        self.W2 -= learning_rate * dW2
        self.b2 -= learning_rate * db2
        self.W3 -= learning_rate * dW3
        self.b3 -= learning_rate * db3

    def train(self, X, Y, epochs, learning_rate):
        history = []
        for epoch in range(epochs):
            # Forward pass
            Y_hat = self.forward_pass(X)

            # Calculate loss (MSE)
            loss = np.mean((Y_hat - Y)**2)
            history.append(loss)

            # Backward pass and update weights
            self.backward_pass(X, Y, learning_rate)

            # Print progress
            if (epoch + 1) % 100 == 0:
                print(f"Epoch {epoch + 1}/{epochs}, Loss: {loss:.6f}")
        return history

    def predict(self, X):
        return self.forward_pass(X)

    def display_parameters(self):
        print("--- Final Weights and Biases ---")
        print(f"W1 Shape: {self.W1.shape}\n{self.W1}")
        print(f"b1 Shape: {self.b1.shape}\n{self.b1}")
        print(f"W2 Shape: {self.W2.shape}\n{self.W2}")
        print(f"b2 Shape: {self.b2.shape}\n{self.b2}")
        print(f"W3 Shape: {self.W3.shape}\n{self.W3}")
        print(f"b3 Shape: {self.b3.shape}\n{self.b3}")

# --- Main Execution ---
if __name__ == "__main__":
    # Example: XOR problem (needs N=2 inputs)
    # For N inputs, generate synthetic data
    N = 4 # Number of inputs (as required by some experiments, adjust if needed)
    num_samples = 100

    # Generate random binary input data
    X_train = np.random.randint(0, 2, size=(num_samples, N))

    # Generate synthetic binary output Y based on some logic (e.g., complex XOR-like)
    # Example: Output is 1 if the sum of inputs is odd, 0 otherwise
    Y_train = (np.sum(X_train, axis=1) % 2).reshape(-1, 1)

    print(f"Generated {num_samples} samples with N={N} inputs.")
    # print("Sample X:", X_train[:5])
    # print("Sample Y:", Y_train[:5])

    # Define network structure
    n_hidden1 = 8 # Number of neurons in first hidden layer
    n_hidden2 = 4 # Number of neurons in second hidden layer
    n_outputs = 1 # Single output neuron

    # Hyperparameters
    epochs = 5000
    learning_rate = 0.1
```

```
# Create and train the MLP
mlp = MLP_Sigmoid(N, n_hidden1, n_hidden2, n_outputs)
print("\n--- Training MLP with Sigmoid Activation ---")
loss_history = mlp.train(X_train, Y_train, epochs, learning_rate)

print("\nTraining complete.")

# Display final parameters
mlp.display_parameters()

# Optional: Make predictions on training data
predictions = mlp.predict(X_train)
# Convert probabilities to binary output (0 or 1) using a threshold
binary_predictions = (predictions > 0.5).astype(int)

# Calculate accuracy
accuracy = np.mean(binary_predictions == Y_train) * 100
print(f"\nFinal Training Accuracy: {accuracy:.2f}%")
print(f"Number of steps (epochs): {epochs}")

# Optional: Plot loss history
# import matplotlib.pyplot as plt
# plt.plot(loss_history)
# plt.title('Training Loss over Epochs')
# plt.xlabel('Epoch')
# plt.ylabel('Mean Squared Error Loss')
# plt.show()
```

Explanation:

1. **`sigmoid` and `sigmoid_derivative`:** Implement the activation function and its derivative.
2. **`MLP_Sigmoid` Class:**
    - `__init__`: Initializes the network structure, weights (small random values), and biases (zeros).
    - `forward_pass`: Computes the output of the network for given input `X`, storing intermediate values (Z1, A1, Z2, A2) needed for backpropagation.
    - `backward_pass`: Calculates the gradients of the loss with respect to all weights and biases using the chain rule and updates the parameters using gradient descent.
    - `train`: Iterates through the training data for a specified number of epochs, performing forward and backward passes and recording the loss.
    - `predict`: Performs a forward pass to get predictions for new data.
    - `display_parameters`: Shows the final learned weights and biases.
3. **Main Block:**
    - Sets `N` (number of inputs).
    - Generates synthetic binary input data `X_train` and corresponding target output `Y_train`.
    - Defines the hidden layer sizes, number of epochs, and learning rate.
    - Creates an instance of the `MLP_Sigmoid` class.
    - Calls the `train` method.
    - Calls `display_parameters` to show the final weights and biases.
    - Calculates and prints the training accuracy and the number of epochs (steps).

This code provides a complete implementation of an MLP with two hidden layers, using the Sigmoid activation function and the backpropagation algorithm for training.

Experiment 20: MLP with Backpropagation (ReLU Activation)

Theory

Multi-Layer Perceptron (MLP) & Backpropagation:

Refer to Experiment 19 for the general concepts of MLPs and the Backpropagation algorithm. This experiment modifies the activation function used in the hidden layers.

ReLU (Rectified Linear Unit) Activation Function:

- **Formula:** `ReLU(x) = max(0, x)`

- **Output Range:** [0, u221e)
- **Derivative:**
  - `ReLU_derivative(x) = 1` if `x > 0`
  - `ReLU_derivative(x) = 0` if `x <= 0`
- **Advantages:**
  - Computationally efficient (simple max operation).
  - Helps mitigate the vanishing gradient problem for positive inputs, often leading to faster training compared to Sigmoid/Tanh.
- **Disadvantages:**
  - **Dying ReLU Problem:** Neurons can become inactive if their input consistently results in a negative weighted sum, causing their output (and gradient) to be zero. This prevents weight updates for that neuron.
- **Use:** Very common activation function for hidden layers in deep neural networks.

Task Specifics:

- N binary inputs.
- Two hidden layers using **ReLU** activation.
- One output neuron using **Sigmoid** activation (suitable for binary classification).
- Implement the backpropagation algorithm for training.

Backpropagation with ReLU:

The core backpropagation process remains the same, but the calculation of gradients `dZ` for the hidden layers uses the ReLU derivative:

1. **Initialization:** Initialize weights (W1, W2, W3) and biases (b1, b2, b3).
2. **Forward Pass:**
   - Hidden Layer 1: `Z1 = X.dot(W1) + b1`, `A1 = ReLU(Z1)`
   - Hidden Layer 2: `Z2 = A1.dot(W2) + b2`, `A2 = ReLU(Z2)`
   - Output Layer: `Z3 = A2.dot(W3) + b3`, `Output (Y_hat) = sigmoid(Z3)`
3. **Calculate Error:** (e.g., MSE or Binary Cross-Entropy)
   - MSE: `Loss = 1/m * sum((Y - Y_hat)^2)`
4. **Backward Pass (Gradient Calculation):**
   - Calculate `dLoss/dY_hat`.
   - **Output Layer Gradients (Sigmoid):**
     - `dZ3 = dLoss/dY_hat * sigmoid_derivative(Y_hat)` (Derivative w.r.t Z3)
     - `dW3 = A2.T.dot(dZ3)`
     - `db3 = sum(dZ3, axis=0)`
     - `dA2 = dZ3.dot(W3.T)`
   - **Hidden Layer 2 Gradients (ReLU):**
     - `dZ2 = dA2 * ReLU_derivative(Z2)` (Note: Derivative depends on Z2, not A2)
     - `dW2 = A1.T.dot(dZ2)`
     - `db2 = sum(dZ2, axis=0)`
     - `dA1 = dZ2.dot(W2.T)`
   - **Hidden Layer 1 Gradients (ReLU):**
     - `dZ1 = dA1 * ReLU_derivative(Z1)` (Note: Derivative depends on Z1, not A1)
     - `dW1 = X.T.dot(dZ1)`
     - `db1 = sum(dZ1, axis=0)`
5. **Update Weights and Biases:**
   - `W = W - learning_rate * dW`
   - `b = b - learning_rate * db`
6. **Repeat:** Iterate steps 2-5.

Pseudocode/Algorithm

```
// Define Network Structure (same as Exp 19)
Input: N
Define h1_size, h2_size
Output_size = 1
Learning_rate
Epochs

// Activation Functions
Function sigmoid(x):
  Return 1 / (1 + exp(-x))

Function sigmoid_derivative(x): // x is the output of sigmoid(z)
  Return x * (1 - x)

Function relu(x):
  Return max(0, x)

Function relu_derivative(x): // x is the input Z to ReLU
  If x > 0:
    Return 1
  Else:
    Return 0

// Initialize Weights and Biases (same as Exp 19)
Function initialize_mlp(N, h1_size, h2_size, output_size):
  // ... (same initialization)
  Return W1, b1, W2, b2, W3, b3

// Training Function
Function train_mlp(X_train, Y_train, N, h1_size, h2_size, output_size, epochs, learning_rate):
  W1, b1, W2, b2, W3, b3 = initialize_mlp(N, h1_size, h2_size, output_size)

  For epoch from 1 to epochs:
    // --- Forward Pass --- (Using ReLU for hidden, Sigmoid for output)
    Z1 = X_train.dot(W1) + b1
    A1 = relu(Z1)
    Z2 = A1.dot(W2) + b2
    A2 = relu(Z2)
    Z3 = A2.dot(W3) + b3
    Y_hat = sigmoid(Z3) // Output layer uses sigmoid

    // --- Calculate Error (e.g., MSE) ---
    error = Y_train - Y_hat
    loss = mean(error^2)

    // --- Backward Pass ---
    // Output Layer Gradients (Sigmoid)
    dLoss_dYhat = -2 * error / number_of_samples // Derivative of MSE
    dYhat_dZ3 = sigmoid_derivative(Y_hat) // Derivative of sigmoid(Z3) w.r.t Z3
    dZ3 = dLoss_dYhat * dYhat_dZ3

    dW3 = A2.T.dot(dZ3)
    db3 = sum(dZ3, axis=0, keepdims=True)
    dA2 = dZ3.dot(W3.T)

    // Hidden Layer 2 Gradients (ReLU)
    // Apply element-wise: dA2 * derivative(Z2)
    dZ2 = dA2 * relu_derivative(Z2)
    dW2 = A1.T.dot(dZ2)
    db2 = sum(dZ2, axis=0, keepdims=True)
    dA1 = dZ2.dot(W2.T)

    // Hidden Layer 1 Gradients (ReLU)
    // Apply element-wise: dA1 * derivative(Z1)
    dZ1 = dA1 * relu_derivative(Z1)
```

```
    dW1 = X_train.T.dot(dZ1)
    db1 = sum(dZ1, axis=0, keepdims=True)

    // --- Update Weights and Biases ---
    W1 = W1 - learning_rate * dW1
    b1 = b1 - learning_rate * db1
    W2 = W2 - learning_rate * dW2
    b2 = b2 - learning_rate * db2
    W3 = W3 - learning_rate * dW3
    b3 = b3 - learning_rate * db3

    If epoch % 100 == 0:
      Print "Epoch:", epoch, "Loss:", loss

  Return W1, b1, W2, b2, W3, b3

Main:
  // ... (Similar to Exp 19, define N, data, hyperparameters)
  W1, b1, W2, b2, W3, b3 = train_mlp(X_train, Y_train, N, h1_size, h2_size, 1, epochs, learning_rate)
  // ... (Print results)
```

Python Code

```
import numpy as np

# Sigmoid activation function (for output layer)
def sigmoid(x):
    # Add clipping to prevent overflow/underflow in exp
    x_clipped = np.clip(x, -500, 500)
    return 1 / (1 + np.exp(-x_clipped))

def sigmoid_derivative(x): # x is the output of sigmoid
    return x * (1 - x)

# ReLU activation function and its derivative
def relu(x):
    return np.maximum(0, x)

def relu_derivative(x): # x is the input Z to ReLU
    return (x > 0).astype(float) # Returns 1 if x > 0, else 0

class MLP_ReLU_SigmoidOutput:
    def __init__(self, n_inputs, n_hidden1, n_hidden2, n_outputs):
        self.n_inputs = n_inputs
        self.n_hidden1 = n_hidden1
        self.n_hidden2 = n_hidden2
        self.n_outputs = n_outputs

        # Initialize weights and biases (He initialization often preferred for ReLU, but using simple randn for consistency)
        self.W1 = np.random.randn(self.n_inputs, self.n_hidden1) * np.sqrt(2. / self.n_inputs) # He init scale
        self.b1 = np.zeros((1, self.n_hidden1))
        self.W2 = np.random.randn(self.n_hidden1, self.n_hidden2) * np.sqrt(2. / self.n_hidden1) # He init scale
        self.b2 = np.zeros((1, self.n_hidden2))
        self.W3 = np.random.randn(self.n_hidden2, self.n_outputs) * 0.1 # Output layer (Sigmoid) might use smaller init
        self.b3 = np.zeros((1, self.n_outputs))

        # Placeholders for activations and weighted sums during backprop
        self.Z1, self.A1 = None, None
        self.Z2, self.A2 = None, None
        self.Z3, self.Y_hat = None, None

    def forward_pass(self, X):
        self.Z1 = X.dot(self.W1) + self.b1
        self.A1 = relu(self.Z1)
        self.Z2 = self.A1.dot(self.W2) + self.b2
        self.A2 = relu(self.Z2)
        self.Z3 = self.A2.dot(self.W3) + self.b3
        self.Y_hat = sigmoid(self.Z3) # Sigmoid output
        return self.Y_hat

    def backward_pass(self, X, Y, learning_rate):
        m = Y.shape[0] # Number of samples

        # Calculate error and loss derivative (using MSE for simplicity)
        error = self.Y_hat - Y
        dLoss_dYhat = 2 * error / m

        # Output Layer Gradients (Sigmoid)
        dYhat_dZ3 = sigmoid_derivative(self.Y_hat)
        dZ3 = dLoss_dYhat * dYhat_dZ3
        dW3 = self.A2.T.dot(dZ3)
        db3 = np.sum(dZ3, axis=0, keepdims=True)
        dA2 = dZ3.dot(self.W3.T)

        # Hidden Layer 2 Gradients (ReLU)
        dZ2 = dA2 * relu_derivative(self.Z2) # Use Z2 for ReLU derivative
        dW2 = self.A1.T.dot(dZ2)
        db2 = np.sum(dZ2, axis=0, keepdims=True)
```

```
        dA1 = dZ2.dot(self.W2.T)

        # Hidden Layer 1 Gradients (ReLU)
        dZ1 = dA1 * relu_derivative(self.Z1) # Use Z1 for ReLU derivative
        dW1 = X.T.dot(dZ1)
        db1 = np.sum(dZ1, axis=0, keepdims=True)

        # Update Weights and Biases
        self.W1 -= learning_rate * dW1
        self.b1 -= learning_rate * db1
        self.W2 -= learning_rate * dW2
        self.b2 -= learning_rate * db2
        self.W3 -= learning_rate * dW3
        self.b3 -= learning_rate * db3

    def train(self, X, Y, epochs, learning_rate):
        history = []
        for epoch in range(epochs):
            # Forward pass
            Y_hat = self.forward_pass(X)

            # Calculate loss (MSE)
            loss = np.mean((Y_hat - Y)**2)
            history.append(loss)

            # Backward pass and update weights
            self.backward_pass(X, Y, learning_rate)

            # Print progress
            if (epoch + 1) % 100 == 0:
                print(f"Epoch {epoch + 1}/{epochs}, Loss: {loss:.6f}")
        return history

    def predict(self, X):
        return self.forward_pass(X)

    def display_parameters(self):
        print("--- Final Weights and Biases ---")
        print(f"W1 Shape: {self.W1.shape}") # Optionally print weights
        print(f"b1 Shape: {self.b1.shape}")
        print(f"W2 Shape: {self.W2.shape}")
        print(f"b2 Shape: {self.b2.shape}")
        print(f"W3 Shape: {self.W3.shape}")
        print(f"b3 Shape: {self.b3.shape}")

# --- Main Execution ---
if __name__ == "__main__":
    # Example: XOR-like problem for N inputs
    N = 4 # Number of inputs
    num_samples = 100

    # Generate random binary input data
    X_train = np.random.randint(0, 2, size=(num_samples, N))

    # Generate synthetic binary output Y (e.g., Output is 1 if sum of inputs is odd)
    Y_train = (np.sum(X_train, axis=1) % 2).reshape(-1, 1)

    print(f"Generated {num_samples} samples with N={N} inputs.")

    # Define network structure
    n_hidden1 = 16 # ReLU networks might benefit from more neurons
    n_hidden2 = 8
    n_outputs = 1 # Single output neuron (Sigmoid)

    # Hyperparameters (May need tuning for ReLU)
    epochs = 3000 # Might converge faster or need different number
```

```
learning_rate = 0.05 # Often requires smaller learning rate than Sigmoid

# Create and train the MLP
mlp = MLP_ReLU_SigmoidOutput(N, n_hidden1, n_hidden2, n_outputs)
print("\n--- Training MLP with ReLU Hidden Layers & Sigmoid Output ---")
loss_history = mlp.train(X_train, Y_train, epochs, learning_rate)

print("\nTraining complete.")

# Display final parameters
mlp.display_parameters()

# Make predictions
predictions = mlp.predict(X_train)
binary_predictions = (predictions > 0.5).astype(int)

# Calculate accuracy
accuracy = np.mean(binary_predictions == Y_train) * 100
print(f"\nFinal Training Accuracy: {accuracy:.2f}%")
print(f"Number of steps (epochs): {epochs}")

# Optional: Plot loss history
# import matplotlib.pyplot as plt
# plt.plot(loss_history)
# plt.title('Training Loss over Epochs (ReLU)')
# plt.xlabel('Epoch')
# plt.ylabel('Mean Squared Error Loss')
# plt.grid(True)
# plt.show()
```

Explanation:

1. **`relu` and `relu_derivative`:** Implement the ReLU function and its derivative. Note that the derivative takes the *input* to ReLU (`Z`) as its argument.
2. **`sigmoid` and `sigmoid_derivative`:** Kept for the output layer.
3. **`MLP_ReLU_SigmoidOutput` Class:**
    - `__init__`: Initializes the network. He initialization (`np.sqrt(2. / n_inputs)`) is used for weights connected to ReLU layers, which is often recommended.
    - `forward_pass`: Uses `relu` for hidden layers (`A1`, `A2`) and `sigmoid` for the output layer (`Y_hat`).
    - `backward_pass`: Calculates gradients. The key change is using `relu_derivative(self.Z1)` and `relu_derivative(self.Z2)` when calculating `dZ1` and `dZ2` respectively. The output layer gradient calculation remains the same as it depends on the Sigmoid activation.
    - `train`, `predict`, `display_parameters`: Similar functionality to the Sigmoid version.
4. **Main Block:**
    - Sets up the data and network parameters.
    - Note that hyperparameters like hidden layer sizes, `epochs`, and `learning_rate` might need different values compared to the Sigmoid network for optimal performance. ReLU networks can sometimes train faster but might require smaller learning rates to avoid divergence.
    - Trains the network and evaluates accuracy.

This code implements an MLP using ReLU in the hidden layers and Sigmoid in the output layer, trained with backpropagation.

Experiment 21: MLP with Backpropagation (Tanh Activation)

Theory

Multi-Layer Perceptron (MLP) & Backpropagation:

Refer to Experiment 19 for the general concepts of MLPs and the Backpropagation algorithm. This experiment modifies the activation function used in the hidden layers.

Tanh (Hyperbolic Tangent) Activation Function:

- **Formula:** `tanh(x) = (exp(x) - exp(-x)) / (exp(x) + exp(-x))`
- **Output Range:** (-1, 1)

- **Derivative:** `tanh_derivative(x) = 1 - tanh(x)^2`
- **Advantages:**
  - Zero-centered output (unlike Sigmoid), which can help with gradient flow during training.
  - Stronger gradients than Sigmoid for inputs close to zero, potentially leading to faster convergence.
- **Disadvantages:**
  - Still suffers from the vanishing gradient problem for large positive or negative inputs, although generally less severe than Sigmoid.
  - Computationally slightly more expensive than ReLU.
- **Use:** Was a popular choice for hidden layers before ReLU gained prominence, still used in certain architectures like LSTMs/GRUs.

Task Specifics:

- N binary inputs.
- Two hidden layers using **Tanh** activation.
- One output neuron using **Sigmoid** activation (suitable for binary classification).
- Implement the backpropagation algorithm for training.

Backpropagation with Tanh:

The core backpropagation process remains the same, but the calculation of gradients `dZ` for the hidden layers uses the Tanh derivative:

1. **Initialization:** Initialize weights (W1, W2, W3) and biases (b1, b2, b3).
2. **Forward Pass:**
   - Hidden Layer 1: `Z1 = X.dot(W1) + b1`, `A1 = tanh(Z1)`
   - Hidden Layer 2: `Z2 = A1.dot(W2) + b2`, `A2 = tanh(Z2)`
   - Output Layer: `Z3 = A2.dot(W3) + b3`, `Output (Y_hat) = sigmoid(Z3)`
3. **Calculate Error:** (e.g., MSE)
   - MSE: `Loss = 1/m * sum((Y - Y_hat)^2)`
4. **Backward Pass (Gradient Calculation):**
   - Calculate `dLoss/dY_hat`.
   - **Output Layer Gradients (Sigmoid):**
     - `dZ3 = dLoss/dY_hat * sigmoid_derivative(Y_hat)`
     - `dW3 = A2.T.dot(dZ3)`
     - `db3 = sum(dZ3, axis=0)`
     - `dA2 = dZ3.dot(W3.T)`
   - **Hidden Layer 2 Gradients (Tanh):**
     - `dZ2 = dA2 * tanh_derivative(A2)` (Note: Derivative depends on A2, the output of tanh)
     - `dW2 = A1.T.dot(dZ2)`
     - `db2 = sum(dZ2, axis=0)`
     - `dA1 = dZ2.dot(W2.T)`
   - **Hidden Layer 1 Gradients (Tanh):**
     - `dZ1 = dA1 * tanh_derivative(A1)` (Note: Derivative depends on A1, the output of tanh)
     - `dW1 = X.T.dot(dZ1)`
     - `db1 = sum(dZ1, axis=0)`
5. **Update Weights and Biases:**
   - `W = W - learning_rate * dW`
   - `b = b - learning_rate * db`
6. **Repeat:** Iterate steps 2-5.

Pseudocode/Algorithm

```
// Define Network Structure (same as Exp 19)
Input: N
Define h1_size, h2_size
Output_size = 1
Learning_rate
Epochs

// Activation Functions
Function sigmoid(x):
  Return 1 / (1 + exp(-x))

Function sigmoid_derivative(x): // x is the output of sigmoid(z)
  Return x * (1 - x)

Function tanh(x):
  Return (exp(x) - exp(-x)) / (exp(x) + exp(-x))

Function tanh_derivative(x): // x is the output of tanh(z)
  Return 1 - x^2

// Initialize Weights and Biases (same as Exp 19)
Function initialize_mlp(N, h1_size, h2_size, output_size):
  // ... (same initialization, maybe Xavier/Glorot for Tanh)
  Return W1, b1, W2, b2, W3, b3

// Training Function
Function train_mlp(X_train, Y_train, N, h1_size, h2_size, output_size, epochs, learning_rate):
  W1, b1, W2, b2, W3, b3 = initialize_mlp(N, h1_size, h2_size, output_size)

  For epoch from 1 to epochs:
    // --- Forward Pass --- (Using Tanh for hidden, Sigmoid for output)
    Z1 = X_train.dot(W1) + b1
    A1 = tanh(Z1)
    Z2 = A1.dot(W2) + b2
    A2 = tanh(Z2)
    Z3 = A2.dot(W3) + b3
    Y_hat = sigmoid(Z3) // Output layer uses sigmoid

    // --- Calculate Error (e.g., MSE) ---
    error = Y_train - Y_hat
    loss = mean(error^2)

    // --- Backward Pass ---
    // Output Layer Gradients (Sigmoid)
    dLoss_dYhat = -2 * error / number_of_samples // Derivative of MSE
    dYhat_dZ3 = sigmoid_derivative(Y_hat) // Derivative of sigmoid(Z3) w.r.t Z3
    dZ3 = dLoss_dYhat * dYhat_dZ3

    dW3 = A2.T.dot(dZ3)
    db3 = sum(dZ3, axis=0, keepdims=True)
    dA2 = dZ3.dot(W3.T)

    // Hidden Layer 2 Gradients (Tanh)
    // Apply element-wise: dA2 * derivative(A2)
    dZ2 = dA2 * tanh_derivative(A2)
    dW2 = A1.T.dot(dZ2)
    db2 = sum(dZ2, axis=0, keepdims=True)
    dA1 = dZ2.dot(W2.T)

    // Hidden Layer 1 Gradients (Tanh)
    // Apply element-wise: dA1 * derivative(A1)
    dZ1 = dA1 * tanh_derivative(A1)
    dW1 = X_train.T.dot(dZ1)
    db1 = sum(dZ1, axis=0, keepdims=True)
```

```
    // --- Update Weights and Biases ---
    W1 = W1 - learning_rate * dW1
    b1 = b1 - learning_rate * db1
    W2 = W2 - learning_rate * dW2
    b2 = b2 - learning_rate * db2
    W3 = W3 - learning_rate * dW3
    b3 = b3 - learning_rate * db3

    If epoch % 100 == 0:
      Print "Epoch:", epoch, "Loss:", loss

  Return W1, b1, W2, b2, W3, b3

Main:
  // ... (Similar to Exp 19/20, define N, data, hyperparameters)
  W1, b1, W2, b2, W3, b3 = train_mlp(X_train, Y_train, N, h1_size, h2_size, 1, epochs, learning_rate)
  // ... (Print results)
```

Python Code

```
import numpy as np

# Sigmoid activation function (for output layer)
def sigmoid(x):
    # Add clipping to prevent overflow/underflow in exp
    x_clipped = np.clip(x, -500, 500)
    return 1 / (1 + np.exp(-x_clipped))

def sigmoid_derivative(x): # x is the output of sigmoid
    return x * (1 - x)

# Tanh activation function and its derivative
def tanh(x):
    # Add clipping to prevent overflow/underflow in exp
    x_clipped = np.clip(x, -500, 500)
    return np.tanh(x_clipped)

def tanh_derivative(x): # x is the output of tanh
    return 1 - x**2

class MLP_Tanh_SigmoidOutput:
    def __init__(self, n_inputs, n_hidden1, n_hidden2, n_outputs):
        self.n_inputs = n_inputs
        self.n_hidden1 = n_hidden1
        self.n_hidden2 = n_hidden2
        self.n_outputs = n_outputs

        # Initialize weights and biases (Xavier/Glorot initialization often preferred for Tanh)
        limit1 = np.sqrt(6. / (self.n_inputs + self.n_hidden1))
        self.W1 = np.random.uniform(-limit1, limit1, (self.n_inputs, self.n_hidden1))
        self.b1 = np.zeros((1, self.n_hidden1))

        limit2 = np.sqrt(6. / (self.n_hidden1 + self.n_hidden2))
        self.W2 = np.random.uniform(-limit2, limit2, (self.n_hidden1, self.n_hidden2))
        self.b2 = np.zeros((1, self.n_hidden2))

        # Output layer (Sigmoid) might use smaller init or Xavier
        limit3 = np.sqrt(6. / (self.n_hidden2 + self.n_outputs))
        self.W3 = np.random.uniform(-limit3, limit3, (self.n_hidden2, self.n_outputs))
        # self.W3 = np.random.randn(self.n_hidden2, self.n_outputs) * 0.1 # Alternative
        self.b3 = np.zeros((1, self.n_outputs))

        # Placeholders for activations and weighted sums during backprop
        self.Z1, self.A1 = None, None
        self.Z2, self.A2 = None, None
        self.Z3, self.Y_hat = None, None

    def forward_pass(self, X):
        self.Z1 = X.dot(self.W1) + self.b1
        self.A1 = tanh(self.Z1)
        self.Z2 = self.A1.dot(self.W2) + self.b2
        self.A2 = tanh(self.Z2)
        self.Z3 = self.A2.dot(self.W3) + self.b3
        self.Y_hat = sigmoid(self.Z3) # Sigmoid output
        return self.Y_hat

    def backward_pass(self, X, Y, learning_rate):
        m = Y.shape[0] # Number of samples

        # Calculate error and loss derivative (using MSE for simplicity)
        error = self.Y_hat - Y
        dLoss_dYhat = 2 * error / m

        # Output Layer Gradients (Sigmoid)
        dYhat_dZ3 = sigmoid_derivative(self.Y_hat)
```

```
        dZ3 = dLoss_dYhat * dYhat_dZ3
        dW3 = self.A2.T.dot(dZ3)
        db3 = np.sum(dZ3, axis=0, keepdims=True)
        dA2 = dZ3.dot(self.W3.T)

        # Hidden Layer 2 Gradients (Tanh)
        dZ2 = dA2 * tanh_derivative(self.A2) # Use A2 for Tanh derivative
        dW2 = self.A1.T.dot(dZ2)
        db2 = np.sum(dZ2, axis=0, keepdims=True)
        dA1 = dZ2.dot(self.W2.T)

        # Hidden Layer 1 Gradients (Tanh)
        dZ1 = dA1 * tanh_derivative(self.A1) # Use A1 for Tanh derivative
        dW1 = X.T.dot(dZ1)
        db1 = np.sum(dZ1, axis=0, keepdims=True)

        # Update Weights and Biases
        self.W1 -= learning_rate * dW1
        self.b1 -= learning_rate * db1
        self.W2 -= learning_rate * dW2
        self.b2 -= learning_rate * db2
        self.W3 -= learning_rate * dW3
        self.b3 -= learning_rate * db3

    def train(self, X, Y, epochs, learning_rate):
        history = []
        for epoch in range(epochs):
            # Forward pass
            Y_hat = self.forward_pass(X)

            # Calculate loss (MSE)
            loss = np.mean((Y_hat - Y)**2)
            history.append(loss)

            # Backward pass and update weights
            self.backward_pass(X, Y, learning_rate)

            # Print progress
            if (epoch + 1) % 100 == 0:
                print(f"Epoch {epoch + 1}/{epochs}, Loss: {loss:.6f}")
        return history

    def predict(self, X):
        return self.forward_pass(X)

    def display_parameters(self):
        print("--- Final Weights and Biases ---")
        print(f"W1 Shape: {self.W1.shape}") # Optionally print weights
        print(f"b1 Shape: {self.b1.shape}")
        print(f"W2 Shape: {self.W2.shape}")
        print(f"b2 Shape: {self.b2.shape}")
        print(f"W3 Shape: {self.W3.shape}")
        print(f"b3 Shape: {self.b3.shape}")

# --- Main Execution ---
if __name__ == "__main__":
    # Example: XOR-like problem for N inputs
    N = 4 # Number of inputs
    num_samples = 100

    # Generate random binary input data
    X_train = np.random.randint(0, 2, size=(num_samples, N))

    # Generate synthetic binary output Y (e.g., Output is 1 if sum of inputs is odd)
    Y_train = (np.sum(X_train, axis=1) % 2).reshape(-1, 1)
```

```
print(f"Generated {num_samples} samples with N={N} inputs.")

# Define network structure
n_hidden1 = 10 # Tanh might work well with fewer neurons than ReLU sometimes
n_hidden2 = 5
n_outputs = 1 # Single output neuron (Sigmoid)

# Hyperparameters (May need tuning for Tanh)
epochs = 2000
learning_rate = 0.1 # Tanh can sometimes handle slightly larger learning rates than Sigmoid

# Create and train the MLP
mlp = MLP_Tanh_SigmoidOutput(N, n_hidden1, n_hidden2, n_outputs)
print("\n--- Training MLP with Tanh Hidden Layers & Sigmoid Output ---")
loss_history = mlp.train(X_train, Y_train, epochs, learning_rate)

print("\nTraining complete.")

# Display final parameters
mlp.display_parameters()

# Make predictions
predictions = mlp.predict(X_train)
binary_predictions = (predictions > 0.5).astype(int)

# Calculate accuracy
accuracy = np.mean(binary_predictions == Y_train) * 100
print(f"\nFinal Training Accuracy: {accuracy:.2f}%")
print(f"Number of steps (epochs): {epochs}")

# Optional: Plot loss history
# import matplotlib.pyplot as plt
# plt.plot(loss_history)
# plt.title('Training Loss over Epochs (Tanh)')
# plt.xlabel('Epoch')
# plt.ylabel('Mean Squared Error Loss')
# plt.grid(True)
# plt.show()
```

Explanation:

1. **`tanh` and `tanh_derivative`:** Implement the Tanh function and its derivative. Note that the derivative takes the *output* of Tanh (`A`) as its argument, unlike ReLU's derivative which takes the input (`Z`). Clipping is added to `tanh` to prevent potential numerical issues with `exp`.
2. **`sigmoid` and `sigmoid_derivative`:** Kept for the output layer.
3. **`MLP_Tanh_SigmoidOutput` Class:**
   - `__init__`: Initializes the network. Xavier/Glorot initialization (`np.sqrt(6. / (n_in + n_out))`) is used, which is often recommended for Tanh layers.
   - `forward_pass`: Uses `tanh` for hidden layers (`A1`, `A2`) and `sigmoid` for the output layer (`Y_hat`).
   - `backward_pass`: Calculates gradients. The key change is using `tanh_derivative(self.A1)` and `tanh_derivative(self.A2)` when calculating `dZ1` and `dZ2` respectively. The output layer gradient calculation remains the same.
   - `train`, `predict`, `display_parameters`: Similar functionality to previous versions.
4. **Main Block:**
   - Sets up the data and network parameters.
   - Hyperparameters like hidden layer sizes, `epochs`, and `learning_rate` might need different values compared to Sigmoid or ReLU networks.
   - Trains the network and evaluates accuracy.

This code implements an MLP using Tanh in the hidden layers and Sigmoid in the output layer, trained with backpropagation.

Experiment 22: MLP with Backpropagation (Leaky ReLU Activation)

Theory

Multi-Layer Perceptron (MLP) & Backpropagation:

Refer to Experiment 19 for the general concepts of MLPs and the Backpropagation algorithm. This experiment modifies the activation function used in the hidden layers.

Leaky ReLU (Leaky Rectified Linear Unit) Activation Function:

- **Formula:** `LeakyReLU(x) = max(alpha * x, x)` where `alpha` is a small positive constant (e.g., 0.01).
  - If `x > 0`, `LeakyReLU(x) = x`
  - If `x <= 0`, `LeakyReLU(x) = alpha * x`
- **Output Range:** (-u221e, u221e)
- **Derivative:**
  - `LeakyReLU_derivative(x) = 1` if `x > 0`
  - `LeakyReLU_derivative(x) = alpha` if `x <= 0`
- **Advantages:**
  - Addresses the "Dying ReLU" problem by allowing a small, non-zero gradient when the unit is not active (`x <= 0`).
  - Retains the computational efficiency of ReLU for positive inputs.
- **Disadvantages:**
  - Results are not always consistently better than standard ReLU.
  - Introduces an extra hyperparameter (`alpha`) to tune.
- **Use:** An alternative to ReLU, particularly when encountering the dying ReLU issue.

Task Specifics:

- N binary inputs.
- Two hidden layers using **Leaky ReLU** activation (with a chosen `alpha`).
- One output neuron using **Sigmoid** activation (suitable for binary classification).
- Implement the backpropagation algorithm for training.

Backpropagation with Leaky ReLU:

The core backpropagation process remains the same, but the calculation of gradients `dZ` for the hidden layers uses the Leaky ReLU derivative:

1. **Initialization:** Initialize weights (W1, W2, W3) and biases (b1, b2, b3).
2. **Forward Pass:**
   - Hidden Layer 1: `Z1 = X.dot(W1) + b1`, `A1 = LeakyReLU(Z1)`
   - Hidden Layer 2: `Z2 = A1.dot(W2) + b2`, `A2 = LeakyReLU(Z2)`
   - Output Layer: `Z3 = A2.dot(W3) + b3`, `Output (Y_hat) = sigmoid(Z3)`
3. **Calculate Error:** (e.g., MSE)
   - MSE: `Loss = 1/m * sum((Y - Y_hat)^2)`
4. **Backward Pass (Gradient Calculation):**
   - Calculate `dLoss/dY_hat`.
   - **Output Layer Gradients (Sigmoid):**
     - `dZ3 = dLoss/dY_hat * sigmoid_derivative(Y_hat)`
     - `dW3 = A2.T.dot(dZ3)`
     - `db3 = sum(dZ3, axis=0)`
     - `dA2 = dZ3.dot(W3.T)`
   - **Hidden Layer 2 Gradients (Leaky ReLU):**
     - `dZ2 = dA2 * LeakyReLU_derivative(Z2)` (Note: Derivative depends on Z2)
     - `dW2 = A1.T.dot(dZ2)`
     - `db2 = sum(dZ2, axis=0)`
     - `dA1 = dZ2.dot(W2.T)`
   - **Hidden Layer 1 Gradients (Leaky ReLU):**
     - `dZ1 = dA1 * LeakyReLU_derivative(Z1)` (Note: Derivative depends on Z1)
     - `dW1 = X.T.dot(dZ1)`
     - `db1 = sum(dZ1, axis=0)`
5. **Update Weights and Biases:**
   - `W = W - learning_rate * dW`
   - `b = b - learning_rate * db`
6. **Repeat:** Iterate steps 2-5.

Pseudocode/Algorithm

```
// Define Network Structure (same as Exp 19)
Input: N
Define h1_size, h2_size
Output_size = 1
Learning_rate
Epochs
Alpha = 0.01 // Leaky ReLU parameter

// Activation Functions
Function sigmoid(x):
  Return 1 / (1 + exp(-x))

Function sigmoid_derivative(x): // x is the output of sigmoid(z)
  Return x * (1 - x)

Function leaky_relu(x, alpha):
  Return max(alpha * x, x)

Function leaky_relu_derivative(x, alpha): // x is the input Z to Leaky ReLU
  If x > 0:
    Return 1
  Else:
    Return alpha

// Initialize Weights and Biases (same as Exp 20 - He init often suitable)
Function initialize_mlp(N, h1_size, h2_size, output_size):
  // ... (He initialization recommended)
  Return W1, b1, W2, b2, W3, b3

// Training Function
Function train_mlp(X_train, Y_train, N, h1_size, h2_size, output_size, epochs, learning_rate, alpha):
  W1, b1, W2, b2, W3, b3 = initialize_mlp(N, h1_size, h2_size, output_size)

  For epoch from 1 to epochs:
    // --- Forward Pass --- (Using Leaky ReLU for hidden, Sigmoid for output)
    Z1 = X_train.dot(W1) + b1
    A1 = leaky_relu(Z1, alpha)
    Z2 = A1.dot(W2) + b2
    A2 = leaky_relu(Z2, alpha)
    Z3 = A2.dot(W3) + b3
    Y_hat = sigmoid(Z3) // Output layer uses sigmoid

    // --- Calculate Error (e.g., MSE) ---
    error = Y_train - Y_hat
    loss = mean(error^2)

    // --- Backward Pass ---
    // Output Layer Gradients (Sigmoid)
    dLoss_dYhat = -2 * error / number_of_samples // Derivative of MSE
    dYhat_dZ3 = sigmoid_derivative(Y_hat) // Derivative of sigmoid(Z3) w.r.t Z3
    dZ3 = dLoss_dYhat * dYhat_dZ3

    dW3 = A2.T.dot(dZ3)
    db3 = sum(dZ3, axis=0, keepdims=True)
    dA2 = dZ3.dot(W3.T)

    // Hidden Layer 2 Gradients (Leaky ReLU)
    // Apply element-wise: dA2 * derivative(Z2, alpha)
    dZ2 = dA2 * leaky_relu_derivative(Z2, alpha)
    dW2 = A1.T.dot(dZ2)
    db2 = sum(dZ2, axis=0, keepdims=True)
    dA1 = dZ2.dot(W2.T)

    // Hidden Layer 1 Gradients (Leaky ReLU)
    // Apply element-wise: dA1 * derivative(Z1, alpha)
```

```
    dZ1 = dA1 * leaky_relu_derivative(Z1, alpha)
    dW1 = X_train.T.dot(dZ1)
    db1 = sum(dZ1, axis=0, keepdims=True)

    // --- Update Weights and Biases ---
    W1 = W1 - learning_rate * dW1
    b1 = b1 - learning_rate * db1
    W2 = W2 - learning_rate * dW2
    b2 = b2 - learning_rate * db2
    W3 = W3 - learning_rate * dW3
    b3 = b3 - learning_rate * db3

    If epoch % 100 == 0:
      Print "Epoch:", epoch, "Loss:", loss

  Return W1, b1, W2, b2, W3, b3

Main:
  // ... (Similar to Exp 19/20/21, define N, data, hyperparameters, alpha)
  W1, b1, W2, b2, W3, b3 = train_mlp(X_train, Y_train, N, h1_size, h2_size, 1, epochs, learning_rate, alpha)
  // ... (Print results)
```

Python Code

```
import numpy as np

# Sigmoid activation function (for output layer)
def sigmoid(x):
    x_clipped = np.clip(x, -500, 500)
    return 1 / (1 + np.exp(-x_clipped))

def sigmoid_derivative(x): # x is the output of sigmoid
    return x * (1 - x)

# Leaky ReLU activation function and its derivative
def leaky_relu(x, alpha=0.01):
    return np.where(x > 0, x, x * alpha)

def leaky_relu_derivative(x, alpha=0.01): # x is the input Z to Leaky ReLU
    return np.where(x > 0, 1, alpha)

class MLP_LeakyReLU_SigmoidOutput:
    def __init__(self, n_inputs, n_hidden1, n_hidden2, n_outputs, alpha=0.01):
        self.n_inputs = n_inputs
        self.n_hidden1 = n_hidden1
        self.n_hidden2 = n_hidden2
        self.n_outputs = n_outputs
        self.alpha = alpha # Store alpha

        # Initialize weights and biases (He initialization often suitable for ReLU variants)
        self.W1 = np.random.randn(self.n_inputs, self.n_hidden1) * np.sqrt(2. / self.n_inputs)
        self.b1 = np.zeros((1, self.n_hidden1))
        self.W2 = np.random.randn(self.n_hidden1, self.n_hidden2) * np.sqrt(2. / self.n_hidden1)
        self.b2 = np.zeros((1, self.n_hidden2))
        self.W3 = np.random.randn(self.n_hidden2, self.n_outputs) * 0.1 # Output layer (Sigmoid)
        self.b3 = np.zeros((1, self.n_outputs))

        # Placeholders
        self.Z1, self.A1 = None, None
        self.Z2, self.A2 = None, None
        self.Z3, self.Y_hat = None, None

    def forward_pass(self, X):
        self.Z1 = X.dot(self.W1) + self.b1
        self.A1 = leaky_relu(self.Z1, self.alpha)
        self.Z2 = self.A1.dot(self.W2) + self.b2
        self.A2 = leaky_relu(self.Z2, self.alpha)
        self.Z3 = self.A2.dot(self.W3) + self.b3
        self.Y_hat = sigmoid(self.Z3) # Sigmoid output
        return self.Y_hat

    def backward_pass(self, X, Y, learning_rate):
        m = Y.shape[0]
        error = self.Y_hat - Y
        dLoss_dYhat = 2 * error / m

        # Output Layer Gradients (Sigmoid)
        dYhat_dZ3 = sigmoid_derivative(self.Y_hat)
        dZ3 = dLoss_dYhat * dYhat_dZ3
        dW3 = self.A2.T.dot(dZ3)
        db3 = np.sum(dZ3, axis=0, keepdims=True)
        dA2 = dZ3.dot(self.W3.T)

        # Hidden Layer 2 Gradients (Leaky ReLU)
        dZ2 = dA2 * leaky_relu_derivative(self.Z2, self.alpha) # Use Z2 and alpha
        dW2 = self.A1.T.dot(dZ2)
        db2 = np.sum(dZ2, axis=0, keepdims=True)
        dA1 = dZ2.dot(self.W2.T)
```

```
        # Hidden Layer 1 Gradients (Leaky ReLU)
        dZ1 = dA1 * leaky_relu_derivative(self.Z1, self.alpha) # Use Z1 and alpha
        dW1 = X.T.dot(dZ1)
        db1 = np.sum(dZ1, axis=0, keepdims=True)

        # Update Weights and Biases
        self.W1 -= learning_rate * dW1
        self.b1 -= learning_rate * db1
        self.W2 -= learning_rate * dW2
        self.b2 -= learning_rate * db2
        self.W3 -= learning_rate * dW3
        self.b3 -= learning_rate * db3

    def train(self, X, Y, epochs, learning_rate):
        history = []
        for epoch in range(epochs):
            Y_hat = self.forward_pass(X)
            loss = np.mean((Y_hat - Y)**2)
            history.append(loss)
            self.backward_pass(X, Y, learning_rate)
            if (epoch + 1) % 100 == 0:
                print(f"Epoch {epoch + 1}/{epochs}, Loss: {loss:.6f}")
        return history

    def predict(self, X):
        return self.forward_pass(X)

    def display_parameters(self):
        print("--- Final Weights and Biases ---")
        print(f"W1 Shape: {self.W1.shape}")
        print(f"b1 Shape: {self.b1.shape}")
        print(f"W2 Shape: {self.W2.shape}")
        print(f"b2 Shape: {self.b2.shape}")
        print(f"W3 Shape: {self.W3.shape}")
        print(f"b3 Shape: {self.b3.shape}")

# --- Main Execution ---
if __name__ == "__main__":
    N = 4
    num_samples = 100
    X_train = np.random.randint(0, 2, size=(num_samples, N))
    Y_train = (np.sum(X_train, axis=1) % 2).reshape(-1, 1)

    print(f"Generated {num_samples} samples with N={N} inputs.")

    n_hidden1 = 16
    n_hidden2 = 8
    n_outputs = 1
    leaky_alpha = 0.01 # Define alpha for Leaky ReLU

    epochs = 3000
    learning_rate = 0.05 # Similar LR as ReLU might work

    mlp = MLP_LeakyReLU_SigmoidOutput(N, n_hidden1, n_hidden2, n_outputs, alpha=leaky_alpha)
    print(f"\n--- Training MLP with Leaky ReLU (alpha={leaky_alpha}) Hidden Layers & Sigmoid Output ---")
    loss_history = mlp.train(X_train, Y_train, epochs, learning_rate)

    print("\nTraining complete.")
    mlp.display_parameters()

    predictions = mlp.predict(X_train)
    binary_predictions = (predictions > 0.5).astype(int)
    accuracy = np.mean(binary_predictions == Y_train) * 100
    print(f"\nFinal Training Accuracy: {accuracy:.2f}%")
    print(f"Number of steps (epochs): {epochs}")
```

```
# Optional: Plot loss history
# import matplotlib.pyplot as plt
# plt.plot(loss_history)
# plt.title(f'Training Loss over Epochs (Leaky ReLU alpha={leaky_alpha})')
# plt.xlabel('Epoch')
# plt.ylabel('Mean Squared Error Loss')
# plt.grid(True)
# plt.show()
```

Explanation:

1. **`leaky_relu` and `leaky_relu_derivative`:** Implement the Leaky ReLU function and its derivative. Both take `alpha` as an argument (defaulting to 0.01). The derivative depends on the *input* `Z`.
2. **`sigmoid` and `sigmoid_derivative`:** Kept for the output layer.
3. **`MLP_LeakyReLU_SigmoidOutput` Class:**
   - `__init__`: Initializes the network, storing the `alpha` value. He initialization is used, similar to standard ReLU.
   - `forward_pass`: Uses `leaky_relu` (passing `self.alpha`) for hidden layers (`A1`, `A2`) and `sigmoid` for the output layer (`Y_hat`).
   - `backward_pass`: Calculates gradients. The key change is using `leaky_relu_derivative(self.Z1, self.alpha)` and `leaky_relu_derivative(self.Z2, self.alpha)` when calculating `dZ1` and `dZ2` respectively.
   - `train`, `predict`, `display_parameters`: Similar functionality.
4. **Main Block:**
   - Sets up data, network parameters, and defines the `leaky_alpha`.
   - Instantiates the `MLP_LeakyReLU_SigmoidOutput` class, passing `alpha`.
   - Trains the network and evaluates accuracy.

This code implements an MLP using Leaky ReLU in the hidden layers and Sigmoid in the output layer, trained with backpropagation, demonstrating how to incorporate this ReLU variant.

Experiment 23: MLP with Backpropagation (ELU Activation)

Theory

Multi-Layer Perceptron (MLP) & Backpropagation:

Refer to Experiment 19 for the general concepts of MLPs and the Backpropagation algorithm. This experiment modifies the activation function used in the hidden layers.

ELU (Exponential Linear Unit) Activation Function:

- **Formula:**
  - `ELU(x) = x` if `x > 0`
  - `ELU(x) = alpha * (exp(x) - 1)` if `x <= 0`
  - `alpha` is a positive hyperparameter, often set to 1.0.
- **Output Range:** (`alpha * (exp(-inf) - 1)`) = `-alpha`, u221e)
- **Derivative:**
  - `ELU_derivative(x) = 1` if `x > 0`
  - `ELU_derivative(x) = ELU(x) + alpha` if `x <= 0` (Note: derivative depends on the ELU output for x<=0)
- **Advantages:**
  - Like Leaky ReLU, it addresses the dying ReLU problem by having non-zero gradients for negative inputs.
  - Produces negative outputs, which can help push the mean activation closer to zero (similar to Tanh), potentially improving learning.
  - Often reported to lead to faster convergence and better generalization than ReLU/Leaky ReLU in some tasks.
- **Disadvantages:**
  - Computationally more expensive than ReLU/Leaky ReLU due to the exponential function.
  - Introduces the `alpha` hyperparameter.
- **Use:** An alternative to ReLU and its variants, especially when seeking faster convergence or potentially better performance.

Task Specifics:

- N binary inputs.
- Two hidden layers using **ELU** activation (with a chosen `alpha`).
- One output neuron using **Sigmoid** activation (suitable for binary classification).
- Implement the backpropagation algorithm for training.

Backpropagation with ELU:

The core backpropagation process remains the same, but the calculation of gradients `dZ` for the hidden layers uses the ELU derivative:

1. **Initialization:** Initialize weights (W1, W2, W3) and biases (b1, b2, b3).
2. **Forward Pass:**
    - Hidden Layer 1: `Z1 = X.dot(W1) + b1, A1 = ELU(Z1)`
    - Hidden Layer 2: `Z2 = A1.dot(W2) + b2, A2 = ELU(Z2)`
    - Output Layer: `Z3 = A2.dot(W3) + b3, Output (Y_hat) = sigmoid(Z3)`
3. **Calculate Error:** (e.g., MSE)
4. **Backward Pass (Gradient Calculation):**
    - Calculate `dLoss/dY_hat`.
    - **Output Layer Gradients (Sigmoid):**
        - `dZ3 = dLoss/dY_hat * sigmoid_derivative(Y_hat)`
        - `dW3 = A2.T.dot(dZ3)`
        - `db3 = sum(dZ3, axis=0)`
        - `dA2 = dZ3.dot(W3.T)`
    - **Hidden Layer 2 Gradients (ELU):**
        - `dZ2 = dA2 * ELU_derivative(Z2, A2, alpha)` (Note: Derivative depends on Z2 and A2 for x<=0)
        - `dW2 = A1.T.dot(dZ2)`
        - `db2 = sum(dZ2, axis=0)`
        - `dA1 = dZ2.dot(W2.T)`
    - **Hidden Layer 1 Gradients (ELU):**
        - `dZ1 = dA1 * ELU_derivative(Z1, A1, alpha)` (Note: Derivative depends on Z1 and A1 for x<=0)
        - `dW1 = X.T.dot(dZ1)`
        - `db1 = sum(dZ1, axis=0)`
5. **Update Weights and Biases:**
    - `W = W - learning_rate * dW`
    - `b = b - learning_rate * db`
6. **Repeat:** Iterate steps 2-5.

Pseudocode/Algorithm

```
// Define Network Structure (same as Exp 19)
Input: N
Define h1_size, h2_size
Output_size = 1
Learning_rate
Epochs
Alpha = 1.0 // ELU parameter

// Activation Functions
Function sigmoid(x):
  Return 1 / (1 + exp(-x))

Function sigmoid_derivative(x): // x is the output of sigmoid(z)
  Return x * (1 - x)

Function elu(x, alpha):
  If x > 0:
    Return x
  Else:
    Return alpha * (exp(x) - 1)

Function elu_derivative(z, a, alpha): // z is input, a is elu(z)
  If z > 0:
    Return 1
  Else:
    Return a + alpha // Derivative for x <= 0 is ELU(x) + alpha

// Initialize Weights and Biases (He init often suitable)
Function initialize_mlp(N, h1_size, h2_size, output_size):
  // ... (He initialization recommended)
  Return W1, b1, W2, b2, W3, b3

// Training Function
Function train_mlp(X_train, Y_train, N, h1_size, h2_size, output_size, epochs, learning_rate, alpha):
  W1, b1, W2, b2, W3, b3 = initialize_mlp(N, h1_size, h2_size, output_size)

  For epoch from 1 to epochs:
    // --- Forward Pass --- (Using ELU for hidden, Sigmoid for output)
    Z1 = X_train.dot(W1) + b1
    A1 = elu(Z1, alpha)
    Z2 = A1.dot(W2) + b2
    A2 = elu(Z2, alpha)
    Z3 = A2.dot(W3) + b3
    Y_hat = sigmoid(Z3) // Output layer uses sigmoid

    // --- Calculate Error (e.g., MSE) ---
    error = Y_train - Y_hat
    loss = mean(error^2)

    // --- Backward Pass ---
    // Output Layer Gradients (Sigmoid)
    dLoss_dYhat = -2 * error / number_of_samples
    dYhat_dZ3 = sigmoid_derivative(Y_hat)
    dZ3 = dLoss_dYhat * dYhat_dZ3

    dW3 = A2.T.dot(dZ3)
    db3 = sum(dZ3, axis=0, keepdims=True)
    dA2 = dZ3.dot(W3.T)

    // Hidden Layer 2 Gradients (ELU)
    // Apply element-wise: dA2 * derivative(Z2, A2, alpha)
    dZ2 = dA2 * elu_derivative(Z2, A2, alpha)
    dW2 = A1.T.dot(dZ2)
    db2 = sum(dZ2, axis=0, keepdims=True)
    dA1 = dZ2.dot(W2.T)
```

```
    // Hidden Layer 1 Gradients (ELU)
    // Apply element-wise: dA1 * derivative(Z1, A1, alpha)
    dZ1 = dA1 * elu_derivative(Z1, A1, alpha)
    dW1 = X_train.T.dot(dZ1)
    db1 = sum(dZ1, axis=0, keepdims=True)

    // --- Update Weights and Biases ---
    W1 = W1 - learning_rate * dW1
    b1 = b1 - learning_rate * db1
    W2 = W2 - learning_rate * dW2
    b2 = b2 - learning_rate * db2
    W3 = W3 - learning_rate * dW3
    b3 = b3 - learning_rate * db3

    If epoch % 100 == 0:
      Print "Epoch:", epoch, "Loss:", loss

  Return W1, b1, W2, b2, W3, b3

Main:
  // ... (Similar to previous experiments, define N, data, hyperparameters, alpha)
  W1, b1, W2, b2, W3, b3 = train_mlp(X_train, Y_train, N, h1_size, h2_size, 1, epochs, learning_rate, alpha)
  // ... (Print results)
```

Python Code

```
import numpy as np

# Sigmoid activation function (for output layer)
def sigmoid(x):
    x_clipped = np.clip(x, -500, 500)
    return 1 / (1 + np.exp(-x_clipped))

def sigmoid_derivative(x): # x is the output of sigmoid
    return x * (1 - x)

# ELU activation function and its derivative
def elu(x, alpha=1.0):
    x_clipped = np.clip(x, -500, 500) # Clip input to exp
    return np.where(x > 0, x, alpha * (np.exp(x_clipped) - 1))

def elu_derivative(z, a, alpha=1.0): # z is input, a is elu(z)
    # Note: The derivative for z <= 0 is a + alpha
    return np.where(z > 0, 1, a + alpha)

class MLP_ELU_SigmoidOutput:
    def __init__(self, n_inputs, n_hidden1, n_hidden2, n_outputs, alpha=1.0):
        self.n_inputs = n_inputs
        self.n_hidden1 = n_hidden1
        self.n_hidden2 = n_hidden2
        self.n_outputs = n_outputs
        self.alpha = alpha # Store alpha

        # Initialize weights and biases (He initialization often suitable)
        self.W1 = np.random.randn(self.n_inputs, self.n_hidden1) * np.sqrt(2. / self.n_inputs)
        self.b1 = np.zeros((1, self.n_hidden1))
        self.W2 = np.random.randn(self.n_hidden1, self.n_hidden2) * np.sqrt(2. / self.n_hidden1)
        self.b2 = np.zeros((1, self.n_hidden2))
        self.W3 = np.random.randn(self.n_hidden2, self.n_outputs) * 0.1 # Output layer (Sigmoid)
        self.b3 = np.zeros((1, self.n_outputs))

        # Placeholders
        self.Z1, self.A1 = None, None
        self.Z2, self.A2 = None, None
        self.Z3, self.Y_hat = None, None

    def forward_pass(self, X):
        self.Z1 = X.dot(self.W1) + self.b1
        self.A1 = elu(self.Z1, self.alpha)
        self.Z2 = self.A1.dot(self.W2) + self.b2
        self.A2 = elu(self.Z2, self.alpha)
        self.Z3 = self.A2.dot(self.W3) + self.b3
        self.Y_hat = sigmoid(self.Z3) # Sigmoid output
        return self.Y_hat

    def backward_pass(self, X, Y, learning_rate):
        m = Y.shape[0]
        error = self.Y_hat - Y
        dLoss_dYhat = 2 * error / m

        # Output Layer Gradients (Sigmoid)
        dYhat_dZ3 = sigmoid_derivative(self.Y_hat)
        dZ3 = dLoss_dYhat * dYhat_dZ3
        dW3 = self.A2.T.dot(dZ3)
        db3 = np.sum(dZ3, axis=0, keepdims=True)
        dA2 = dZ3.dot(self.W3.T)

        # Hidden Layer 2 Gradients (ELU)
        # Pass Z2 and A2 to the derivative function
        dZ2 = dA2 * elu_derivative(self.Z2, self.A2, self.alpha)
        dW2 = self.A1.T.dot(dZ2)
```

```
        db2 = np.sum(dZ2, axis=0, keepdims=True)
        dA1 = dZ2.dot(self.W2.T)

        # Hidden Layer 1 Gradients (ELU)
        # Pass Z1 and A1 to the derivative function
        dZ1 = dA1 * elu_derivative(self.Z1, self.A1, self.alpha)
        dW1 = X.T.dot(dZ1)
        db1 = np.sum(dZ1, axis=0, keepdims=True)

        # Update Weights and Biases
        self.W1 -= learning_rate * dW1
        self.b1 -= learning_rate * db1
        self.W2 -= learning_rate * dW2
        self.b2 -= learning_rate * db2
        self.W3 -= learning_rate * dW3
        self.b3 -= learning_rate * db3

    def train(self, X, Y, epochs, learning_rate):
        history = []
        for epoch in range(epochs):
            Y_hat = self.forward_pass(X)
            loss = np.mean((Y_hat - Y)**2)
            history.append(loss)
            self.backward_pass(X, Y, learning_rate)
            if (epoch + 1) % 100 == 0:
                print(f"Epoch {epoch + 1}/{epochs}, Loss: {loss:.6f}")
        return history

    def predict(self, X):
        return self.forward_pass(X)

    def display_parameters(self):
        print("--- Final Weights and Biases ---")
        print(f"W1 Shape: {self.W1.shape}")
        print(f"b1 Shape: {self.b1.shape}")
        print(f"W2 Shape: {self.W2.shape}")
        print(f"b2 Shape: {self.b2.shape}")
        print(f"W3 Shape: {self.W3.shape}")
        print(f"b3 Shape: {self.b3.shape}")

# --- Main Execution ---
if __name__ == "__main__":
    N = 4
    num_samples = 100
    X_train = np.random.randint(0, 2, size=(num_samples, N))
    Y_train = (np.sum(X_train, axis=1) % 2).reshape(-1, 1)

    print(f"Generated {num_samples} samples with N={N} inputs.")

    n_hidden1 = 16
    n_hidden2 = 8
    n_outputs = 1
    elu_alpha = 1.0 # Define alpha for ELU (common default)

    epochs = 3000
    learning_rate = 0.05 # May need tuning

    mlp = MLP_ELU_SigmoidOutput(N, n_hidden1, n_hidden2, n_outputs, alpha=elu_alpha)
    print(f"\n--- Training MLP with ELU (alpha={elu_alpha}) Hidden Layers & Sigmoid Output ---")
    loss_history = mlp.train(X_train, Y_train, epochs, learning_rate)

    print("\nTraining complete.")
    mlp.display_parameters()

    predictions = mlp.predict(X_train)
    binary_predictions = (predictions > 0.5).astype(int)
```

```
    accuracy = np.mean(binary_predictions == Y_train) * 100
    print(f"\nFinal Training Accuracy: {accuracy:.2f}%")
    print(f"Number of steps (epochs): {epochs}")

    # Optional: Plot loss history
    # import matplotlib.pyplot as plt
    # plt.plot(loss_history)
    # plt.title(f'Training Loss over Epochs (ELU alpha={elu_alpha})')
    # plt.xlabel('Epoch')
    # plt.ylabel('Mean Squared Error Loss')
    # plt.grid(True)
    # plt.show()
```

Explanation:

1. **`elu` and `elu_derivative`:** Implement the ELU function and its derivative. Note that the derivative for `x <= 0` is `ELU(x) + alpha`, so it depends on both the input `z` (to check the condition) and the output `a = elu(z)`.
2. **`sigmoid` and `sigmoid_derivative`:** Kept for the output layer.
3. **`MLP_ELU_SigmoidOutput` Class:**
   - `__init__`: Initializes the network, storing the `alpha` value (defaulting to 1.0). He initialization is often suitable.
   - `forward_pass`: Uses `elu` (passing `self.alpha`) for hidden layers (`A1`, `A2`) and `sigmoid` for the output layer (`Y_hat`).
   - `backward_pass`: Calculates gradients. The key change is using `elu_derivative(self.Z1, self.A1, self.alpha)` and `elu_derivative(self.Z2, self.A2, self.alpha)` when calculating `dZ1` and `dZ2` respectively, passing both the pre-activation (`Z`) and activation (`A`) values.
   - `train`, `predict`, `display_parameters`: Similar functionality.
4. **Main Block:**
   - Sets up data, network parameters, and defines the `elu_alpha`.
   - Instantiates the `MLP_ELU_SigmoidOutput` class, passing `alpha`.
   - Trains the network and evaluates accuracy.

This code implements an MLP using ELU in the hidden layers and Sigmoid in the output layer, trained with backpropagation, showcasing the implementation details of ELU and its derivative.

Experiment 24: MLP with Backpropagation (Swish Activation)

Theory

Multi-Layer Perceptron (MLP) & Backpropagation:

Refer to Experiment 19 for the general concepts of MLPs and the Backpropagation algorithm. This experiment modifies the activation function used in the hidden layers.

Swish Activation Function:

- **Formula:** `Swish(x) = x * sigmoid(beta * x)`
  - `beta` is a constant or a trainable parameter. Often, `beta` is set to 1, resulting in `Swish(x) = x * sigmoid(x)`.
- **Output Range:** Approximately (-0.28, u221e) for `beta=1`.
- **Derivative (for beta=1):** `Swish_derivative(x) = Swish(x) + sigmoid(x) * (1 - Swish(x))`
  - Alternatively: `Swish_derivative(x) = sigmoid(x) + x * sigmoid_derivative(sigmoid(x))`
- **Advantages:**
  - Non-monotonic: Unlike most common activation functions, Swish can decrease even when the input increases, which might help with optimization.
  - Smooth and non-linear.
  - Often performs better than ReLU on deeper models across various tasks.
  - Unbounded above (like ReLU), avoiding saturation for large positive values.
  - Bounded below, which can help with regularization.
- **Disadvantages:**
  - Computationally more expensive than ReLU due to the sigmoid calculation.
- **Use:** A promising alternative to ReLU, developed by Google Brain.

Task Specifics:

- N binary inputs.
- Two hidden layers using **Swish** activation (with `beta=1`).

- One output neuron using **Sigmoid** activation (suitable for binary classification).
- Implement the backpropagation algorithm for training.

Backpropagation with Swish (beta=1):

The core backpropagation process remains the same, but the calculation of gradients `dZ` for the hidden layers uses the Swish derivative:

1. **Initialization:** Initialize weights (W1, W2, W3) and biases (b1, b2, b3).
2. **Forward Pass:**
    - Hidden Layer 1: `Z1 = X.dot(W1) + b1`, `A1 = Swish(Z1)`
    - Hidden Layer 2: `Z2 = A1.dot(W2) + b2`, `A2 = Swish(Z2)`
    - Output Layer: `Z3 = A2.dot(W3) + b3`, `Output (Y_hat) = sigmoid(Z3)`
3. **Calculate Error:** (e.g., MSE)
4. **Backward Pass (Gradient Calculation):**
    - Calculate `dLoss/dY_hat`.
    - **Output Layer Gradients (Sigmoid):**
        - `dZ3 = dLoss/dY_hat * sigmoid_derivative(Y_hat)`
        - `dW3 = A2.T.dot(dZ3)`
        - `db3 = sum(dZ3, axis=0)`
        - `dA2 = dZ3.dot(W3.T)`
    - **Hidden Layer 2 Gradients (Swish):**
        - `dZ2 = dA2 * Swish_derivative(Z2, A2)` (Note: Derivative depends on Z2 and A2)
        - `dW2 = A1.T.dot(dZ2)`
        - `db2 = sum(dZ2, axis=0)`
        - `dA1 = dZ2.dot(W2.T)`
    - **Hidden Layer 1 Gradients (Swish):**
        - `dZ1 = dA1 * Swish_derivative(Z1, A1)` (Note: Derivative depends on Z1 and A1)
        - `dW1 = X.T.dot(dZ1)`
        - `db1 = sum(dZ1, axis=0)`
5. **Update Weights and Biases:**
    - `W = W - learning_rate * dW`
    - `b = b - learning_rate * db`
6. **Repeat:** Iterate steps 2-5.

Pseudocode/Algorithm

```
// Define Network Structure (same as Exp 19)
Input: N
Define h1_size, h2_size
Output_size = 1
Learning_rate
Epochs
Beta = 1.0 // Swish parameter (fixed)

// Activation Functions
Function sigmoid(x):
  Return 1 / (1 + exp(-x))

Function sigmoid_derivative(x): // x is the output of sigmoid(z)
  Return x * (1 - x)

Function swish(x, beta):
  Return x * sigmoid(beta * x)

Function swish_derivative(z, a, beta): // z is input, a is swish(z)
  sig_beta_z = sigmoid(beta * z)
  Return a + sig_beta_z * (1 - a) // Using the formula: Swish(x) + sigmoid(beta*x)*(1-Swish(x))
  // Alternative: return sig_beta_z + z * beta * sigmoid_derivative(sig_beta_z)

// Initialize Weights and Biases (He init often suitable)
Function initialize_mlp(N, h1_size, h2_size, output_size):
  // ... (He initialization recommended)
  Return W1, b1, W2, b2, W3, b3

// Training Function
Function train_mlp(X_train, Y_train, N, h1_size, h2_size, output_size, epochs, learning_rate, beta):
  W1, b1, W2, b2, W3, b3 = initialize_mlp(N, h1_size, h2_size, output_size)

  For epoch from 1 to epochs:
    // --- Forward Pass --- (Using Swish for hidden, Sigmoid for output)
    Z1 = X_train.dot(W1) + b1
    A1 = swish(Z1, beta)
    Z2 = A1.dot(W2) + b2
    A2 = swish(Z2, beta)
    Z3 = A2.dot(W3) + b3
    Y_hat = sigmoid(Z3) // Output layer uses sigmoid

    // --- Calculate Error (e.g., MSE) ---
    error = Y_train - Y_hat
    loss = mean(error^2)

    // --- Backward Pass ---
    // Output Layer Gradients (Sigmoid)
    dLoss_dYhat = -2 * error / number_of_samples
    dYhat_dZ3 = sigmoid_derivative(Y_hat)
    dZ3 = dLoss_dYhat * dYhat_dZ3

    dW3 = A2.T.dot(dZ3)
    db3 = sum(dZ3, axis=0, keepdims=True)
    dA2 = dZ3.dot(W3.T)

    // Hidden Layer 2 Gradients (Swish)
    // Apply element-wise: dA2 * derivative(Z2, A2, beta)
    dZ2 = dA2 * swish_derivative(Z2, A2, beta)
    dW2 = A1.T.dot(dZ2)
    db2 = sum(dZ2, axis=0, keepdims=True)
    dA1 = dZ2.dot(W2.T)

    // Hidden Layer 1 Gradients (Swish)
    // Apply element-wise: dA1 * derivative(Z1, A1, beta)
    dZ1 = dA1 * swish_derivative(Z1, A1, beta)
```

```
    dW1 = X_train.T.dot(dZ1)
    db1 = sum(dZ1, axis=0, keepdims=True)

    // --- Update Weights and Biases ---
    W1 = W1 - learning_rate * dW1
    b1 = b1 - learning_rate * db1
    W2 = W2 - learning_rate * dW2
    b2 = b2 - learning_rate * db2
    W3 = W3 - learning_rate * dW3
    b3 = b3 - learning_rate * db3

    If epoch % 100 == 0:
      Print "Epoch:", epoch, "Loss:", loss

  Return W1, b1, W2, b2, W3, b3

Main:
  // ... (Similar to previous experiments, define N, data, hyperparameters, beta=1)
  W1, b1, W2, b2, W3, b3 = train_mlp(X_train, Y_train, N, h1_size, h2_size, 1, epochs, learning_rate, 1.0)
  // ... (Print results)
```

Python Code

```
import numpy as np

# Sigmoid activation function (used by Swish and output layer)
def sigmoid(x):
    x_clipped = np.clip(x, -500, 500)
    return 1 / (1 + np.exp(-x_clipped))

def sigmoid_derivative(x): # x is the output of sigmoid
    return x * (1 - x)

# Swish activation function and its derivative (beta=1)
def swish(x, beta=1.0):
    return x * sigmoid(beta * x)

def swish_derivative(z, a, beta=1.0): # z is input, a is swish(z)
    sig_beta_z = sigmoid(beta * z)
    # Using the formula: Swish(x) + sigmoid(beta*x)*(1-Swish(x))
    # Which simplifies to: a + sig_beta_z * (1 - a)
    return a + sig_beta_z * (1 - a)
    # Alternative implementation:
    # return sig_beta_z + z * beta * sigmoid_derivative(sig_beta_z)

class MLP_Swish_SigmoidOutput:
    def __init__(self, n_inputs, n_hidden1, n_hidden2, n_outputs, beta=1.0):
        self.n_inputs = n_inputs
        self.n_hidden1 = n_hidden1
        self.n_hidden2 = n_hidden2
        self.n_outputs = n_outputs
        self.beta = beta # Store beta

        # Initialize weights and biases (He initialization often suitable)
        self.W1 = np.random.randn(self.n_inputs, self.n_hidden1) * np.sqrt(2. / self.n_inputs)
        self.b1 = np.zeros((1, self.n_hidden1))
        self.W2 = np.random.randn(self.n_hidden1, self.n_hidden2) * np.sqrt(2. / self.n_hidden1)
        self.b2 = np.zeros((1, self.n_hidden2))
        self.W3 = np.random.randn(self.n_hidden2, self.n_outputs) * 0.1 # Output layer (Sigmoid)
        self.b3 = np.zeros((1, self.n_outputs))

        # Placeholders
        self.Z1, self.A1 = None, None
        self.Z2, self.A2 = None, None
        self.Z3, self.Y_hat = None, None

    def forward_pass(self, X):
        self.Z1 = X.dot(self.W1) + self.b1
        self.A1 = swish(self.Z1, self.beta)
        self.Z2 = self.A1.dot(self.W2) + self.b2
        self.A2 = swish(self.Z2, self.beta)
        self.Z3 = self.A2.dot(self.W3) + self.b3
        self.Y_hat = sigmoid(self.Z3) # Sigmoid output
        return self.Y_hat

    def backward_pass(self, X, Y, learning_rate):
        m = Y.shape[0]
        error = self.Y_hat - Y
        dLoss_dYhat = 2 * error / m

        # Output Layer Gradients (Sigmoid)
        dYhat_dZ3 = sigmoid_derivative(self.Y_hat)
        dZ3 = dLoss_dYhat * dYhat_dZ3
        dW3 = self.A2.T.dot(dZ3)
        db3 = np.sum(dZ3, axis=0, keepdims=True)
        dA2 = dZ3.dot(self.W3.T)

        # Hidden Layer 2 Gradients (Swish)
```

```
        # Pass Z2 and A2 to the derivative function
        dZ2 = dA2 * swish_derivative(self.Z2, self.A2, self.beta)
        dW2 = self.A1.T.dot(dZ2)
        db2 = np.sum(dZ2, axis=0, keepdims=True)
        dA1 = dZ2.dot(self.W2.T)

        # Hidden Layer 1 Gradients (Swish)
        # Pass Z1 and A1 to the derivative function
        dZ1 = dA1 * swish_derivative(self.Z1, self.A1, self.beta)
        dW1 = X.T.dot(dZ1)
        db1 = np.sum(dZ1, axis=0, keepdims=True)

        # Update Weights and Biases
        self.W1 -= learning_rate * dW1
        self.b1 -= learning_rate * db1
        self.W2 -= learning_rate * dW2
        self.b2 -= learning_rate * db2
        self.W3 -= learning_rate * dW3
        self.b3 -= learning_rate * db3

    def train(self, X, Y, epochs, learning_rate):
        history = []
        for epoch in range(epochs):
            Y_hat = self.forward_pass(X)
            loss = np.mean((Y_hat - Y)**2)
            history.append(loss)
            self.backward_pass(X, Y, learning_rate)
            if (epoch + 1) % 100 == 0:
                print(f"Epoch {epoch + 1}/{epochs}, Loss: {loss:.6f}")
        return history

    def predict(self, X):
        return self.forward_pass(X)

    def display_parameters(self):
        print("--- Final Weights and Biases ---")
        print(f"W1 Shape: {self.W1.shape}")
        print(f"b1 Shape: {self.b1.shape}")
        print(f"W2 Shape: {self.W2.shape}")
        print(f"b2 Shape: {self.b2.shape}")
        print(f"W3 Shape: {self.W3.shape}")
        print(f"b3 Shape: {self.b3.shape}")

# --- Main Execution ---
if __name__ == "__main__":
    N = 4
    num_samples = 100
    X_train = np.random.randint(0, 2, size=(num_samples, N))
    Y_train = (np.sum(X_train, axis=1) % 2).reshape(-1, 1)

    print(f"Generated {num_samples} samples with N={N} inputs.")

    n_hidden1 = 16
    n_hidden2 = 8
    n_outputs = 1
    swish_beta = 1.0 # Using standard Swish (beta=1)

    epochs = 3000
    learning_rate = 0.05 # May need tuning

    mlp = MLP_Swish_SigmoidOutput(N, n_hidden1, n_hidden2, n_outputs, beta=swish_beta)
    print(f"\n--- Training MLP with Swish (beta={swish_beta}) Hidden Layers & Sigmoid Output ---")
    loss_history = mlp.train(X_train, Y_train, epochs, learning_rate)

    print("\nTraining complete.")
    mlp.display_parameters()
```

```
predictions = mlp.predict(X_train)
binary_predictions = (predictions > 0.5).astype(int)
accuracy = np.mean(binary_predictions == Y_train) * 100
print(f"\nFinal Training Accuracy: {accuracy:.2f}%")
print(f"Number of steps (epochs): {epochs}")

# Optional: Plot loss history
# import matplotlib.pyplot as plt
# plt.plot(loss_history)
# plt.title(f'Training Loss over Epochs (Swish beta={swish_beta})')
# plt.xlabel('Epoch')
# plt.ylabel('Mean Squared Error Loss')
# plt.grid(True)
# plt.show()
```

Explanation:

1. **`sigmoid` and `sigmoid_derivative`:** Kept as Swish uses sigmoid internally, and it's also used for the output layer.
2. **`swish` and `swish_derivative`:** Implement the Swish function (`x * sigmoid(beta*x)`) and its derivative. The derivative implementation uses the formula `Swish(x) + sigmoid(beta*x)*(1-Swish(x))`, which requires both the input `z` (to calculate `sigmoid(beta*z)`) and the output `a = swish(z)`.
3. **`MLP_Swish_SigmoidOutput` Class:**
   - `__init__`: Initializes the network, storing `beta` (defaulting to 1.0). He initialization is often suitable.
   - `forward_pass`: Uses `swish` (passing `self.beta`) for hidden layers (`A1`, `A2`) and `sigmoid` for the output layer (`Y_hat`).
   - `backward_pass`: Calculates gradients. The key change is using `swish_derivative(self.Z1, self.A1, self.beta)` and `swish_derivative(self.Z2, self.A2, self.beta)` when calculating `dZ1` and `dZ2` respectively, passing both the pre-activation (`Z`) and activation (`A`) values.
   - `train`, `predict`, `display_parameters`: Similar functionality.
4. **Main Block:**
   - Sets up data, network parameters, and defines `swish_beta` (set to 1.0 for standard Swish).
   - Instantiates the `MLP_Swish_SigmoidOutput` class, passing `beta`.
   - Trains the network and evaluates accuracy.

This code implements an MLP using Swish (with beta=1) in the hidden layers and Sigmoid in the output layer, trained with backpropagation, demonstrating the implementation of this modern activation function.

Experiment 25: MLP with Backpropagation (GELU Activation)

Theory

Multi-Layer Perceptron (MLP) & Backpropagation:

Refer to Experiment 19 for the general concepts of MLPs and the Backpropagation algorithm. This experiment modifies the activation function used in the hidden layers.

GELU (Gaussian Error Linear Unit) Activation Function:

- **Formula:** `GELU(x) = 0.5 * x * (1 + tanh(sqrt(2/pi) * (x + 0.044715 * x^3)))`
  - An approximation often used is: `GELU_approx(x) = x * sigmoid(1.702 * x)`
- **Output Range:** Approximately (-0.17, u221e).
- **Derivative (using approximation):** Let `s = sigmoid(1.702 * x)`. Then `GELU_approx'(x) = s + x * (1.702 * s * (1 - s))`
- **Advantages:**
  - Combines properties of ReLU (linearity for positive x), dropout (stochastic regularization implicitly), and zoneout.
  - Smooth and non-monotonic.
  - State-of-the-art results in NLP (e.g., BERT, GPT) and increasingly used in computer vision.
  - Avoids the dying ReLU problem.
- **Disadvantages:**
  - Computationally more expensive than ReLU, Leaky ReLU, ELU, and even Swish (especially the exact form).
- **Use:** Increasingly popular, especially in Transformer-based models and other large networks.

Task Specifics:

- N binary inputs.

- Two hidden layers using **GELU** activation (using the approximation for simplicity).
- One output neuron using **Sigmoid** activation (suitable for binary classification).
- Implement the backpropagation algorithm for training.

Backpropagation with GELU (Approximation):

The core backpropagation process remains the same, but the calculation of gradients `dZ` for the hidden layers uses the GELU approximation's derivative:

1. **Initialization:** Initialize weights (W1, W2, W3) and biases (b1, b2, b3).
2. **Forward Pass:**
    - Hidden Layer 1: `Z1 = X.dot(W1) + b1`, `A1 = GELU_approx(Z1)`
    - Hidden Layer 2: `Z2 = A1.dot(W2) + b2`, `A2 = GELU_approx(Z2)`
    - Output Layer: `Z3 = A2.dot(W3) + b3`, `Output (Y_hat) = sigmoid(Z3)`
3. **Calculate Error:** (e.g., MSE)
4. **Backward Pass (Gradient Calculation):**
    - Calculate `dLoss/dY_hat`.
    - **Output Layer Gradients (Sigmoid):**
        - `dZ3 = dLoss/dY_hat * sigmoid_derivative(Y_hat)`
        - `dW3 = A2.T.dot(dZ3)`
        - `db3 = sum(dZ3, axis=0)`
        - `dA2 = dZ3.dot(W3.T)`
    - **Hidden Layer 2 Gradients (GELU Approx):**
        - `dZ2 = dA2 * GELU_approx_derivative(Z2)` (Note: Derivative depends on Z2)
        - `dW2 = A1.T.dot(dZ2)`
        - `db2 = sum(dZ2, axis=0)`
        - `dA1 = dZ2.dot(W2.T)`
    - **Hidden Layer 1 Gradients (GELU Approx):**
        - `dZ1 = dA1 * GELU_approx_derivative(Z1)` (Note: Derivative depends on Z1)
        - `dW1 = X.T.dot(dZ1)`
        - `db1 = sum(dZ1, axis=0)`
5. **Update Weights and Biases:**
    - `W = W - learning_rate * dW`
    - `b = b - learning_rate * db`
6. **Repeat:** Iterate steps 2-5.

Pseudocode/Algorithm

```
// Define Network Structure (same as Exp 19)
Input: N
Define h1_size, h2_size
Output_size = 1
Learning_rate
Epochs

// Activation Functions
Function sigmoid(x):
  Return 1 / (1 + exp(-x))

Function sigmoid_derivative(x): // x is the output of sigmoid(z)
  Return x * (1 - x)

Function gelu_approx(x):
  Return x * sigmoid(1.702 * x)

Function gelu_approx_derivative(x): // x is the input Z
  s = sigmoid(1.702 * x)
  Return s + x * 1.702 * s * (1 - s)

// Initialize Weights and Biases (He init often suitable)
Function initialize_mlp(N, h1_size, h2_size, output_size):
  // ... (He initialization recommended)
  Return W1, b1, W2, b2, W3, b3

// Training Function
Function train_mlp(X_train, Y_train, N, h1_size, h2_size, output_size, epochs, learning_rate):
  W1, b1, W2, b2, W3, b3 = initialize_mlp(N, h1_size, h2_size, output_size)

  For epoch from 1 to epochs:
    // --- Forward Pass --- (Using GELU Approx for hidden, Sigmoid for output)
    Z1 = X_train.dot(W1) + b1
    A1 = gelu_approx(Z1)
    Z2 = A1.dot(W2) + b2
    A2 = gelu_approx(Z2)
    Z3 = A2.dot(W3) + b3
    Y_hat = sigmoid(Z3) // Output layer uses sigmoid

    // --- Calculate Error (e.g., MSE) ---
    error = Y_train - Y_hat
    loss = mean(error^2)

    // --- Backward Pass ---
    // Output Layer Gradients (Sigmoid)
    dLoss_dYhat = -2 * error / number_of_samples
    dYhat_dZ3 = sigmoid_derivative(Y_hat)
    dZ3 = dLoss_dYhat * dYhat_dZ3

    dW3 = A2.T.dot(dZ3)
    db3 = sum(dZ3, axis=0, keepdims=True)
    dA2 = dZ3.dot(W3.T)

    // Hidden Layer 2 Gradients (GELU Approx)
    // Apply element-wise: dA2 * derivative(Z2)
    dZ2 = dA2 * gelu_approx_derivative(Z2)
    dW2 = A1.T.dot(dZ2)
    db2 = sum(dZ2, axis=0, keepdims=True)
    dA1 = dZ2.dot(W2.T)

    // Hidden Layer 1 Gradients (GELU Approx)
    // Apply element-wise: dA1 * derivative(Z1)
    dZ1 = dA1 * gelu_approx_derivative(Z1)
    dW1 = X_train.T.dot(dZ1)
    db1 = sum(dZ1, axis=0, keepdims=True)
```

```
    // --- Update Weights and Biases ---
    W1 = W1 - learning_rate * dW1
    b1 = b1 - learning_rate * db1
    W2 = W2 - learning_rate * dW2
    b2 = b2 - learning_rate * db2
    W3 = W3 - learning_rate * dW3
    b3 = b3 - learning_rate * db3

    If epoch % 100 == 0:
      Print "Epoch:", epoch, "Loss:", loss

  Return W1, b1, W2, b2, W3, b3

Main:
  // ... (Similar to previous experiments, define N, data, hyperparameters)
  W1, b1, W2, b2, W3, b3 = train_mlp(X_train, Y_train, N, h1_size, h2_size, 1, epochs, learning_rate)
  // ... (Print results)
```

Python Code

```
import numpy as np

# Sigmoid activation function (used by GELU approx and output layer)
def sigmoid(x):
    x_clipped = np.clip(x, -500, 500)
    return 1 / (1 + np.exp(-x_clipped))

def sigmoid_derivative(x): # x is the output of sigmoid
    return x * (1 - x)

# GELU approximation activation function and its derivative
def gelu_approx(x):
    return x * sigmoid(1.702 * x)

def gelu_approx_derivative(x): # x is the input Z
    s = sigmoid(1.702 * x)
    return s + x * 1.702 * s * (1 - s)

# --- Optional: Exact GELU and its derivative (more complex) ---
# from scipy.stats import norm
# def gelu_exact(x):
#     return x * norm.cdf(x)
#
# def gelu_exact_derivative(x):
#     return norm.cdf(x) + x * norm.pdf(x)
# ---------------------------------------------------------------

class MLP_GELU_SigmoidOutput:
    def __init__(self, n_inputs, n_hidden1, n_hidden2, n_outputs):
        self.n_inputs = n_inputs
        self.n_hidden1 = n_hidden1
        self.n_hidden2 = n_hidden2
        self.n_outputs = n_outputs

        # Initialize weights and biases (He initialization often suitable)
        self.W1 = np.random.randn(self.n_inputs, self.n_hidden1) * np.sqrt(2. / self.n_inputs)
        self.b1 = np.zeros((1, self.n_hidden1))
        self.W2 = np.random.randn(self.n_hidden1, self.n_hidden2) * np.sqrt(2. / self.n_hidden1)
        self.b2 = np.zeros((1, self.n_hidden2))
        self.W3 = np.random.randn(self.n_hidden2, self.n_outputs) * 0.1 # Output layer (Sigmoid)
        self.b3 = np.zeros((1, self.n_outputs))

        # Placeholders
        self.Z1, self.A1 = None, None
        self.Z2, self.A2 = None, None
        self.Z3, self.Y_hat = None, None

    def forward_pass(self, X):
        self.Z1 = X.dot(self.W1) + self.b1
        self.A1 = gelu_approx(self.Z1) # Using approximation
        self.Z2 = self.A1.dot(self.W2) + self.b2
        self.A2 = gelu_approx(self.Z2) # Using approximation
        self.Z3 = self.A2.dot(self.W3) + self.b3
        self.Y_hat = sigmoid(self.Z3) # Sigmoid output
        return self.Y_hat

    def backward_pass(self, X, Y, learning_rate):
        m = Y.shape[0]
        error = self.Y_hat - Y
        dLoss_dYhat = 2 * error / m

        # Output Layer Gradients (Sigmoid)
        dYhat_dZ3 = sigmoid_derivative(self.Y_hat)
        dZ3 = dLoss_dYhat * dYhat_dZ3
        dW3 = self.A2.T.dot(dZ3)
```

```
        db3 = np.sum(dZ3, axis=0, keepdims=True)
        dA2 = dZ3.dot(self.W3.T)

        # Hidden Layer 2 Gradients (GELU Approx)
        # Pass Z2 to the derivative function
        dZ2 = dA2 * gelu_approx_derivative(self.Z2)
        dW2 = self.A1.T.dot(dZ2)
        db2 = np.sum(dZ2, axis=0, keepdims=True)
        dA1 = dZ2.dot(self.W2.T)

        # Hidden Layer 1 Gradients (GELU Approx)
        # Pass Z1 to the derivative function
        dZ1 = dA1 * gelu_approx_derivative(self.Z1)
        dW1 = X.T.dot(dZ1)
        db1 = np.sum(dZ1, axis=0, keepdims=True)

        # Update Weights and Biases
        self.W1 -= learning_rate * dW1
        self.b1 -= learning_rate * db1
        self.W2 -= learning_rate * dW2
        self.b2 -= learning_rate * db2
        self.W3 -= learning_rate * dW3
        self.b3 -= learning_rate * db3

    def train(self, X, Y, epochs, learning_rate):
        history = []
        for epoch in range(epochs):
            Y_hat = self.forward_pass(X)
            loss = np.mean((Y_hat - Y)**2)
            history.append(loss)
            self.backward_pass(X, Y, learning_rate)
            if (epoch + 1) % 100 == 0:
                print(f"Epoch {epoch + 1}/{epochs}, Loss: {loss:.6f}")
        return history

    def predict(self, X):
        return self.forward_pass(X)

    def display_parameters(self):
        print("--- Final Weights and Biases ---")
        print(f"W1 Shape: {self.W1.shape}")
        print(f"b1 Shape: {self.b1.shape}")
        print(f"W2 Shape: {self.W2.shape}")
        print(f"b2 Shape: {self.b2.shape}")
        print(f"W3 Shape: {self.W3.shape}")
        print(f"b3 Shape: {self.b3.shape}")

# --- Main Execution ---
if __name__ == "__main__":
    N = 4
    num_samples = 100
    X_train = np.random.randint(0, 2, size=(num_samples, N))
    Y_train = (np.sum(X_train, axis=1) % 2).reshape(-1, 1)

    print(f"Generated {num_samples} samples with N={N} inputs.")

    n_hidden1 = 16
    n_hidden2 = 8
    n_outputs = 1

    epochs = 3000
    learning_rate = 0.05 # May need tuning

    mlp = MLP_GELU_SigmoidOutput(N, n_hidden1, n_hidden2, n_outputs)
    print(f"\n--- Training MLP with GELU (Approximation) Hidden Layers & Sigmoid Output ---")
    loss_history = mlp.train(X_train, Y_train, epochs, learning_rate)
```

```
print("\nTraining complete.")
mlp.display_parameters()

predictions = mlp.predict(X_train)
binary_predictions = (predictions > 0.5).astype(int)
accuracy = np.mean(binary_predictions == Y_train) * 100
print(f"\nFinal Training Accuracy: {accuracy:.2f}%")
print(f"Number of steps (epochs): {epochs}")

# Optional: Plot loss history
# import matplotlib.pyplot as plt
# plt.plot(loss_history)
# plt.title('Training Loss over Epochs (GELU Approx)')
# plt.xlabel('Epoch')
# plt.ylabel('Mean Squared Error Loss')
# plt.grid(True)
# plt.show()
```

Explanation:

1. **`sigmoid` and `sigmoid_derivative`:** Kept as the GELU approximation uses sigmoid internally, and it's also used for the output layer.
2. **`gelu_approx` and `gelu_approx_derivative`:** Implement the common approximation `x * sigmoid(1.702 * x)` and its derivative. The derivative depends only on the input `x` (or `Z` in the network context).
3. **`MLP_GELU_SigmoidOutput` Class:**
   - `__init__`: Standard initialization. He initialization is often suitable.
   - `forward_pass`: Uses `gelu_approx` for hidden layers (`A1`, `A2`) and `sigmoid` for the output layer (`Y_hat`).
   - `backward_pass`: Calculates gradients. The key change is using `gelu_approx_derivative(self.Z1)` and `gelu_approx_derivative(self.Z2)` when calculating `dZ1` and `dZ2` respectively, passing only the pre-activation (`Z`) values.
   - `train`, `predict`, `display_parameters`: Similar functionality.
4. **Main Block:**
   - Sets up data and network parameters.
   - Instantiates the `MLP_GELU_SigmoidOutput` class.
   - Trains the network and evaluates accuracy.

This code implements an MLP using the GELU approximation in the hidden layers and Sigmoid in the output layer, trained with backpropagation. It demonstrates how to integrate this modern activation function, often found in state-of-the-art models.

Experiment 26: MLP with Backpropagation (SELU Activation)

Theory

Multi-Layer Perceptron (MLP) & Backpropagation:

Refer to Experiment 19 for the general concepts of MLPs and the Backpropagation algorithm. This experiment modifies the activation function used in the hidden layers.

SELU (Scaled Exponential Linear Unit) Activation Function:

- **Formula:** `SELU(x) = scale * ELU(x, alpha)`
  - Where `ELU(x, alpha)` is the standard ELU function.
  - `scale` (lambda) u2248 1.0507
  - `alpha` u2248 1.6733
  - Explicitly:
    - `SELU(x) = scale * x` if `x > 0`
    - `SELU(x) = scale * alpha * (exp(x) - 1)` if `x <= 0`
- **Output Range:** (`scale * alpha * (exp(-inf) - 1)`) u2248 -1.758, u221e)
- **Derivative:**
  - `SELU_derivative(x) = scale` if `x > 0`
  - `SELU_derivative(x) = scale * alpha * exp(x)` if `x <= 0` (Note: This can also be written as `SELU(x) + scale * alpha` for `x <= 0`)
- **Advantages:**

    - **Self-Normalizing:** With specific weight initialization (Lecun normal) and network conditions, SELU can push neuron activations towards zero mean and unit variance, helping to prevent vanishing/exploding gradients without explicit batch normalization.
    - Addresses the dying ReLU problem.
- **Disadvantages:**
    - The self-normalizing property requires specific weight initialization (`lecun_normal`) and assumes a certain network architecture (sequential dense layers).
    - Computationally more expensive than ReLU due to the exponential function.
    - The fixed `alpha` and `scale` values are derived theoretically and might not be optimal for all situations.
- **Use:** Designed for deep feedforward networks where self-normalization is desired. Less common than ReLU/variants but powerful when its conditions are met.

Task Specifics:

- N binary inputs.
- Two hidden layers using **SELU** activation.
- One output neuron using **Sigmoid** activation (suitable for binary classification).
- Implement the backpropagation algorithm for training.
- **Crucially:** Use Lecun Normal weight initialization for layers preceding SELU units.

Backpropagation with SELU:

The core backpropagation process remains the same, but the calculation of gradients `dZ` for the hidden layers uses the SELU derivative:

1. **Initialization:** Initialize weights (W1, W2) using Lecun Normal, W3 can use standard init. Initialize biases (b1, b2, b3) to zero.
2. **Forward Pass:**
    - Hidden Layer 1: `Z1 = X.dot(W1) + b1`, `A1 = SELU(Z1)`
    - Hidden Layer 2: `Z2 = A1.dot(W2) + b2`, `A2 = SELU(Z2)`
    - Output Layer: `Z3 = A2.dot(W3) + b3`, `Output (Y_hat) = sigmoid(Z3)`
3. **Calculate Error:** (e.g., MSE)
4. **Backward Pass (Gradient Calculation):**
    - Calculate `dLoss/dY_hat`.
    - **Output Layer Gradients (Sigmoid):**
        - `dZ3 = dLoss/dY_hat * sigmoid_derivative(Y_hat)`
        - `dW3 = A2.T.dot(dZ3)`
        - `db3 = sum(dZ3, axis=0)`
        - `dA2 = dZ3.dot(W3.T)`
    - **Hidden Layer 2 Gradients (SELU):**
        - `dZ2 = dA2 * SELU_derivative(Z2)` (Note: Derivative depends on Z2)
        - `dW2 = A1.T.dot(dZ2)`
        - `db2 = sum(dZ2, axis=0)`
        - `dA1 = dZ2.dot(W2.T)`
    - **Hidden Layer 1 Gradients (SELU):**
        - `dZ1 = dA1 * SELU_derivative(Z1)` (Note: Derivative depends on Z1)
        - `dW1 = X.T.dot(dZ1)`
        - `db1 = sum(dZ1, axis=0)`
5. **Update Weights and Biases:**
    - `W = W - learning_rate * dW`
    - `b = b - learning_rate * db`
6. **Repeat:** Iterate steps 2-5.

Pseudocode/Algorithm

```
// Define Network Structure (same as Exp 19)
Input: N
Define h1_size, h2_size
Output_size = 1
Learning_rate
Epochs

// SELU Constants
Scale = 1.0507
Alpha = 1.6733

// Activation Functions
Function sigmoid(x):
  Return 1 / (1 + exp(-x))

Function sigmoid_derivative(x): // x is the output of sigmoid(z)
  Return x * (1 - x)

Function selu(x, scale, alpha):
  If x > 0:
    Return scale * x
  Else:
    Return scale * alpha * (exp(x) - 1)

Function selu_derivative(x, scale, alpha): // x is the input Z
  If x > 0:
    Return scale
  Else:
    Return scale * alpha * exp(x)

// Initialize Weights and Biases (Lecun Normal for SELU layers)
Function initialize_mlp_selu(N, h1_size, h2_size, output_size):
  // Lecun Normal: stddev = sqrt(1 / n_inputs)
  W1 = random_normal(mean=0, stddev=sqrt(1/N), size=(N, h1_size))
  b1 = zeros((1, h1_size))
  W2 = random_normal(mean=0, stddev=sqrt(1/h1_size), size=(h1_size, h2_size))
  b2 = zeros((1, h2_size))
  // Output layer (Sigmoid) - standard init might be okay
  W3 = random_normal(mean=0, stddev=0.1, size=(h2_size, output_size))
  b3 = zeros((1, output_size))
  Return W1, b1, W2, b2, W3, b3

// Training Function
Function train_mlp(X_train, Y_train, N, h1_size, h2_size, output_size, epochs, learning_rate, scale, alpha):
  W1, b1, W2, b2, W3, b3 = initialize_mlp_selu(N, h1_size, h2_size, output_size)

  For epoch from 1 to epochs:
    // --- Forward Pass --- (Using SELU for hidden, Sigmoid for output)
    Z1 = X_train.dot(W1) + b1
    A1 = selu(Z1, scale, alpha)
    Z2 = A1.dot(W2) + b2
    A2 = selu(Z2, scale, alpha)
    Z3 = A2.dot(W3) + b3
    Y_hat = sigmoid(Z3) // Output layer uses sigmoid

    // --- Calculate Error (e.g., MSE) ---
    error = Y_train - Y_hat
    loss = mean(error^2)

    // --- Backward Pass ---
    // Output Layer Gradients (Sigmoid)
    dLoss_dYhat = -2 * error / number_of_samples
    dYhat_dZ3 = sigmoid_derivative(Y_hat)
    dZ3 = dLoss_dYhat * dYhat_dZ3
```

```
    dW3 = A2.T.dot(dZ3)
    db3 = sum(dZ3, axis=0, keepdims=True)
    dA2 = dZ3.dot(W3.T)

    // Hidden Layer 2 Gradients (SELU)
    // Apply element-wise: dA2 * derivative(Z2, scale, alpha)
    dZ2 = dA2 * selu_derivative(Z2, scale, alpha)
    dW2 = A1.T.dot(dZ2)
    db2 = sum(dZ2, axis=0, keepdims=True)
    dA1 = dZ2.dot(W2.T)

    // Hidden Layer 1 Gradients (SELU)
    // Apply element-wise: dA1 * derivative(Z1, scale, alpha)
    dZ1 = dA1 * selu_derivative(Z1, scale, alpha)
    dW1 = X_train.T.dot(dZ1)
    db1 = sum(dZ1, axis=0, keepdims=True)

    // --- Update Weights and Biases ---
    W1 = W1 - learning_rate * dW1
    b1 = b1 - learning_rate * db1
    W2 = W2 - learning_rate * dW2
    b2 = b2 - learning_rate * db2
    W3 = W3 - learning_rate * dW3
    b3 = b3 - learning_rate * db3

    If epoch % 100 == 0:
      Print "Epoch:", epoch, "Loss:", loss

  Return W1, b1, W2, b2, W3, b3

Main:
  // ... (Similar to previous experiments, define N, data, hyperparameters)
  W1, b1, W2, b2, W3, b3 = train_mlp(X_train, Y_train, N, h1_size, h2_size, 1, epochs, learning_rate, Scale, Alpha)
  // ... (Print results)
```

Python Code

```
import numpy as np

# Sigmoid activation function (for output layer)
def sigmoid(x):
    x_clipped = np.clip(x, -500, 500)
    return 1 / (1 + np.exp(-x_clipped))

def sigmoid_derivative(x): # x is the output of sigmoid
    return x * (1 - x)

# SELU constants
SELU_SCALE = 1.0507009873554804934193349852946
SELU_ALPHA = 1.6732632423543772848170429916717

# SELU activation function and its derivative
def selu(x, scale=SELU_SCALE, alpha=SELU_ALPHA):
    x_clipped = np.clip(x, -500, 500) # Clip input to exp
    return scale * np.where(x > 0, x, alpha * (np.exp(x_clipped) - 1))

def selu_derivative(x, scale=SELU_SCALE, alpha=SELU_ALPHA): # x is the input Z
    x_clipped = np.clip(x, -500, 500) # Clip input to exp
    return scale * np.where(x > 0, 1, alpha * np.exp(x_clipped))

class MLP_SELU_SigmoidOutput:
    def __init__(self, n_inputs, n_hidden1, n_hidden2, n_outputs):
        self.n_inputs = n_inputs
        self.n_hidden1 = n_hidden1
        self.n_hidden2 = n_hidden2
        self.n_outputs = n_outputs

        # Initialize weights with Lecun Normal and biases to zero for SELU layers
        self.W1 = np.random.randn(self.n_inputs, self.n_hidden1) * np.sqrt(1. / self.n_inputs)
        self.b1 = np.zeros((1, self.n_hidden1))
        self.W2 = np.random.randn(self.n_hidden1, self.n_hidden2) * np.sqrt(1. / self.n_hidden1)
        self.b2 = np.zeros((1, self.n_hidden2))

        # Output layer (Sigmoid) - standard init might be okay, or Lecun Normal
        # self.W3 = np.random.randn(self.n_hidden2, self.n_outputs) * np.sqrt(1. / self.n_hidden2)
        self.W3 = np.random.randn(self.n_hidden2, self.n_outputs) * 0.1 # Using small random init
        self.b3 = np.zeros((1, self.n_outputs))

        # Placeholders
        self.Z1, self.A1 = None, None
        self.Z2, self.A2 = None, None
        self.Z3, self.Y_hat = None, None

    def forward_pass(self, X):
        self.Z1 = X.dot(self.W1) + self.b1
        self.A1 = selu(self.Z1)
        self.Z2 = self.A1.dot(self.W2) + self.b2
        self.A2 = selu(self.Z2)
        self.Z3 = self.A2.dot(self.W3) + self.b3
        self.Y_hat = sigmoid(self.Z3) # Sigmoid output
        return self.Y_hat

    def backward_pass(self, X, Y, learning_rate):
        m = Y.shape[0]
        error = self.Y_hat - Y
        dLoss_dYhat = 2 * error / m

        # Output Layer Gradients (Sigmoid)
        dYhat_dZ3 = sigmoid_derivative(self.Y_hat)
        dZ3 = dLoss_dYhat * dYhat_dZ3
        dW3 = self.A2.T.dot(dZ3)
        db3 = np.sum(dZ3, axis=0, keepdims=True)
```

```
        dA2 = dZ3.dot(self.W3.T)

        # Hidden Layer 2 Gradients (SELU)
        # Pass Z2 to the derivative function
        dZ2 = dA2 * selu_derivative(self.Z2)
        dW2 = self.A1.T.dot(dZ2)
        db2 = np.sum(dZ2, axis=0, keepdims=True)
        dA1 = dZ2.dot(self.W2.T)

        # Hidden Layer 1 Gradients (SELU)
        # Pass Z1 to the derivative function
        dZ1 = dA1 * selu_derivative(self.Z1)
        dW1 = X.T.dot(dZ1)
        db1 = np.sum(dZ1, axis=0, keepdims=True)

        # Update Weights and Biases
        self.W1 -= learning_rate * dW1
        self.b1 -= learning_rate * db1
        self.W2 -= learning_rate * dW2
        self.b2 -= learning_rate * db2
        self.W3 -= learning_rate * dW3
        self.b3 -= learning_rate * db3

    def train(self, X, Y, epochs, learning_rate):
        history = []
        for epoch in range(epochs):
            Y_hat = self.forward_pass(X)
            loss = np.mean((Y_hat - Y)**2)
            history.append(loss)
            self.backward_pass(X, Y, learning_rate)
            if (epoch + 1) % 100 == 0:
                print(f"Epoch {epoch + 1}/{epochs}, Loss: {loss:.6f}")
        return history

    def predict(self, X):
        return self.forward_pass(X)

    def display_parameters(self):
        print("--- Final Weights and Biases ---")
        print(f"W1 Shape: {self.W1.shape}")
        print(f"b1 Shape: {self.b1.shape}")
        print(f"W2 Shape: {self.W2.shape}")
        print(f"b2 Shape: {self.b2.shape}")
        print(f"W3 Shape: {self.W3.shape}")
        print(f"b3 Shape: {self.b3.shape}")

# --- Main Execution ---
if __name__ == "__main__":
    N = 4
    num_samples = 100
    X_train = np.random.randint(0, 2, size=(num_samples, N))
    Y_train = (np.sum(X_train, axis=1) % 2).reshape(-1, 1)

    print(f"Generated {num_samples} samples with N={N} inputs.")

    n_hidden1 = 16
    n_hidden2 = 8
    n_outputs = 1

    epochs = 3000
    learning_rate = 0.05 # May need tuning

    mlp = MLP_SELU_SigmoidOutput(N, n_hidden1, n_hidden2, n_outputs)
    print(f"\n--- Training MLP with SELU Hidden Layers & Sigmoid Output ---")
    loss_history = mlp.train(X_train, Y_train, epochs, learning_rate)
```

```
    print("\nTraining complete.")
    mlp.display_parameters()

    predictions = mlp.predict(X_train)
    binary_predictions = (predictions > 0.5).astype(int)
    accuracy = np.mean(binary_predictions == Y_train) * 100
    print(f"\nFinal Training Accuracy: {accuracy:.2f}%")
    print(f"Number of steps (epochs): {epochs}")

    # Optional: Plot loss history
    # import matplotlib.pyplot as plt
    # plt.plot(loss_history)
    # plt.title('Training Loss over Epochs (SELU)')
    # plt.xlabel('Epoch')
    # plt.ylabel('Mean Squared Error Loss')
    # plt.grid(True)
    # plt.show()
```

Explanation:

1. **`sigmoid` and `sigmoid_derivative`:** Kept for the output layer.
2. **`SELU_SCALE`, `SELU_ALPHA`:** Constants defined for SELU.
3. **`selu` and `selu_derivative`:** Implement the SELU function and its derivative using the defined constants. The derivative depends only on the input `x` (or `Z`).
4. **`MLP_SELU_SigmoidOutput` Class:**
   - `__init__`: **Crucially**, initializes weights `W1` and `W2` using **Lecun Normal initialization** (`stddev = sqrt(1 / n_inputs)`). Biases are initialized to zero. The output layer weight `W3` can use a different initialization as it's followed by Sigmoid.
   - `forward_pass`: Uses `selu` for hidden layers (`A1`, `A2`) and `sigmoid` for the output layer (`Y_hat`).
   - `backward_pass`: Calculates gradients. The key change is using `selu_derivative(self.Z1)` and `selu_derivative(self.Z2)` when calculating `dZ1` and `dZ2` respectively, passing only the pre-activation (`Z`) values.
   - `train`, `predict`, `display_parameters`: Similar functionality.
5. **Main Block:**
   - Sets up data and network parameters.
   - Instantiates the `MLP_SELU_SigmoidOutput` class.
   - Trains the network and evaluates accuracy.

This code implements an MLP using SELU in the hidden layers and Sigmoid in the output layer, trained with backpropagation. It highlights the specific weight initialization (Lecun Normal) required for SELU to potentially achieve its self-normalizing properties.